



© 2019 Pearson Italia, Milano-Torino

Authorized translation from the English language edition, entitled *Operating System Concepts*, 10th Edition, (9781119124825/1119124824) by Abraham Silberschatz, Peter B. Galvin and Greg Gagne, published by John Wiley & Sons, Inc, Copyright © 2018.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Italian language edition published by Pearson Italia S.p.A., Copyright © 2019.

Le informazioni contenute in questo libro sono state verificate e documentate con la massima cura possibile. Nessuna responsabilità derivante dal loro utilizzo potrà venire imputata agli Autori, a Pearson Italia S.p.A. o a ogni persona e società coinvolta nella creazione, produzione e distribuzione di questo libro.

Per i passi antologici, per le citazioni, per le riproduzioni grafiche, cartografiche e fotografiche appartenenti alla proprietà di terzi, inseriti in quest'opera, l'editore è a disposizione degli aventi diritto non potuti reperire nonché per eventuali non volute omissioni e/o errori di attribuzione nei riferimenti.

Edizione italiana a cura di: Riccardo Melen

Traduzione: Pietro Codara

Copy editing: Donatella Pepe

Grafica di copertina: Giulia Boffi

Immagine di copertina: AllNikArt/Shutterstock

Produzione digitale: Digitaltypes - Milano

Tutti i marchi citati nel testo sono di proprietà dei loro detentori.

ISBN: 9788891904560

Sommario

Copertina

Colophon

Sommario

PREFAZIONE

Prefazione all'edizione italiana

Prefazione

PARTE I Generalità

CAPITOLO 1 Introduzione

- 1.1 Che cosa fa un sistema operativo
- 1.2 Organizzazione di un sistema elaborativo
- 1.3 Architettura degli elaboratori
- 1.4 Attività del sistema operativo
- 1.5 Gestione delle risorse
- 1.6 Sicurezza e protezione
- 1.7 Virtualizzazione
- 1.8 Sistemi distribuiti
- 1.9 Strutture dati del kernel
- 1.10 Ambienti d'elaborazione
- 1.11 Sistemi operativi liberi e open-source
- 1.12 Sommario

Esercizi di ripasso

Note bibliografiche

Bibliografia

Esercizi

CAPITOLO 2 Strutture dei sistemi operativi

- 2.1 Servizi di un sistema operativo
 - 2.2 Interfaccia con l'utente del sistema operativo
 - 2.3 Chiamate di sistema
 - 2.4 Servizi di sistema
 - 2.5 Linker e loader
 - 2.6 Perché le applicazioni dipendono dal sistema operativo
 - 2.7 Progettazione e realizzazione di un sistema operativo
 - 2.8 Struttura del sistema operativo
 - 2.9 Generare e avviare un sistema operativo
 - 2.10 Debugging dei sistemi operativi
 - 2.11 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

PARTE II Gestione dei processi**CAPITOLO 3 Processi**

- 3.1 Concetto di processo
 - 3.2 Scheduling dei processi
 - 3.3 Operazioni sui processi
 - 3.4 Comunicazione tra processi
 - 3.5 IPC in sistemi a memoria condivisa
 - 3.6 IPC in sistemi a scambio di messaggi
 - 3.7 Esempi di sistemi IPC
 - 3.8 Comunicazione nei sistemi client-server
 - 3.9 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 4 Thread e concorrenza

- 4.1 Introduzione
 - 4.2 Programmazione multicore
 - 4.3 Modelli di supporto al multithreading
 - 4.4 Librerie dei thread
 - 4.5 Threading隐式
 - 4.6 Problematiche di programmazione multithread
 - 4.7 Esempi di sistemi operativi
 - 4.8 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 5 Scheduling della CPU

- 5.1 Concetti fondamentali
 - 5.2 Criteri di scheduling
 - 5.3 Algoritmi di scheduling
 - 5.4 Scheduling dei thread
 - 5.5 Scheduling per sistemi multiprocessore
 - 5.6 Scheduling real-time della CPU
 - 5.7 Esempi di sistemi operativi
 - 5.8 Valutazione degli algoritmi
 - 5.9 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

PARTE III Sincronizzazione dei processi**CAPITOLO 6 Strumenti di sincronizzazione**

- 6.1 Introduzione
 - 6.2 Problema della sezione critica
 - 6.3 Soluzione di Peterson
 - 6.4 Supporto hardware per la sincronizzazione
 - 6.5 Lock mutex
 - 6.6 Semafori
 - 6.7 Monitor
 - 6.8 Liveness
 - 6.9 Valutazione delle soluzioni
 - 6.10 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 7 Esempi di sincronizzazione

- 7.1 Classici problemi di sincronizzazione

- 7.2 Sincronizzazione all'interno del kernel
 - 7.3 Sincronizzazione POSIX
 - 7.4 Sincronizzazione in Java
 - 7.5 Approcci alternativi
 - 7.6 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 8 Stallo dei processi

- 8.1 Modello di sistema
 - 8.2 Situazioni di stallo in applicazioni multithread
 - 8.3 Caratterizzazione delle situazioni di stallo
 - 8.4 Metodi per la gestione delle situazioni di stallo
 - 8.5 Prevenzione delle situazioni di stallo
 - 8.6 Evitare le situazioni di stallo
 - 8.7 Rilevamento delle situazioni di stallo
 - 8.8 Ripristino da situazioni di stallo
 - 8.9 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

PARTE IV Gestione della memoria

CAPITOLO 9 Memoria centrale

- 9.1 Introduzione
 - 9.2 Allocazione contigua della memoria
 - 9.3 Paginazione
 - 9.4 Struttura della tabella delle pagine
 - 9.5 Avvicendamento dei processi (swapping)
 - 9.6 Esempio: le architetture Intel a 32 e 64 bit
 - 9.7 Esempio: architettura ARMv8
 - 9.8 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 10 Memoria virtuale

- 10.1 Introduzione
 - 10.2 Paginazione su richiesta
 - 10.3 Copiatura su scrittura
 - 10.4 Sostituzione delle pagine
 - 10.5 Allocazione dei frame
 - 10.6 Thrashing
 - 10.7 Compressione della memoria
 - 10.8 Allocazione di memoria del kernel
 - 10.9 Altre considerazioni
 - 10.10 Esempi di sistemi operativi
 - 10.11 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

PARTE V Gestione della memoria di massa

CAPITOLO 11 Memoria di massa

- 11.1 Struttura dei dispositivi di memorizzazione
 - 11.2 Scheduling dei dischi rigidi
 - 11.3 Scheduling di dispositivi NVM
 - 11.4 Rilevamento e correzione di errori
 - 11.5 Gestione delle unità di memoria secondaria
 - 11.6 Gestione dell'area d'avvicendamento
 - 11.7 Connessione dei dispositivi di memorizzazione
 - 11.8 Strutture RAID
 - 11.9 Sommario
- Esercizi di ripasso
Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 12 Sistemi di I/O

- 12.1 Introduzione
- 12.2 Hardware di I/O
- 12.3 Interfaccia di I/O delle applicazioni
- 12.4 Sottosistema di I/O del kernel
- 12.5 Trasformazione delle richieste di I/O in operazioni hardware
- 12.6 STREAMS
- 12.7 Prestazioni

- 12.8 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

PARTE VI File system

CAPITOLO 13 Interfaccia del file system

- 13.1 Concetto di file
- 13.2 Metodi d'accesso
- 13.3 Struttura delle directory
- 13.4 Protezione
- 13.5 File mappati in memoria
- 13.6 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

CAPITOLO 14 Realizzazione del file system

- 14.1 Struttura del file system
- 14.2 Operazioni del file system
- 14.3 Realizzazione delle directory
- 14.4 Metodi di allocazione
- 14.5 Gestione dello spazio libero
- 14.6 Efficienza e prestazioni
- 14.7 Ripristino
- 14.8 Esempio: il file system WAFL
- 14.9 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

CAPITOLO 15 Dettagli interni del file system

- 15.1 I file system
- 15.2 Montaggio di un file system
- 15.3 Partizioni e montaggio
- 15.4 Condivisione di file
- 15.5 File system virtuali
- 15.6 File system remoti
- 15.7 Semantica della coerenza
- 15.8 NFS
- 15.9 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

PARTE VII Sicurezza e protezione

CAPITOLO 16 Sicurezza

- 16.1 Il problema della sicurezza
- 16.2 Minacce legate ai programmi
- 16.3 Minacce relative al sistema e alla rete
- 16.4 Crittografia come strumento per la sicurezza
- 16.5 Autenticazione degli utenti
- 16.6 Realizzazione delle misure di sicurezza
- 16.7 Un esempio: Windows 10
- 16.8 Sommario
- Note bibliografiche
- Bibliografia
- Esercizi

CAPITOLO 17 Protezione

- 17.1 Scopi della protezione
- 17.2 Principi di protezione
- 17.3 Anelli di protezione
- 17.4 Domini di protezione
- 17.5 Matrice d'accesso
- 17.6 Realizzazione della matrice d'accesso
- 17.7 Revoca dei diritti d'accesso
- 17.8 Controllo dell'accesso basato sui ruoli
- 17.9 Controllo obbligatorio dell'accesso (MAC)
- 17.10 Sistemi basati su abilitazioni
- 17.11 Altri metodi per il miglioramento della protezione
- 17.12 Protezione basata sul linguaggio
- 17.13 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

CAPITOLO 18 Macchine virtuali

- 18.1 Introduzione
 - 18.2 Storia
 - 18.3 Vantaggi e caratteristiche
 - 18.4 Blocchi costituenti
 - 18.5 Tipologie di macchine virtuali e loro implementazioni
 - 18.6 Virtualizzazione e componenti dei sistemi operativi
 - 18.7 Esempi
 - 18.8 La ricerca sulla virtualizzazione
 - 18.9 Sommario
- Note bibliografiche
Bibliografia
Esercizi

CAPITOLO 19 Reti e sistemi distribuiti

- 19.1 Vantaggi dei sistemi distribuiti
- 19.2 Struttura delle reti
- 19.3 Struttura della comunicazione
- 19.4 Sistemi operativi di rete e distribuiti
- 19.5 Progettazione di sistemi distribuiti
- 19.6 File system distribuiti
- 19.7 Naming e trasparenza nei DFS
- 19.8 Accesso ai file remoti
- 19.9 Considerazioni finali sui file system distribuiti
- 19.10 Sommario

Esercizi di ripasso

Note bibliografiche

Bibliografia

Esercizi

CAPITOLO 20 Linux

- 20.1 Storia di Linux
- 20.2 Principi di progettazione
- 20.3 Moduli del kernel
- 20.4 Gestione dei processi
- 20.5 Scheduling
- 20.6 Gestione della memoria
- 20.7 File system
- 20.8 Input e output
- 20.9 Comunicazione fra processi
- 20.11 Sicurezza
- 20.12 Sommario

Esercizi di ripasso

Note bibliografiche

Bibliografia

Esercizi

CAPITOLO 21 Windows 10

- 21.1 Storia
- 21.2 Principi di progettazione
- 21.3 Componenti del sistema
- 21.4 Terminal services e cambio rapido utente
- 21.5 File system
- 21.6 Servizi di rete
- 21.7 Interfaccia di programmazione
- 21.8 Sommario

Esercizi di ripasso

Note bibliografiche

Bibliografia

Esercizi

APPENDICE A Prospettiva storica

- A.1 Migrazione delle caratteristiche
 - A.2 Primi sistemi
 - A.3 Atlas
 - A.4 XDS-940
 - A.5 THE
 - A.6 RC 4000
 - A.7 CTSS
 - A.8 MULTICS
 - A.9 OS/360 di IBM
 - A.10 TOPS-20
 - A.11 CP/M e MS/DOS
 - A.12 Macintosh OS e Windows
 - A.13 Mach
 - A.14 Sistemi basati su abilitazioni
 - A.15 Altri sistemi
- Note bibliografiche
Bibliografia
Esercizi

APPENDICE B Windows 7

- B.1 Storia
- B.2 Principi di progettazione
- B.3 Componenti del sistema
- B.4 Terminal services e cambio rapido utente
- B.5 File system
- B.6 Servizi di rete
- B.7 Interfaccia di programmazione
- B.8 Sommario
- Esercizi di ripasso
- Note bibliografiche
- Bibliografia
- Esercizi

PREFAZIONE

Prefazione all'edizione italiana

Nella decima edizione di questo classico testo sui sistemi operativi gli Autori hanno profuso un grande impegno nel mantenere i contenuti del libro al passo con un'evoluzione tecnologica che è diventata molto rapida: il testo presenta infatti numerosi cambiamenti e ammodernamenti rispetto alla precedente edizione.

In particolare è stata ampliata la trattazione delle tematiche relative ai sistemi per dispositivi mobili e alla virtualizzazione (con un capitolo che comprende anche i container applicativi), sono stati approfonditi vari aspetti legati alle architetture multicore ed è stata data una giusta enfasi alle memorie di massa non volatili. Un nuovo capitolo dedicato a Windows 10 e l'aggiornamento del capitolo dedicato a Linux costituiscono due esempi molto utili e dettagliati, e permettono al lettore di capire a fondo come i principi generali illustrati nel testo si applichino nei sistemi moderni.

Gli Autori si sono anche impegnati in un'opera di miglioramento capillare dei contenuti già presenti nelle scorse edizioni (per esempio quelli del capitolo su reti e sistemi distribuiti) e hanno inoltre sfondato il libro di argomenti ormai di interesse essenzialmente storico.

Bisogna sottolineare infine che una gran parte del valore del libro consiste nel ricchissimo corredo di esercizi, problemi e progetti di programmazione, reperibili sulla piattaforma MyLab. In particolare i progetti vengono sviluppati indirizzando lo studente nei vari passi di sviluppo, fino a ottenere risultati ragionevolmente complessi e particolarmente efficaci per chiarire i concetti spiegati nel testo.

Come nell'edizione precedente, la traduzione italiana cerca di impiegare una terminologia moderna e coerente con quella normalmente utilizzata nella pratica professionale dell'informatica.

Prof. Riccardo Melen

Dipartimento di Informatica,

Sistemistica e Comunicazione

Università di Milano Bicocca

Prefazione

Così come i sistemi operativi sono una parte essenziale dei sistemi elaborativi, un corso sui sistemi operativi è una parte essenziale di un percorso di studio d'informatica. Con i calcolatori ormai presenti praticamente in ogni ambito del nostro quotidiano, dai sistemi integrati nelle automobili ai più complessi e raffinati strumenti di pianificazione impiegati dagli enti governativi e dalle grandi multinazionali, la loro evoluzione ha ormai assunto un ritmo vertiginoso. D'altra parte, i concetti fondamentali restano molto chiari; su di essi si fonda la trattazione svolta in questo libro.

Il testo è stato concepito e scritto per un corso introduttivo sui sistemi operativi, di cui fornisce una chiara descrizione dei concetti di base. Gli Autori si augurano che anche i professionisti del settore giudichino il libro un'utile guida. Prerequisiti essenziali per la comprensione del testo da parte del lettore sono la familiarità con l'organizzazione di un calcolatore, la conoscenza di un linguaggio di programmazione ad alto livello, come il C o Java e delle principali strutture dati. Nel Capitolo 1 si introducono le nozioni riguardanti l'hardware dei calcolatori necessarie alla comprensione dei sistemi operativi. Nello stesso capitolo offriamo inoltre una panoramica delle fondamentali strutture dati prevalentemente utilizzate nei sistemi operativi. Benché siano scritti prevalentemente in C, o anche in Java, gli algoritmi analizzati nel testo sono facilmente comprensibili anche senza una conoscenza approfondita di questi linguaggi di programmazione.

I concetti sono esposti attraverso descrizioni intuitive, che evidenziano i risultati importanti sul piano teorico senza, tuttavia, ricorrere a dimostrazioni formali. Le note bibliografiche rinviano il lettore agli articoli di ricerca in cui sono stati presentati e dimostrati, per la prima volta, tali risultati, e contengono anche indicazioni utili a reperire materiale aggiornato di approfondimento. In luogo di prove formali, abbiamo utilizzato – a corredo dei risultati – grafici ed esempi che ne illustrano la validità.

I concetti fondamentali e gli algoritmi trattati in questo testo spesso si basano su quelli impiegati sia nei sistemi operativi commerciali sia in quelli open-source. Si è in ogni modo cercato di presentare questi concetti e algoritmi in una forma generale, non legata a un particolare sistema operativo. Nonostante ciò, nel libro sono presenti molti esempi che riguardano i sistemi operativi più diffusi, oltre che i più innovativi, tra cui Linux, Microsoft Windows, Apple macos (il nome originale os x è stato cambiato nel 2016 in modo da allinearsi allo schema di nomenclatura degli altri prodotti Apple) e Solaris. Sono inclusi anche alcuni esempi riguardanti Android e ios, i due sistemi operativi attualmente dominanti nel settore dei dispositivi mobili.

La struttura di questo testo rispecchia la lunga esperienza degli Autori, in qualità di docenti, nei rispettivi corsi di sistemi operativi. Nel redigere il testo si è tenuto conto delle valutazioni dei revisori del testo, nonché dei commenti e dei suggerimenti inviati dai lettori delle precedenti edizioni e dai nostri studenti. Questa decima edizione, inoltre, recepisce le linee guida relative all'area dei sistemi operativi presenti nel *Computer Science Curricula 2013*, la più recente raccolta di linee guida per gli insegnamenti di informatica a livello undergraduate pubblicata dalla ieee Computing Society e dall'Association for Computing Machinery (acm).

Contenuti del libro

Il testo è suddiviso in dieci parti principali.

- **Generalità.** Nei Capitoli 1 e 2 si spiega che cosa sono i sistemi operativi, che cosa fanno e come sono *progettati e realizzati*, attraverso un'analisi delle comuni caratteristiche dei sistemi operativi e di quei servizi che un sistema operativo deve fornire agli utenti. Vengono considerati sia sistemi operativi per pc e server sia sistemi operativi per dispositivi mobili. La presentazione è di tipo descrittivo e mira a motivare lo studio di queste tematiche. Essendo privi di riferimenti al funzionamento interno, questi capitoli sono consigliabili a chiunque voglia sapere che cos'è un sistema operativo, senza soffermarsi sui dettagli degli algoritmi che ne controllano il funzionamento.
- **Gestione dei processi.** Nei Capitoli dal 3 al 5 si descrivono i concetti di processo e concorrenza che costituiscono il fondamento dei moderni sistemi operativi. Per *processo* s'intende l'unità di lavoro di un sistema, che consiste quindi in un insieme di processi eseguiti in modo *concorrente*, alcuni dei quali eseguono il codice del sistema operativo, mentre i rimanenti eseguono il codice utente. In questi capitoli si affrontano i differenti metodi impiegati per lo scheduling e la comunicazione tra processi. Tra questi argomenti è compresa un'analisi dei thread e un esame dei temi relativi ai sistemi multicore e alla programmazione parallela.
- **Sincronizzazione dei processi.** I Capitoli dal 6 all'8 riguardano i metodi per la sincronizzazione dei processi e la gestione dei deadlock. Poiché abbiamo esteso la copertura della sincronizzazione dei processi, abbiamo diviso il vecchio capitolo sulla sincronizzazione dei processi in due capitoli separati: Capitolo 6, Strumenti di sincronizzazione e Capitolo 7, Esempi di sincronizzazione.
- **Gestione della memoria.** Nei Capitoli 9 e 10 è trattata la gestione della memoria centrale durante l'esecuzione di un processo. Al fine di migliorare sia l'utilizzo della cpu sia la velocità di risposta ai propri utenti, un calcolatore deve essere in grado di mantenere contemporaneamente più processi in memoria. Esistono molti schemi per la gestione della memoria centrale; questi schemi riflettono diverse strategie di gestione della memoria e l'efficacia dei diversi algoritmi dipende dal particolare contesto in cui si applicano.
- **Gestione della memoria secondaria.** I Capitoli 11 e 12 spiegano come gli elaboratori moderni gestiscano la memoria di massa e l'i/o. Dal momento che i dispositivi di i/o collegabili a un calcolatore sono del più vario genere è necessario che il sistema operativo fornisca funzionalità ampie e diversificate alle applicazioni, cosicché queste possano tenere sotto controllo i dispositivi in ogni loro aspetto. La trattazione dell'i/o mira ad approfondire progettazione, interfacce, strutture e funzioni interne del sistema. Per molti aspetti, i dispositivi di i/o sono i più lenti fra i componenti principali del calcolatore: vengono dunque analizzati i problemi che derivano da questo collo di bottiglia nelle prestazioni.
- **File system.** Nei Capitoli dal 13 al 15 si spiega come gli elaboratori moderni gestiscano i file system. Il file system fornisce il meccanismo per la memorizzazione e l'accesso ai dati e ai programmi che risiedono sulla memoria di massa. Vengono descritti i fondamentali algoritmi interni e le strutture di gestione della memoria di massa e viene fornita una solida conoscenza pratica degli algoritmi utilizzati, analizzandone le proprietà, i vantaggi e gli svantaggi.
- **Sicurezza e protezione.** I Capitoli 16 e 17 illustrano i meccanismi necessari a garantire sicurezza e protezione dei sistemi elaborativi. Tutti i processi di un sistema operativo devono essere reciprocamente protetti; a tale scopo bisogna garantire che solo i processi autorizzati dal sistema operativo possano impiegare le risorse del sistema, come file, segmenti di memoria, cpu e altre risorse. La protezione è il meccanismo attraverso il quale il sistema controlla l'accesso alle risorse da parte di programmi,

processi o utenti. Tale meccanismo deve fornire un metodo per definire i controlli e i vincoli ai quali gli utenti vanno sottoposti, e i mezzi per realizzarli. La sicurezza, invece, consiste nel proteggere sia le informazioni memorizzate all'interno del sistema (dati e codice) sia le risorse fisiche del sistema elaborativo da accessi non autorizzati, tentativi di alterazione o distruzione e dall'introduzione accidentale di incongruenze nel funzionamento.

- **Argomenti avanzati.** I Capitoli 18 e 19 trattano le macchine virtuali e i sistemi distribuiti. Il Capitolo 18 fornisce una panoramica delle macchine virtuali e della loro relazione con i sistemi operativi moderni; vengono inoltre illustrate le tecniche hardware e software che rendono possibile la virtualizzazione. Il Capitolo 19 fornisce una panoramica delle reti di calcolatori e dei sistemi distribuiti, con particolare attenzione a Internet e ai protocolli tcp/ip.
- **Casi di studio.** I Capitoli 20 e 21 presentano casi di studio dettagliati di due sistemi operativi reali – Linux e Windows 10. Sono reperibili sulla piattaforma MyLab.
- **Appendici.** L'Appendice A descrive alcuni sistemi operativi storici non più utilizzati. Le Appendici da B a D descrivono in maggiore dettaglio tre sistemi operativi del passato: Windows 7, bsd e Mach. Tutte le appendici sono reperibili sulla piattaforma MyLab.

Ambienti di programmazione

Il testo fornisce diversi programmi di esempio scritti in C e in Java, concepiti per i seguenti ambienti di programmazione.

- **posix.** La sigla posix (ossia *interfaccia portabile del sistema operativo*) rappresenta una serie di standard creati essenzialmente per sistemi operativi della famiglia unix. Sebbene il sistema operativo Windows possa eseguire alcuni programmi posix, la nostra trattazione è prettamente incentrata sui sistemi unix e Linux. I sistemi compatibili con posix devono implementare lo standard di base (posix.1); è questo il caso di Linux e macos. Esistono poi numerose estensioni degli standard di base; tra queste, l'estensione real-time (posix1.b) e quella per i thread (posix1.c, meglio nota come Pthreads). Vari programmi, scritti in C, fungono da esempio per chiarire non solo il funzionamento dell'api posix di base, ma anche quello di Pthreads e dello standard per la programmazione real-time. Questi programmi dimostrativi sono stati testati sulle versioni 4.4 di Linux, e 10.11 di macos con l'ausilio del compilatore `gcc`.
- **Java.** Java è un linguaggio di programmazione largamente utilizzato, dotato di una ricca api, oltre che di funzionalità integrate per la programmazione concorrente e parallela. I programmi Java sono eseguibili da qualsiasi sistema operativo, purché vi sia installata una macchina virtuale Java (o jvm). Vengono illustrati vari concetti relativi ai sistemi operativi e alle architetture di rete, grazie a programmi testati con la versione 1.8 del Java Development Kit (jdk).
- **Sistemi Windows.** Il più importante ambiente di programmazione per i sistemi Windows è l'api di Windows, che dispone di un insieme completo di funzioni per la gestione di processi, thread, memoria e periferiche. Per illustrare l'uso di questa api viene fornito un piccolo numero di programmi in C. I programmi dimostrativi sono stati testati su un sistema Windows 10.

Abbiamo scelto i tre sudetti ambienti di programmazione poiché li riteniamo i più adatti a rappresentare i due modelli di sistemi operativi più diffusi, Windows e Linux/unix, al pari dell'ambiente Java, ampiamente diffuso. I programmi dimostrativi, scritti prevalentemente in C, presuppongono una certa dimestichezza da parte dei lettori con tale linguaggio; i lettori con buona padronanza di Java, oltre che del linguaggio C, dovrebbero comprendere senza problemi la maggior parte dei programmi.

In alcuni casi – come per la creazione dei thread – ci serviamo di tutti e tre gli ambienti di programmazione per illustrare un dato concetto, invitando il lettore a confrontare le diverse soluzioni delle tre librerie, in riferimento al medesimo problema. In altri casi, soltanto una delle api è utilizzata per esemplificare un concetto. Per descrivere la memoria condivisa, per esempio, ricorriamo esclusivamente alla api di posix; la programmazione con le socket tcp/ip è illustrata tramite la api di Java.

Macchina virtuale Linux

Per aiutare gli studenti nella comprensione del sistema Linux mettiamo a disposizione con questo testo una macchina virtuale Linux con distribuzione Ubuntu. La macchina virtuale, scaricabile dal sito di riferimento del testo originale (<http://www.os-book.com>), fornisce anche un ambiente di sviluppo con i compilatori `gcc` e Java. La maggior parte degli esercizi di programmazione contenuta nel libro può essere risolta su questa macchina virtuale, a eccezione di quelli che richiedono le api di Windows. La macchina virtuale può essere installata ed eseguita su qualsiasi sistema host che supporti il software di virtualizzazione VirtualBox; fra questi sistemi vi sono Windows 10, Linux e macos.

Decima edizione

Nella stesura di questa decima edizione del testo siamo stati guidati dal recente sviluppo di quattro settori fondamentali, in grado di influenzare i sistemi operativi:

1. sistemi operativi per dispositivi mobili;
2. sistemi multicore;
3. virtualizzazione;
4. memoria secondaria non volatile.

Per dare enfasi ai suddetti argomenti sono state fatte integrazioni rilevanti in questa nuova edizione. Per esempio abbiamo ampliato in maniera significativa la trattazione dei sistemi operativi mobili Android e ios, così come dell'architettura armv8, che domina il mercato dei dispositivi mobili. Abbiamo anche esteso la trattazione dei sistemi multicore, comprendendovi la descrizione di api che supportano la concorrenza e il parallelismo. I dispositivi di memoria non volatile come gli ssd vengono trattati allo stesso livello delle unità a disco rigido nei capitoli che discutono di i/o, memoria di massa e file system.

Molti dei nostri lettori hanno espresso il loro interesse per una trattazione più estesa Java e abbiamo quindi fornito ulteriori esempi Java in questa edizione.

Inoltre, abbiamo riscritto del materiale in quasi tutti i capitoli aggiornando il materiale precedente e rimuovendo materiale che non è più interessante o rilevante. Abbiamo riordinato molti capitoli e, in alcuni casi, abbiamo spostato paragrafi da un capitolo all'altro. Abbiamo anche notevolmente rivisto le figure, creandone di nuove e modificando molte figure esistenti.

Cambiamenti principali

L'aggiornamento della decima edizione comprende molti più materiali rispetto agli aggiornamenti precedenti, sia in termini di contenuti sia di materiali di supporto. Nel seguito, forniamo una breve descrizione delle principali modifiche ai contenuti in ciascun capitolo.

- **Capitolo 1: Introduzione** include una copertura aggiornata dei sistemi multicore, nonché una nuova trattazione dei sistemi numa e dei cluster Hadoop. Il vecchio materiale è stato aggiornato e sono state aggiunte nuove motivazioni per lo studio dei sistemi operativi.
- **Capitolo 2: Strutture dei sistemi operativi** contiene una discussione profondamente rivista sulla progettazione e sull'implementazione dei sistemi operativi. Abbiamo aggiornato il trattamento di Android e iOS e abbiamo rivisto la nostra trattazione del processo di avvio del sistema con un focus su grub per i sistemi Linux. È inclusa anche una nuova descrizione del sottosistema Windows per Linux. Abbiamo aggiunto nuovi paragrafi su linker e loader e analizziamo perché le applicazioni siano spesso specifiche per un sistema operativo. Infine, abbiamo aggiunto una discussione sul set di strumenti di debug BCC.
- **Capitolo 3: Processi** semplifica la discussione sullo scheduling in modo da includere ora solo problemi di scheduling della CPU. La nuova trattazione descrive il layout di memoria di un programma C, la gerarchia dei processi Android, il trasferimento dei messaggi Mach e le RPC Android. Abbiamo anche sostituito la descrizione del tradizionale processo `init` di Unix/Linux con la descrizione di `systemd`.
- **Capitolo 4: Thread e concorrenza** estende la trattazione del supporto per la programmazione concorrente e parallela a livello di API e di librerie. Abbiamo rivisto il paragrafo sui thread Java in modo che ora includa i Future e abbiamo aggiornato la copertura di Grand Central Dispatch di Apple in modo che ora includa Swift. Nuovi paragrafi discutono il parallelismo fork-join usando il framework fork-join di Java e i Thread Building Blocks di Intel.
- **Capitolo 5: Scheduling della CPU** (ex Capitolo 6) rivede la copertura delle code multilivello e la schedulazione dei sistemi multicore. Abbiamo integrato la copertura delle problematiche di scheduling numa-aware in tutto il testo, includendo il modo in cui questo scheduling influenza sul bilanciamento del carico. Discutiamo anche le relative modifiche allo scheduler Linux CFS. La nuova trattazione comprende discussioni su round robin e scheduling con priorità, multiprocessing eterogeneo e scheduling di Windows 10.
- **Capitolo 6: Strumenti di sincronizzazione** (ex Capitolo 5) si concentra su vari strumenti per la sincronizzazione dei processi. Una parte nuova e rilevante discute questioni di architettura, come il riordino delle istruzioni e le scritture ritardate nei buffer. Il capitolo introduce anche algoritmi lock-free che utilizzano le istruzioni di compare-and-swap (CAS). Non sono presentate API specifiche; piuttosto, il capitolo fornisce un'introduzione alle corse critiche e agli strumenti generali che possono essere utilizzati per prevenire le corse critiche sui dati. Fra gli ulteriori dettagli vi sono una nuova copertura di modelli di memoria, barriere di memoria e problemi di liveness.
- **Capitolo 7: Esempi di sincronizzazione** (ex Capitolo 5) introduce i classici problemi di sincronizzazione e discute il supporto specifico di API per la progettazione di soluzioni che risolvono questi problemi. Il capitolo include una nuova trattazione dei semafori POSIX con nome e anonimi, oltre alle variabili condizionali. È compresa anche un nuovo paragrafo sulla sincronizzazione Java.
- **Capitolo 8: Stallo dei processi** (ex Capitolo 7) fornisce aggiornamenti minori, tra cui un nuovo paragrafo sui livelock e una discussione del deadlock come esempio di rischio di liveness. Il capitolo include una nuova copertura del `lockdep` Linux e degli strumenti `deadlock_detector` di BCC, oltre alla copertura del rilevamento dei deadlock Java mediante il dump dei thread.
- **Capitolo 9: Memoria centrale** (ex Capitolo 8) include diverse revisioni che aggiornano il capitolo per quanto riguarda la gestione della memoria sui computer moderni. Abbiamo aggiunto una nuova descrizione dell'architettura ARMv8 a 64 bit, aggiornato la trattazione delle librerie collegate dinamicamente e cambiato la descrizione dello swapping in modo che ora si concentrano sullo swapping delle pagine piuttosto che dei processi. Abbiamo anche eliminato la trattazione della segmentazione.
- **Capitolo 10: Memoria virtuale** (ex Capitolo 9) contiene diverse revisioni, inclusa la trattazione aggiornata dell'allocazione di memoria sui sistemi NUMA e l'allocazione globale utilizzando l'elenco dei frame liberi. Il nuovo materiale comprende la memoria compressa, gli errori di pagina principali/secondari e la gestione della memoria in Linux e Windows 10.
- **Capitolo 11: Memoria di massa** (ex Capitolo 10) aggiunge la trattazione dei dispositivi di memoria non volatile, come i dischi flash e a stato solido. Lo scheduling del disco rigido è semplificato per mostrare solo gli algoritmi attualmente utilizzati. Sono inclusi anche un nuovo paragrafo su cloud storage, l'aggiornamento della trattazione dei RAID e una nuova discussione sull'archiviazione di oggetti.
- **Capitolo 12: Sistemi di I/O** (ex Capitolo 13) aggiorna la trattazione delle tecnologie e degli indici di prestazioni, approfondisce la descrizione di I/O sincroni/asincroni, bloccanti/non bloccanti e aggiunge un paragrafo sull'I/O vettORIZZATO. Amplia inoltre la trattazione del power management per i sistemi mobili.
- **Capitolo 13: Interfaccia del file system** (ex Capitolo 11) è stato aggiornato con informazioni sulle tecnologie attuali. In particolare, la trattazione delle strutture di directory è stata migliorata e la descrizione della protezione è stata aggiornata. Il paragrafo riguardante i file mappati in memoria è stato ampliato ed è stato aggiunto alla discussione sulla memoria condivisa un esempio basato su API Windows. Nei Capitoli 13 e 14 l'ordine degli argomenti è stato riorganizzato.
- **Capitolo 14: Realizzazione del file system** (ex Capitolo 12) è stato aggiornato con dettagli sulle tecnologie attuali. Il capitolo include ora discussioni su TRIM e Apple File System. Inoltre, la discussione sulle prestazioni è stata aggiornata e la copertura del journaling è stata ampliata.
- **Capitolo 15: Dettagli interni del file system** è nuovo e contiene informazioni aggiornate dai precedenti Capitoli 11 e 12.
- **Capitolo 16: Sicurezza** (ex Capitolo 15) precede ora il capitolo sulla protezione. Include termini rivisti e aggiornati per le attuali minacce e soluzioni per la sicurezza, inclusi ransomware e strumenti di accesso remoto. Il principio del minimo privilegio è enfatizzato. La trattazione delle vulnerabilità e degli attacchi di code-injection è stata rivista e ora include esempi di codice. La discussione sulle tecnologie di crittografia è stata aggiornata per concentrarsi sulle tecnologie attualmente utilizzate. La trattazione dell'autenticazione (mediante password e altri metodi) è stata aggiornata e ampliata con indicazioni utili nella pratica. Le aggiunte includono una discussione sulla randomizzazione del layout dello spazio degli indirizzi e un nuovo riepilogo delle difese di sicurezza. L'esempio di Windows 7 è stato aggiornato a Windows 10.
- **Capitolo 17: Protezione** (ex Capitolo 14) contiene modifiche rilevanti. La discussione sugli anelli e livelli di protezione è stata aggiornata e ora fa riferimento al modello Bell-LaPadula e approfondisce il modello arm di Trust Zones e Secure Monitor Calls. La copertura del principio della necessità di sapere è stata ampliata, così come la trattazione del controllo obbligatorio degli accessi. Sono stati aggiunti sottoparagrafi sulle capability di Linux, autorizzazioni di Darwin, protezione dell'integrità, filtraggio delle chiamate di sistema, sandboxing e firma del codice. È stata aggiunta anche la copertura dell'applicazione delle regole basata sul tempo di esecuzione in Java, inclusa la tecnica di ispezione dello stack.
- **Capitolo 18: Macchine virtuali** (ex Capitolo 16) include ulteriori dettagli sulle tecnologie hardware di supporto. Inoltre è stato ampliato l'argomento dei contenitori di applicazioni, che ora include container, zone, Docker e Kubernetes. Un nuovo

paragrafo descrive la ricerca in corso sulla virtualizzazione, compresi gli unikernel, i sistemi operativi library, gli hypervisor di partizionamento e gli hypervisor di separazione.

- **Capitolo 19: Reti e sistemi distribuiti** (ex Capitolo 17) è stato sostanzialmente aggiornato e ora combina la trattazione delle reti di computer e dei sistemi distribuiti. Il materiale è stato rivisto per aggiornarlo rispetto alle reti contemporanee e ai sistemi distribuiti. Il modello tcp/ip viene trattato con ulteriore enfasi ed è stata aggiunta una discussione sul cloud storage. Il paragrafo sulle topologie di rete è stato eliminato. La copertura della risoluzione dei nomi di dominio è stata ampliata ed è stato aggiunto un esempio Java. Il capitolo include anche una nuova copertura dei file system distribuiti, tra cui MapReduce sul file system di Google, Hadoop, gpfs e Lustre.
- **Capitolo 20: Linux** (reperibile sulla piattaforma MyLab) (in precedenza Capitolo 18 on-line) è stato aggiornato per coprire il kernel 4.i di Linux.
- **Capitolo 21: Windows 10** (reperibile sulla piattaforma MyLab) è un nuovo capitolo che copre i componenti interni di Windows 10.
- **Appendice A: Prospettiva storica** (reperibile sulla piattaforma MyLab) (in precedenza Capitolo 20 on-line) è stata aggiornata per includere materiale proveniente da capitoli che non sono più trattati nel testo.

Contatti

Ci siamo sforzati di eliminare errori di battitura, bug e simili problemi dal testo. Ma, come nelle nuove versioni del software, alcuni bug rimangono quasi sicuramente. Un errata/correge aggiornato è accessibile dal sito web del libro. Vi saremmo grati se voleste comunicarci eventuali errori o omissioni nel libro che non sono nella lista corrente di errata.

Sono benvenuti i suggerimenti su come migliorare il libro. Riceviamo inoltre volentieri ogni contributo al sito web che possa essere di aiuto agli altri lettori, come esercizi di programmazione, progetti, laboratori e tutorial on-line e consigli per l'insegnamento. Inviate le vostre e-mail all'indirizzo os-book-authors@cs.yale.edu.

Ringraziamenti

Molte persone ci hanno aiutato per questa decima edizione, così come per le precedenti nove edizioni da cui è derivata.

Decima edizione

- Rick Farrow ci ha fornito consulenza come redattore tecnico.
- Jonathan Levin ci ha aiutato con la trattazione dei sistemi mobili, della protezione e della sicurezza.
- Alex Ionescu ha aggiornato il capitolo precedente di Windows 7 per realizzare il Capitolo 21: Windows 10.
- Sarah Diesburg ha rivisto il Capitolo 19: Reti e sistemi distribuiti.
- Brendan Gregg ha fornito indicazioni sul set di strumenti bcc.
- Richard Stallman (rms) ha fornito feedback sulla descrizione del software free e open-source.
- Robert Love ha fornito aggiornamenti al Capitolo 20: Linux.
- Michael Shapiro ci ha aiutato con i dettagli delle tecnologie di storage e i/o.
- Richard West ha fornito informazioni sulle aree di ricerca sulla virtualizzazione.
- Clay Breshears ci ha aiutato con la trattazione di Intel Threading Building Blocks.
- Gerry Howser ci ha fornito un feedback sulla motivazione dello studio dei sistemi operativi e anche provato nuovi materiali nella sua classe.
- Judi Paige ha aiutato nella realizzazione di figure e slide.
- Jay Gagne e Audra Rissmeyer hanno preparato una nuova grafica per questa edizione.
- Owen Galvin ha fornito l'editing tecnico per il Capitolo 11 e il Capitolo 12.
- Mark Wogahn ha fatto in modo che il software per produrre questo libro (LaTex e font) funzionasse correttamente.
- Ranjan Kumar Meher ha riscritto parte del software LaTex utilizzato nella realizzazione di questo libro.

Edizioni precedenti

- **Prime tre edizioni.** Questo testo deriva dalle precedenti edizioni, le prime tre delle quali sono state scritte insieme con James Peterson.
- **Contributi generali.** Tra le altre persone che sono state d'aiuto per quanto riguarda le precedenti edizioni ci sono Hamid Arbabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Cambell, P.C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Bart Childs, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Rasit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailparin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Don Heller, Bruce Hillyer, Mark Holliday, Dean Hougen, Michael Huang, Ahmed Kamel, Richard Kieburz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Euripides Montagne, Yoichi Muraoka, Jim M. Ng, Banu Ozden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J.C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovich, Adam Stauffer, Steven Stepanek, John Sterling, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L.Wear, John Werth, James M. Westall, J.S. Weston e Yang Xiang.

Contributi specifici

- Robert Love ha aggiornato il Capitolo 20 e le parti riguardanti Linux presenti nel testo, oltre ad aver risposto a tutte le nostre domande relative ad Android.
- L'Appendice B è stata scritta da Dave Probert sulla base del Capitolo 22 dell'ottava edizione.
- Jonathan Katz ha contribuito al Capitolo 16. Richard West ci ha dato suggerimenti per il Capitolo 18. Slahuddin Khan ha aggiornato il Paragrafo 16.7 per introdurre la sicurezza in Windows 7.
- Parte del Capitolo 19 deriva da un articolo di Levy e Silberschatz del 1990.

- Il Capitolo 20 è basato su un lavoro non pubblicato di Stephen Tweedie.
- Cliff Martin ci ha aiutato nell'introdurre Freebsd nell'Appendice su unix.
- Alcuni esercizi con le relative soluzioni ci sono stati forniti da Arvind Krishnamurthy.
- Andrew DeNicola ha preparato la guida allo studio che è disponibile sul sito web. Alcune delle slide sono state realizzate da Marilyn Turnamian.
- Mike Shapiro, Bryan Cantrill e Jim Mauro hanno risposto a diverse domande riguardanti Solaris. Bryan Cantrill della Sun Microsystems ci ha aiutato per la parte su zfs. Josh Dees e Rob Reynolds hanno contribuito alla parte su Microsoft .net.
- Owen Gavin ha aiutato nell'editing del Capitolo 18.

Note personali

Avi desidera ringraziare Valerie per il suo amore, la sua pazienza e il suo supporto durante la revisione di questo libro.

Peter vuole ringraziare sua moglie Carla e i suoi figli, Gwen, Owen, e Maddie.

Greg vuole riconoscere il continuo sostegno della sua famiglia: sua moglie Pat e i figli Thomas e Jay.

Abraham Silberschatz, New Haven, CT

Peter Baer Galvin, Boston, MA

Greg Gagne, Salt Lake City, UT

CAPITOLO 1

Introduzione

Un sistema operativo è un insieme di programmi (*software*) che gestisce gli elementi fisici di un calcolatore (*hardware*); fornisce una piattaforma ai programmi applicativi e agisce da intermediario fra l'utente e la struttura fisica del calcolatore. Un aspetto sorprendente dei sistemi operativi è quanto siano diversi i modi in cui eseguono questi compiti in un'ampia varietà di ambienti di elaborazione. I sistemi operativi sono ovunque, dalle automobili, agli elettrodomestici che incorporano dispositivi "Internet of Things", agli smartphone, ai personal computer, ai server di un'azienda, agli ambienti di cloud computing.

Per esplorare il ruolo di un sistema operativo in un ambiente di elaborazione moderno è importante capire l'organizzazione e l'architettura dell'hardware del computer, che include la cpu, la memoria, i dispositivi di i/o e i dispositivi di memorizzazione. Una responsabilità fondamentale del sistema operativo è allocare queste risorse ai programmi.

In ragione della sua complessità e ampiezza, un sistema operativo deve essere costruito gradualmente, per parti. Ciascuna di loro dovrebbe rappresentare un'unità ben riconoscibile del sistema, dotata di funzioni, dati in entrata e in uscita accuratamente definiti. Questo capitolo presenta una panoramica generale dei principali componenti di un moderno elaboratore e delle funzionalità fornite dal sistema operativo. Vengono inoltre coperti diversi argomenti aggiuntivi, in modo da preparare il terreno per il resto del testo: strutture dati utilizzate nei sistemi operativi, ambienti elaborativi, sistemi operativi free e open-source.

1.1 Che cosa fa un sistema operativo

La nostra analisi parte dalla considerazione del ruolo del sistema operativo nell'insieme del sistema di elaborazione. Un sistema di elaborazione si può suddividere in quattro componenti: *hardware*, *sistema operativo*, *programmi applicativi* e un *utente* (Figura 1.1).

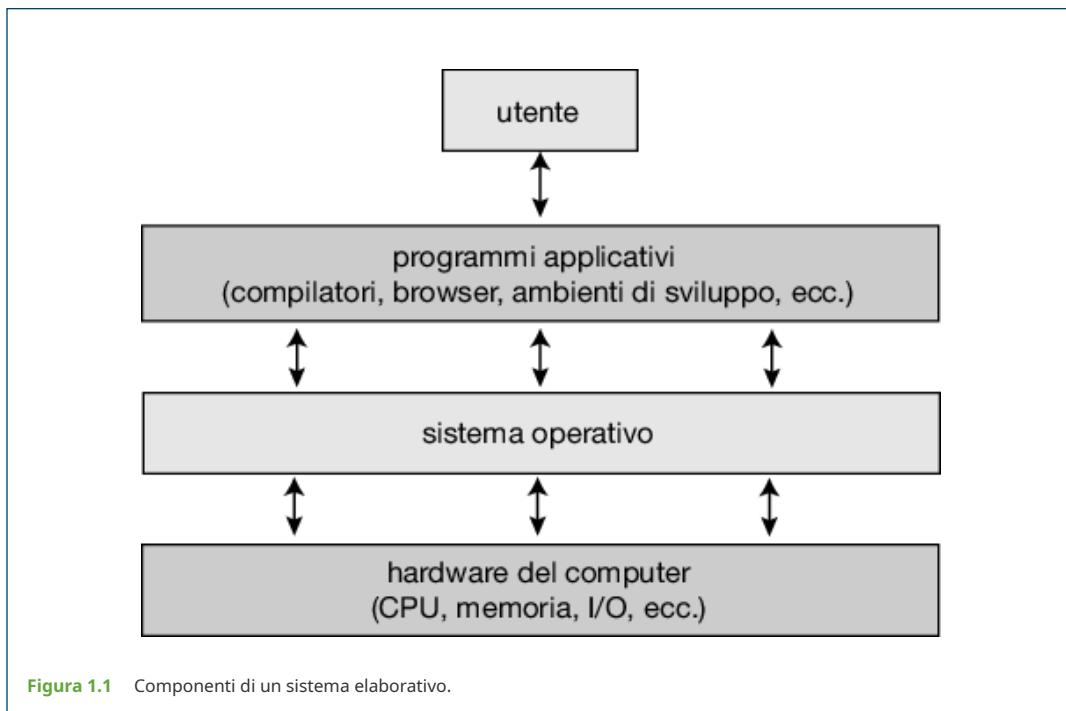


Figura 1.1 Componenti di un sistema elaborativo.

L'hardware, composto dall'unità centrale d'elaborazione o cpu (*central processing unit*), dalla memoria e dai dispositivi d'ingresso e uscita dei dati, cioè l'i/o (*input/output*), fornisce al sistema le risorse elaborate fondamentali. I programmi applicativi (editor di testo, fogli di calcolo, compilatori e browser web) definiscono il modo in cui si usano queste risorse per la risoluzione dei problemi computazionali degli utenti. Il sistema operativo controlla l'hardware e ne coordina l'utilizzo da parte dei programmi applicativi per gli utenti.

Un sistema elaborativo si può anche considerare come l'insieme di hardware, software e dati. Il sistema operativo offre gli strumenti per impiegare in modo corretto queste risorse. Il sistema operativo è simile a un governo: non compie operazioni di per sé utili, ma fornisce semplicemente un *ambiente* nel quale altri programmi possono lavorare in modo utile.

Per meglio approfondire il ruolo del sistema operativo, lo esploriamo da due punti di vista: quello degli utenti e quello del sistema.

1.1.1 Punto di vista dell'utente

La percezione di un calcolatore da parte di un utente dipende principalmente dall'interfaccia impiegata. La maggior parte degli utenti usa un computer portatile o un pc, composti da schermo, tastiera e mouse. Questi dispositivi sono progettati per un singolo utente, che impiega le risorse in modo esclusivo, con lo scopo di massimizzare la quantità di lavoro che l'utente può svolgere. In questo caso il sistema operativo si progetta considerando principalmente la facilità d'uso, con qualche attenzione alle prestazioni, ma nessuna all'utilizzo delle risorse, a come cioè sono condivise le risorse hardware e software.

Sempre più utenti utilizzano dispositivi mobili come smartphone e tablet e, per alcuni utenti, tali dispositivi sostituiscono sistemi desktop e laptop. Questi dispositivi sono in genere connessi alla rete tramite tecnologie cellulari o altre tecnologie wireless. L'interfaccia utente dei dispositivi mobili dispone generalmente di un touch-screen che permette all'utente di interagire con il sistema premendo e facendo scorrere le dita sullo schermo, anziché con l'utilizzo di una tastiera e di un mouse fisici. Molti dispositivi mobili consentono inoltre agli utenti di interagire per mezzo di un'interfaccia di riconoscimento vocale, come Siri di Apple.

Alcuni calcolatori hanno poca o nessuna visibilità per gli utenti: i calcolatori integrati (*embedded*) presenti in elettrodomestici e automobili, per esempio, possono avere una tastiera numerica e accendere o spegnere alcuni indicatori luminosi per segnalare il proprio stato; questi apparati e i relativi sistemi operativi, nella maggior parte dei casi, sono tuttavia progettati per funzionare senza l'intervento degli utenti.

1.1.2 Punto di vista del sistema

Dal punto di vista del calcolatore, il sistema operativo è il programma più strettamente correlato al suo hardware. In tale contesto è possibile considerare un sistema operativo come un assegnatore di risorse. Un sistema elaborativo dispone di risorse utili per la risoluzione di un problema: tempo di cpu, spazio di memoria, spazio per la memorizzazione di file, dispositivi di i/o e così via. Il sistema operativo agisce come gestore di tali risorse. Di fronte a richieste di risorse numerose ed eventualmente conflittuali, il sistema operativo deve decidere come assegnarle agli specifici programmi e utenti affinché il sistema elaborativo operi in modo equo ed efficiente.

Una visione leggermente diversa di un sistema operativo enfatizza la necessità di controllare i dispositivi di i/o e i programmi utenti; un sistema operativo è in effetti un programma di controllo. Un programma di controllo gestisce l'esecuzione dei programmi utenti in modo da impedire che si verifichino errori o che il calcolatore sia usato in modo scorretto. Si occupa soprattutto del funzionamento e del controllo dei dispositivi di i/o.

1.1.3 Definizione di sistema operativo

A questo punto avrete già capito che il termine *sistema operativo* definisce diversi ruoli e funzionalità. Ciò è dovuto, almeno in parte, ai differenti modelli di computer e ai loro utilizzi: i computer sono infatti presenti ovunque, dai tostapane alle automobili, dalle navi ai veicoli spaziali, dalle abitazioni alle aziende e sono la base di piattaforme per videogiochi, lettori multimediali, decoder per la televisione e sistemi di controllo industriale.

Per capire le differenze dobbiamo analizzare la storia dei calcolatori. Anche se i computer hanno una storia piuttosto breve, la loro evoluzione è stata rapida. La storia del calcolo automatico iniziò come esperimento per capire quello che era possibile fare ed evolvette velocemente in sistemi dedicati per scopi militari, come la decifrazione di codici o il disegno di traiettorie, e per usi governativi, come i censimenti. Da questi primi computer si passò a mainframe multifunzione per uso generale: fu in questa fase che nacquero i sistemi operativi. Negli anni '60 la legge di Moore predisse che il numero dei transistor nei circuiti integrati sarebbe raddoppiato ogni 18 mesi e la predizione si dimostrò vera. I computer acquisirono nuove funzionalità e si ridussero di dimensioni, dando avvio a un gran numero di utilizzi e alla creazione di una vasta gamma di sistemi operativi. Si veda l'Appendice A sulla piattaforma MyLab.

Come si può dunque definire che cosa sia un sistema operativo? In generale, non esiste una definizione completa ed esauriente. I sistemi operativi esistono poiché rappresentano una soluzione ragionevole al problema di realizzare un sistema elaborativo che si possa impiegare facilmente, per eseguire i programmi e agevolare la soluzione dei problemi degli utenti. È proprio a questo scopo che viene realizzato l'hardware dei calcolatori; ma, poiché il solo hardware non è molto facile da utilizzare, sono stati sviluppati i programmi applicativi. Questi programmi sono diversi tra loro, ma richiedono alcune funzioni comuni, per esempio il controllo dei dispositivi di i/o. Tali funzioni comuni, di controllo e assegnazione delle risorse, sono state racchiuse in un unico insieme di programmi: il sistema operativo.

Non esiste neppure una definizione universalmente accettata di che cosa faccia parte di un sistema operativo. Un punto di vista semplice è che esso comprenda tutto quello che il rivenditore fornisce quando gli si ordina "il sistema operativo". Tuttavia queste funzioni variano molto da sistema a sistema. Alcuni sistemi usano meno di un megabyte di memoria e non possiedono neppure un *editor* a pieno schermo, mentre altri richiedono gigabyte di memoria e sono interamente basati su interfacce grafiche a finestre. Una definizione più comune è quella secondo cui il sistema operativo è il solo programma che funziona sempre nel calcolatore, generalmente chiamato kernel (*nucleo*). (Oltre al kernel vi sono due tipi di programmi: i programmi di sistema, associati al sistema operativo, ma che non fanno necessariamente parte del kernel, e i programmi applicativi, che includono tutti i programmi non correlati al funzionamento del sistema).

La questione riguardante i componenti di un sistema operativo assunse un'importanza via via maggiore con la vasta diffusione dei personal computer e con la crescente complessità dei sistemi. Nel 1998 in Dipartimento della Giustizia degli Stati Uniti promosse un'azione legale contro la Microsoft, accusata di includere troppe funzioni nel sistema operativo (per esempio il browser era parte integrante del sistema operativo) e quindi di concorrenza sleale nei confronti dei produttori e rivenditori di applicazioni. Microsoft fu dichiarata colpevole di sfruttamento del proprio monopolio sul sistema operativo a danno della concorrenza.

Osservando oggi i sistemi operativi per dispositivi mobili notiamo che, ancora una volta, è aumentato il numero di funzionalità che ne fanno parte. I sistemi operativi mobili non sono costituiti esclusivamente da un kernel, ma anche da un middleware, ovvero da una collezione di ambienti software che fornisce servizi aggiuntivi per chi sviluppa applicazioni. Per esempio, entrambi i principali sistemi operativi per dispositivi mobili (ios di Apple e Android di Google) oltre a un kernel di base dispongono di un middleware che supporta (tra l'altro) database, multimedialità e grafica.

In breve, per i nostri scopi, il sistema operativo include il kernel sempre in esecuzione, i framework middleware che facilitano lo sviluppo di applicazioni e forniscono altre funzionalità e i programmi di sistema che contribuiscono alla gestione del sistema mentre è in esecuzione. La maggior parte di questo testo riguarda il kernel di sistemi operativi general-purpose ma, quando ciò risulta utile, vengono presentati anche altri componenti per capire a fondo il progetto e il funzionamento del sistema operativo.

1.2 Organizzazione di un sistema elaborativo

Un moderno calcolatore general-purpose è composto da una o più cpu e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (*bus*) che permette l'accesso alla memoria condivisa dal sistema (Figura 1.2). Ciascuno di questi controllori si occupa di un particolare tipo di dispositivo fisico (per esempio, unità disco, dispositivi audio e unità video) e può gestire una o più unità a esso connesse. Per esempio, una porta usb può connettersi a un hub usb, a cui possono connettersi diversi dispositivi. Un controllore di dispositivo dispone di una propria memoria interna (*buffer*) e di un insieme di registri specializzati. Il controllore è responsabile del trasferimento dei dati tra i dispositivi periferici a esso connessi e la propria memoria interna.

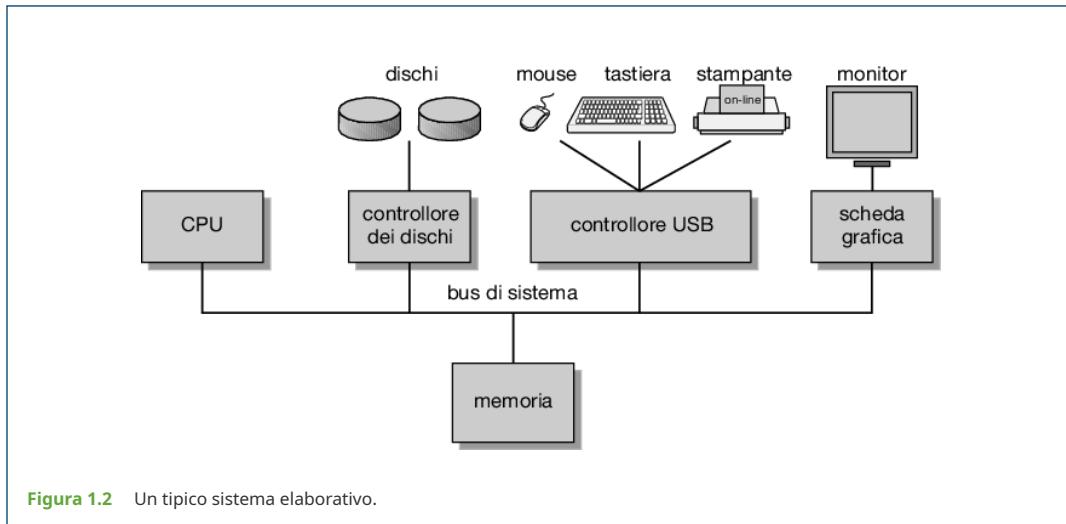


Figura 1.2 Un tipico sistema elaborativo.

I sistemi operativi possiedono in genere per ogni controllore di dispositivo un driver del dispositivo che gestisce le specificità del controllore e funge da interfaccia uniforme con il resto del sistema. La cpu e i controllori possono eseguire operazioni in parallelo, competendo per i cicli di memoria. Per garantire un accesso ordinato alla memoria condivisa un controllore della memoria sincronizza gli accessi.

Nei seguenti paragrafi descriveremo alcune nozioni di base sul funzionamento di un tale sistema, concentrando su tre aspetti chiave: inizieremo dalle interruzioni, che avvisano la cpu di eventi che richiedono attenzione e discuteremo quindi della struttura della memoria e della struttura dell'i/o.

1.2.1 Interruzioni

Si consideri un'operazione tipica del computer: un programma che esegue i/o. Per avviare un'operazione di i/o, il driver del dispositivo scrive negli appropriati registri all'interno del controllore, il quale esamina i contenuti di questi registri per determinare l'azione da intraprendere (per esempio "leggi un carattere dalla tastiera"). Il controllore comincia a trasferire i dati dal dispositivo al proprio buffer e a trasferimento completato informa il driver, tramite un'interruzione, di avere terminato l'operazione. Il driver passa quindi il controllo ad altre parti del sistema operativo, restituendo i dati (o un puntatore a essi) se l'operazione è di lettura; per altre operazioni, il driver restituisce delle informazioni di stato come "scrittura completata correttamente" o "periferica occupata". Come fa il controllore a comunicare al driver del dispositivo che ha terminato la sua operazione? Lo fa per mezzo di un'interruzione (*interrupt*).

1.2.1.1 Panoramica

L'hardware può generare un'interruzione in qualsiasi momento inviando un segnale alla cpu, solitamente tramite il bus di sistema (ci possono essere molti bus all'interno di un sistema di elaborazione, ma il bus di sistema è il principale percorso di comunicazione tra i componenti fondamentali). Le interruzioni sono utilizzate anche per molti altri scopi e costituiscono una componente chiave nell'interazione tra i sistemi operativi e l'hardware.

Quando riceve un segnale d'interruzione, la cpu interrompe l'elaborazione corrente e trasferisce immediatamente l'esecuzione a una locazione fissa della memoria. Di solito, questa locazione contiene l'indirizzo iniziale della procedura di servizio dell'interruzione. Una volta completata l'esecuzione della procedura richiesta, la cpu riprende l'elaborazione precedentemente interrotta. La Figura 1.3 mostra il diagramma temporale della gestione di un simile evento.

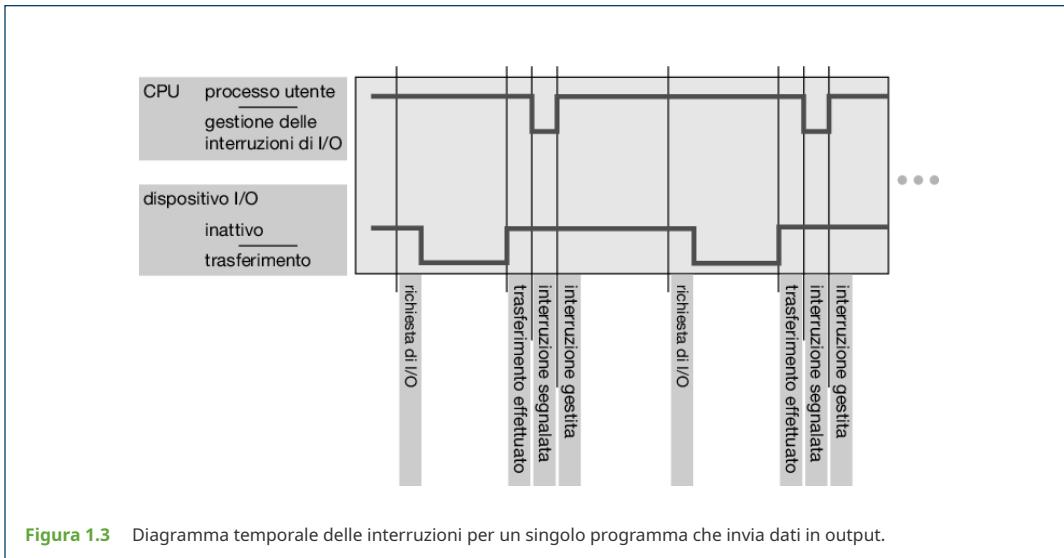


Figura 1.3 Diagramma temporale delle interruzioni per un singolo programma che invia dati in output.

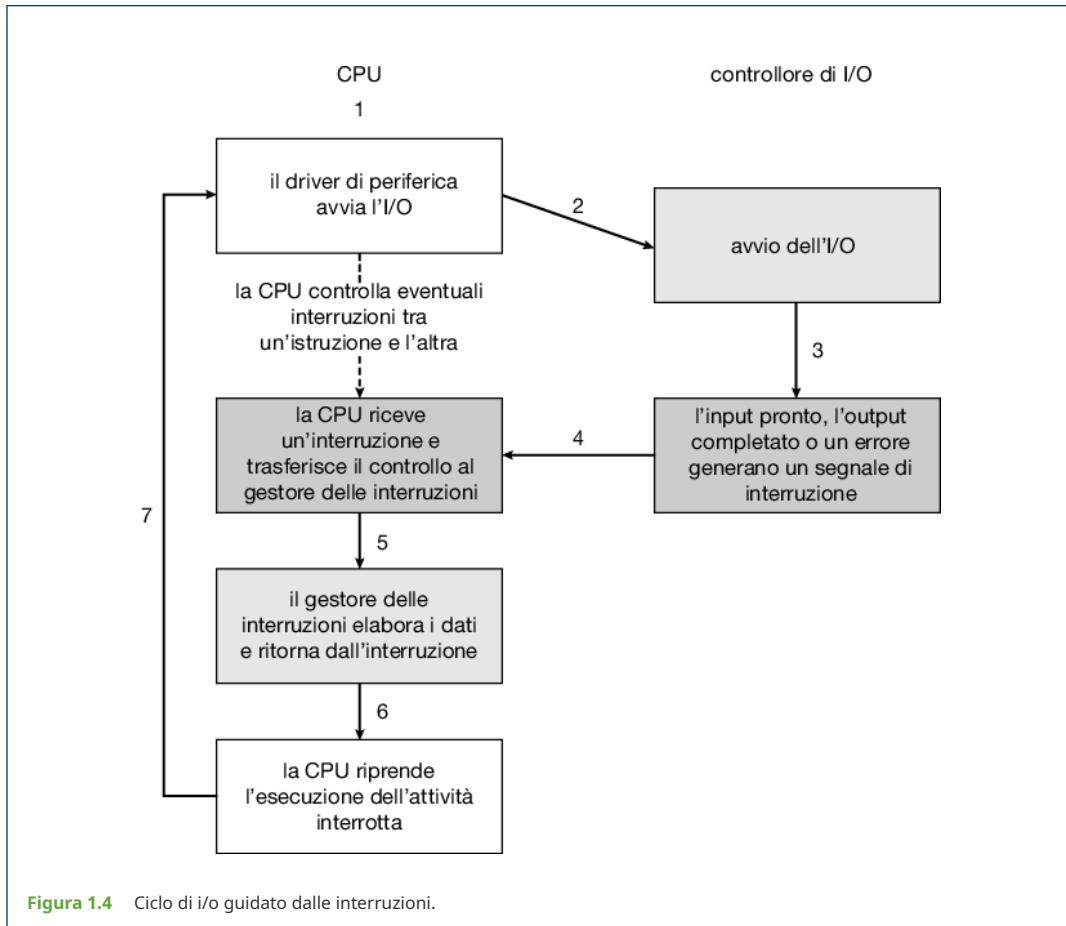
Le interruzioni sono un elemento importante nell'architettura di un calcolatore. Ciascun tipo di calcolatore ha il proprio meccanismo di gestione delle interruzioni; ciò nonostante molte funzioni sono comuni. Ogni interruzione deve causare il trasferimento del controllo all'appropriata procedura di servizio. Il modo più semplice per gestire quest'operazione è quello di impiegare una procedura generale che esamini le informazioni associate all'interruzione, e a sua volta invochi la procedura di gestione dello specifico evento. D'altra parte, la gestione di un'interruzione deve essere molto rapida, perché le interruzioni si verificano frequentemente. È possibile usare una tabella di puntatori alle specifiche procedure, in modo che l'attivazione delle procedure di servizio delle interruzioni avvenga in modo indiretto attraverso questa tabella, senza procedure intermedie. In genere questa tabella di puntatori è memorizzata nella memoria agli indirizzi più bassi (per esempio, le prime 100 locazioni). L'accesso a questa sequenza d'indirizzi, detta vettore delle interruzioni, avviene per mezzo di un indice, codificato nello stesso segnale d'interruzione, allo scopo di fornire l'indirizzo della procedura di servizio relativa all'evento segnalato dall'interruzione. Sistemi operativi molto differenti, come Windows e unix, usano questo stesso meccanismo di gestione delle interruzioni.

L'architettura di gestione delle interruzioni deve anche salvare le informazioni di stato dell'operazione interrotta, in modo da poterle ripristinare una volta che l'interruzione è stata servita. Se la procedura di gestione dell'interruzione richiede la modifica dello stato del processore, per esempio modificando il contenuto di qualche registro, deve salvare esplicitamente lo stato corrente per poterlo ripristinare prima di restituire il controllo. Terminato il servizio dell'interruzione, l'indirizzo di ritorno precedentemente salvato viene caricato nel contatore di programma (*program counter*), consentendo la ripresa della computazione interrotta come se nulla fosse accaduto.

1.2.1.2 Implementazione

Il meccanismo di base delle interruzioni funziona come segue. L'hardware della cpu dispone di un filo chiamato linea di richiesta di interruzione (*interrupt-request line*) che la cpu controlla dopo l'esecuzione di ogni istruzione. Quando la cpu rileva che un controllore ha asserito un segnale sulla linea di richiesta di interruzione, legge il numero di interruzione e salta alla routine di gestione dell'interruzione utilizzando il numero letto come indice nel vettore delle interruzioni, per poi avviare l'esecuzione all'indirizzo associato a tale indice. Il gestore delle interruzioni salva le informazioni di stato che verranno modificate durante le operazioni, determina la causa dell'interruzione, esegue l'elaborazione necessaria, ripristina lo stato ed esegue un'istruzione di ritorno dall'interruzione (*return_from_interrupt*) per riportare la cpu allo stato di esecuzione precedente all'interrupt. In altre parole, il controllore del dispositivo *solleva* un'interruzione asserendo un segnale sulla linea di richiesta di interruzione, la cpu *intercetta* l'interruzione e la *invia* al gestore delle interruzioni che, a sua volta, soddisfa l'interruzione offrendo al dispositivo il servizio richiesto.

La Figura 1.4 riepiloga il ciclo di i/o guidato dalle interruzioni (*interrupt-driven i/o*).



Il meccanismo di interruzione appena descritto consente alla cpu di rispondere a un evento asincrono, come nel caso in cui un controllore di periferica diventi pronto per offrire un servizio. In un moderno sistema operativo, tuttavia, sono necessarie funzionalità di gestione delle interruzioni più sofisticate. In particolare:

- abbiamo bisogno della possibilità di partecipare alla gestione degli interrupt durante un'elaborazione critica;
- abbiamo bisogno di un modo efficiente per inviare l'interruzione al gestore appropriato per un dispositivo;
- abbiamo bisogno di interruzioni multilivello, in modo che il sistema operativo possa distinguere tra interruzioni ad alta e bassa priorità e possa rispondere con il livello di urgenza appropriato.

In un'architettura moderna queste tre funzioni sono fornite dalla cpu e dall'hardware del controllore delle interruzioni.

La maggior parte delle cpu ha due linee di richiesta di interruzione: una è non mascherabile (*nonmaskable interrupt*), riservata a eventi come errori irreversibili di memoria, mentre la seconda è mascherabile (*maskable interrupt*), ovvero può essere disattivata dalla cpu prima dell'esecuzione di sequenze di istruzioni critiche che non devono essere interrotte. L'interruzione mascherabile è quella utilizzata dai controllori dei dispositivi per richiedere un servizio.

Ricordiamo che lo scopo di un meccanismo di interruzione basato sul vettore delle interruzioni è quello di evitare a un singolo gestore di interruzioni di dover cercare fra tutte le possibili fonti per determinare quale ha bisogno di assistenza. Nella pratica, tuttavia, i computer hanno più dispositivi (e, quindi, più gestori di interruzione) di quanti siano i campi indirizzo presenti nel vettore delle interruzioni. Un modo comune per risolvere questo problema è utilizzare il concatenamento delle interruzioni (*interrupt chaining*), in cui ogni elemento nel vettore delle interruzioni punta alla testa di un elenco di gestori. Quando viene sollevata un'interruzione, i gestori nell'elenco corrispondente vengono chiamati a uno a uno, finché non ne viene trovato uno in grado di servire la richiesta. Tale struttura è un compromesso tra lo svantaggio di avere una tabella delle interruzioni troppo grande e l'inefficienza dell'invio dell'interruzione verso un singolo gestore.

La Figura 1.5 mostra il vettore delle interruzioni dei processori Intel. Gli eventi da 0 a 31, che non sono mascherabili, vengono utilizzati per segnalare varie condizioni di errore. Gli eventi da 32 a 255, che sono mascherabili, vengono utilizzati, tra l'altro, per le interruzioni generate dai dispositivi.

numero di vettore	descrizione
0	errore di divisione
1	eccezione di debug
2	interruzione null
3	breakpoint
4	eccezione di overflow
5	eccezione di range exceeded
6	codice operativo non valido
7	dispositivo non disponibile
8	doppio errore
9	overrun del segmento coprocessore (riservato)
10	task state segment (tss) non valido
11	segmento non presente
12	errore di stack
13	protezione generale
14	errore di pagina
15	(riservato Intel, non utilizzare)
16	errore in virgola mobile
17	controllo dell'allineamento
18	controllo della macchina
19–31	(riservato Intel, non utilizzare)
32–255	interruzioni mascherabili

Figura 1.5 Tabella degli eventi di un processore Intel.

Il meccanismo di interrupt implementa anche un sistema di livelli di priorità delle interruzioni. Questi livelli consentono alla cpu di partecipare la gestione delle interruzioni a bassa priorità senza mascherare tutte le interruzioni e permettono a un'interruzione ad alta priorità di essere gestita prima di un'interruzione a bassa priorità.

In sintesi, le interruzioni vengono utilizzate nei moderni sistemi operativi per gestire eventi asincroni (e per altri scopi, che vedremo più avanti nel testo). Le interruzioni sono sollevate dai controllori dei dispositivi e dal verificarsi di errori. Per consentire al lavoro più urgente di essere completato per primo, i computer moderni utilizzano un sistema di priorità delle interruzioni. Poiché le interruzioni vengono pesantemente utilizzate nell'elaborazione time-sensitive, è necessaria una gestione efficiente delle interruzioni per garantire buone prestazioni del sistema.

MEMORIA: DEFINIZIONI E NOTAZIONE

L'unità di memorizzazione di base di un computer è il **bit**. Un bit può assumere uno tra i due valori 0 e 1. Tutto ciò che viene memorizzato su un computer è formato da una collezione di bit. È incredibile quante cose i computer siano in grado di rappresentare con un numero sufficiente di bit: numeri, lettere, immagini, filmati, suoni, documenti e programmi, solo per fare qualche esempio. Un **byte** è formato da 8 bit e nella maggior parte dei calcolatori è la più piccola unità di memorizzazione utile. Ad esempio, la maggior parte dei computer non dispone di un'istruzione per spostare un singolo bit, ma ne possiede una per spostare un byte. Una nozione meno comune è quella di **parola** (*word*), l'unità di memorizzazione nativa di una data architettura. Una parola è costituita da uno o più byte. Ad esempio, un computer con registri di 64 bit e indirizzamento della memoria a 64 bit ha di solito parole di 64 bit (8 byte). Un computer esegue diverse operazioni utilizzando la dimensione di parola nativa, piuttosto che un byte alla volta. La memoria di un computer, come molti altri suoi parametri, è generalmente misurata e manipolata in byte o gruppi di byte. Un **kilobyte**, o **kb**, corrisponde a 1.024 byte; un **megabyte**, o **mb**, è $1,024^2$ byte; un **gigabyte**, o **gb**, è $1,024^3$ byte; un **terabyte**, o **tb**, è $1,024^4$ byte; un **petabyte**, o **pb**, è $1,024^5$ byte. I produttori approssimano spesso questi numeri, parlando di megabyte come di 1 milione di byte e di gigabyte come di 1 miliardo di byte. Le misure relative alle reti

costituiscono un'eccezione a queste regole generali e vengono espresse in bit (perché le reti spostano i dati un bit alla volta).

1.2.2 Struttura della memoria

La cpu può caricare istruzioni esclusivamente dalla memoria, quindi tutti i programmi da eseguire devono esservi caricati. I computer general-purpose eseguono la maggior parte dei programmi da una memoria riscrivibile, la memoria principale, chiamata anche memoria ad accesso casuale (*random access memory*, ram). La memoria principale è realizzata solitamente con una tecnologia basata su semiconduttori chiamata memoria dinamica ad accesso casuale (*dynamic random access memory*, dram).

I computer utilizzano anche altri tipi di memoria. Per esempio, il primo programma che viene eseguito all'avvio di un computer è il programma di avviamento (*bootstrap*), che si occupa di caricare il sistema operativo. Visto che la ram è volatile, ovvero perde il suo contenuto quando la corrente viene a mancare, non può essere utilizzata per il programma di bootstrap. Al suo posto, per questo e per altri scopi, i computer utilizzano una memoria di sola lettura elettricamente cancellabile e programmabile (eprom) e altre forme di archiviazione del firmware che vengono riscritte raramente e che non sono volatili. Il contenuto delle eeprom, anche se modificabile, non può essere modificato di frequente. Inoltre le eeprom sono lente e per questa ragione contengono principalmente programmi statici e dati utilizzati con poca frequenza. L'iPhone, per esempio, utilizza una eeprom per memorizzare il numero di serie e le informazioni sull'hardware del dispositivo.

Tutte le tipologie di memoria forniscono un vettore di byte. Ciascun byte possiede un proprio indirizzo. L'interazione avviene per mezzo di una sequenza di istruzioni *load* e *store* opportunamente indirizzate. L'istruzione *load* trasferisce il contenuto di un byte o una parola della memoria centrale in uno dei registri interni della cpu, mentre la *store* copia il contenuto di uno di questi registri nella locazione di memoria specificata. Oltre agli accessi esplicativi, tramite *load* e *store*, la cpu preleva automaticamente dalla memoria centrale, alla locazione di memoria contenuta nel contatore di programma, le istruzioni da eseguire.

La tipica sequenza d'esecuzione di un'istruzione, in un sistema con architettura di von Neumann, comincia con il prelievo (*fetch*) di un'istruzione dalla memoria centrale e il suo trasferimento nel registro d'istruzione. Quindi si decodifica l'istruzione che eventualmente può richiedere il trasferimento di alcuni operandi dalla memoria in alcuni registri interni. Una volta terminata l'esecuzione dell'istruzione sugli operandi, il risultato si può scrivere nella memoria. Si noti che l'unità di memoria "vede" soltanto un flusso di indirizzi di memoria; non importa né il modo in cui questi sono stati generati (dal contatore di programma, per indicizzazione, riferimento indiretto, indirizzamento immediato, e così via) né a che cosa fanno riferimento (istruzioni o dati). Di conseguenza, anche la presente trattazione non si cura di come questi indirizzi siano generati all'interno dei programmi e si occupa semplicemente della sequenza d'indirizzi della memoria generata dal programma in esecuzione.

Idealmente si vorrebbe che sia i programmi sia i dati da essi trattati risiedessero in modo permanente nella memoria centrale. Questo non è possibile per i seguenti due motivi:

1. la capacità della memoria centrale non è di solito sufficiente a contenere in modo permanente tutti i programmi e i dati richiesti;
2. la memoria centrale è un dispositivo di memorizzazione *volatile*, che perde il proprio contenuto quando l'alimentazione elettrica viene spenta o si interrompe.

Per queste ragioni la maggior parte dei sistemi elaborativi comprende una memoria secondaria come estensione della memoria centrale. La caratteristica fondamentale di questi dispositivi è la capacità di conservare in modo permanente grandi quantità di informazioni.

I dispositivi più comunemente impiegati a questo scopo sono il disco magnetico (*hard-disk drive*, hdd) e i dispositivi di memoria non volatile (nvm), adoperati per la memorizzazione sia di programmi sia di dati. La maggior parte dei programmi (applicativi e di sistema) è mantenuta in un disco sino al momento del caricamento nella memoria. Molti programmi fanno uso del disco come sorgente e destinazione delle informazioni elaborate. Quindi, come si spiega nel Capitolo 11, una corretta gestione delle unità a disco è di fondamentale importanza per un sistema elaborativo.

Occorre tuttavia specificare che la struttura descritta (composta da registri, memoria centrale e unità a disco) rappresenta semplicemente una delle possibili configurazioni del sistema di memorizzazione di un calcolatore. Esistono altri tipi di memorie, per esempio le memorie cache, i cd-rom, i blu-ray e i nastri magnetici (memoria terziaria). Qualsiasi sistema di memorizzazione fornisce le funzioni fondamentali che consentono la memorizzazione di un dato e il suo mantenimento fino all'uso successivo. Le caratteristiche che differenziano i diversi sistemi sono velocità, costo, dimensioni e volatilità.

L'ampio ventaglio dei sistemi di memorizzazione può essere ordinato secondo una scala gerarchica (Figura 1.6), sulla base della capacità e del tempo d'accesso. Come regola generale, vi è un compromesso tra dimensione e velocità. Inoltre, più vicino alla cpu si utilizza una memoria più piccola e più veloce. Come mostrato nella figura, oltre a differenze di velocità e capacità, i vari sistemi di memorizzazione si dividono in volatili o non volatili. La memoria volatile, come accennato in precedenza, perde il suo contenuto quando viene tolta l'alimentazione; quindi i dati vengono scritti su una memoria non volatile se devono essere memorizzati in maniera sicura.

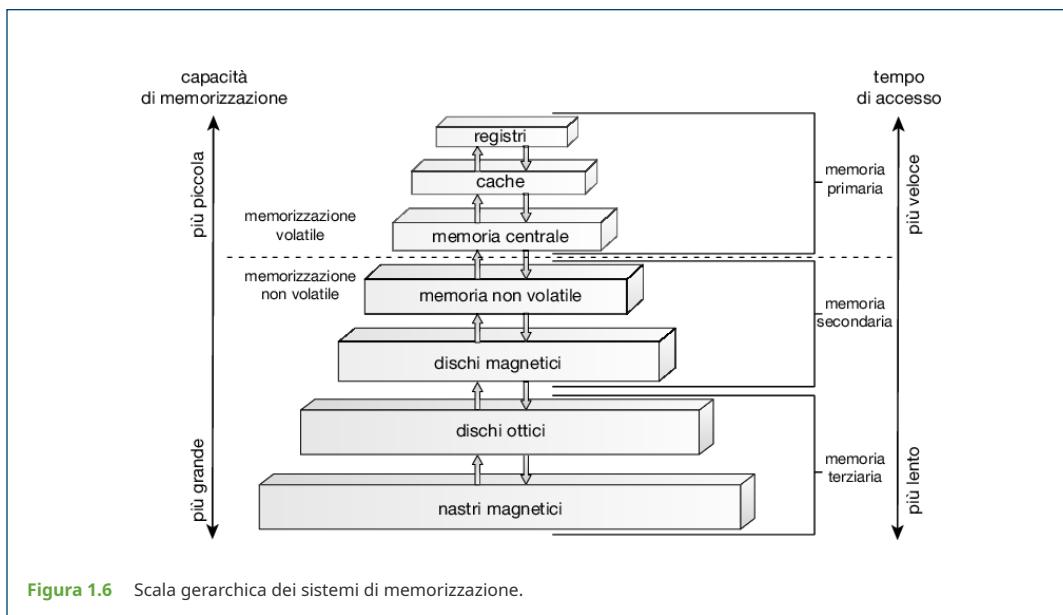


Figura 1.6 Scala gerarchica dei sistemi di memorizzazione.

I primi quattro livelli nella figura sono costituiti da memorie a semiconduttore, consistenti in circuiti elettronici basati su semiconduttori. I dispositivi nvm, al quarto livello, hanno diverse varianti, ma in generale sono più veloci dei dischi magnetici. La forma più comune di dispositivo nvm è la memoria flash, frequentemente usata in dispositivi mobili come smartphone e tablet. Sempre più spesso, la memoria flash viene anche utilizzata per l'archiviazione a lungo termine su laptop, desktop e server.

Dal momento che la memorizzazione gioca un ruolo importante nella struttura del sistema operativo vi faremo riferimento spesso nel testo. In generale, useremo la seguente terminologia.

- La memoria volatile verrà indicata semplicemente come memoria. Se sarà necessario enfatizzare un particolare tipo di dispositivo di memorizzazione (per esempio un registro), lo faremo esplicitamente.
- La memoria non volatile, che indicheremo con l'acronimo nvs (*nonvolatile storage*), conserva il suo contenuto quando viene persa l'alimentazione. La maggior parte del tempo che dedicheremo a nvs riguarderà la memoria secondaria, classificabile in due diverse tipologie:
- Meccanica. Alcuni esempi di tali sistemi di memorizzazione sono gli hdd, i dischi ottici, la memoria olografica e il nastro magnetico. Se avremo bisogno di enfatizzare un particolare tipo di dispositivo di memorizzazione meccanica (per esempio il nastro magnetico), lo faremo esplicitamente.
- Elettrica. Alcuni esempi di tali sistemi di archiviazione sono la memoria flash, la fram, la nram e gli ssd. La memoria elettrica verrà indicata come nvm. Se dovremo sottolineare un particolare tipo di memoria elettrica (per esempio, gli ssd), lo faremo esplicitamente.

La memoria meccanica è generalmente più grande e meno costosa (per byte) rispetto alla memoria elettrica. Viceversa, la memoria elettrica è in genere più cara, più piccola e più veloce della memoria meccanica.

Nel progettare un sistema di memorizzazione completo si deve attribuire la giusta rilevanza a ciascun fattore: l'uso di memoria costosa va limitato al necessario; in compenso, è bene prevedere la massima quantità possibile di memoria non volatile ed economica. L'installazione di memorie cache sopperisce a eventuali macroscopiche disparità nei tempi di accesso o nella velocità di trasferimento tra due componenti, consentendo miglioramenti nelle prestazioni.

1.2.3 Struttura di i/o

Una percentuale cospicua del codice di un sistema operativo è dedicata alla gestione dell'i/o; ciò è dovuto sia alla sua importanza per il progetto di un sistema affidabile ed efficiente, sia alla differente tipologia dei dispositivi.

Un calcolatore general-purpose è composto da un insieme di dispositivi connessi mediante un bus comune.

L'i/o guidato dalle interruzioni descritto nel Paragrafo 1.2.1 è adatto al trasferimento di piccole quantità di dati, ma in caso di trasferimenti massicci può generare un pesante sovraccarico (*overhead*); si pensi, per esempio, all'i/o da e verso nvs. Per risolvere questo problema si utilizza la tecnica dell'accesso diretto alla memoria (dma). Una volta impostati i buffer, i puntatori e i contatori necessari al dispositivo di i/o, il controllore trasferisce un intero blocco di dati dal proprio buffer direttamente nella memoria centrale, o viceversa, senza alcun intervento da parte della cpu. In questo modo l'operazione richiede una sola interruzione per ogni blocco di dati trasferito, piuttosto che per ogni byte, come avviene nel caso dei dispositivi più lenti. Così, mentre il controllore del dispositivo effettua le operazioni descritte, la cpu rimane libera di occuparsi di altri compiti.

Alcuni sistemi ad alte prestazioni hanno abbandonato la configurazione basata sul bus per adottare un'architettura incentrata sugli switch, in cui più dispositivi fisici possono interagire con varie parti del sistema concorrentemente, piuttosto che contendersi un unico

bus condiviso. In questo caso, il dma risulta ancora più efficace. La Figura 1.7 mostra l'interazione di tutti i componenti di un elaboratore.

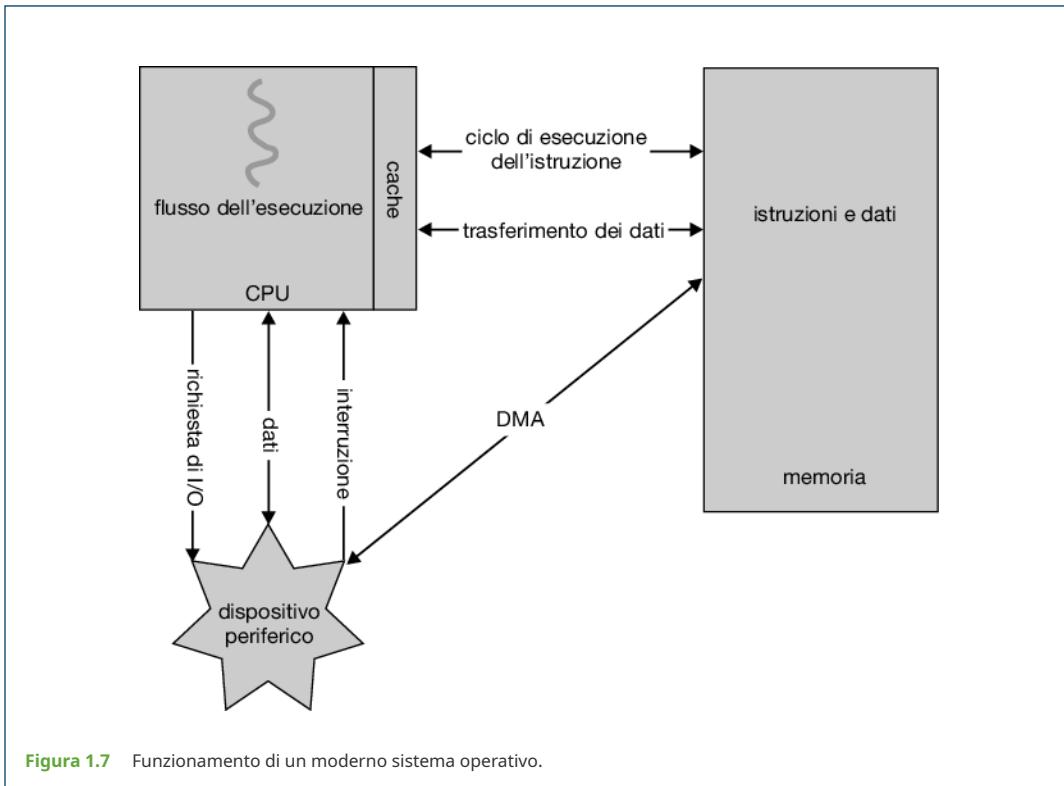


Figura 1.7 Funzionamento di un moderno sistema operativo.

1.3 Architettura degli elaboratori

Nel Paragrafo 1.2 è stata presentata la struttura generale di un calcolatore, che può essere organizzato secondo criteri molto diversi; in prima battuta faremo riferimento al numero di unità di elaborazione di uso generale presenti nell'elaboratore.

1.3.1 Sistemi monoprocessore

Diversi anni fa la maggior parte dei sistemi utilizzava un solo processore contenente un'unica cpu con un unico nucleo di elaborazione (o unità di calcolo, o *core*). Il nucleo di elaborazione è la componente che esegue le istruzioni e utilizza registri per memorizzare i dati localmente. La singola cpu con il suo nucleo di elaborazione è in grado di eseguire un insieme di istruzioni di natura generale, comprese quelle necessarie ai processi utenti. I sistemi monoprocessore, inoltre, possiedono altri processori specializzati, deputati a compiti particolari. Essi possono assumere la forma di processori specifici dedicati a un dispositivo, quali i controllori del disco, della tastiera o del video.

Tutti questi processori specializzati sono dotati di un insieme ristretto di istruzioni, e non eseguono processi utenti. Talvolta sono guidati dal sistema operativo, che può inviare loro informazioni sul compito da eseguire successivamente, e controllarne lo stato. Prendiamo l'esempio di un microprocessore che funge da controllore del disco e che riceve dalla cpu un elenco di richieste; spetta a esso implementare una coda e un algoritmo di scheduling per gestirle. Questa organizzazione alleggerisce la cpu dal sovraccarico di lavoro legato allo scheduling del disco. I pc ospitano un microprocessore all'interno della tastiera, per convertire la pressione di ciascun tasto nel codice appropriato da trasmettere alla cpu. In altre circostanze o sistemi, i processori specializzati sono dispositivi di basso livello integrati nell'hardware. Il sistema operativo non può comunicare con questi processori, che svolgono in autonomia il proprio lavoro. L'utilizzo di microprocessori specializzati è comune e non trasforma un sistema monoprocessore in un sistema multiprocessore. Se è presente una sola cpu, si tratta di un sistema monoprocessore. Secondo questa definizione, tuttavia, pochissimi computer moderni sono sistemi monoprocessore.

1.3.2 Sistemi multiprocessore

Nei moderni computer, dai dispositivi mobili ai server, i sistemi multiprocessore dominano oggi il panorama della computazione. Tradizionalmente, un sistema di questo tipo dispone di due o più processori, ciascuno con una cpu dotata di una singola unità di calcolo (*single-core*), che condividono il bus e talvolta il clock di sistema, la memoria e i dispositivi periferici. Il vantaggio principale dei sistemi multiprocessore è la maggiore capacità elaborativa (*throughput*). Aumentando il numero di unità di elaborazione è infatti possibile svolgere un lavoro maggiore in meno tempo. Con N unità di elaborazione la velocità non aumenta tuttavia di N volte, ma in misura minore. Infatti, se più unità di elaborazione collaborano nell'esecuzione di un compito, è necessario un certo lavoro supplementare (*overhead*) per garantire che tutti i componenti funzionino correttamente. Questo overhead, unito alla contesa per le risorse condivise, riduce il guadagno atteso dalla disponibilità di più unità di elaborazione.

I sistemi più comuni utilizzano la multielaborazione simmetrica (*symmetric multiprocessing*, smp), in cui ogni processore, alla pari degli altri, è abilitato all'esecuzione di tutte le operazioni del sistema, incluse le funzioni del sistema operativo e i processi utente. La Figura 1.8 illustra una tipica architettura smp con due processori, ciascuno con la propria cpu. Si noti che ogni cpu ha il proprio set di registri e una cache privata (locale). Tutti i processori condividono, tuttavia, la memoria fisica sul bus di sistema.

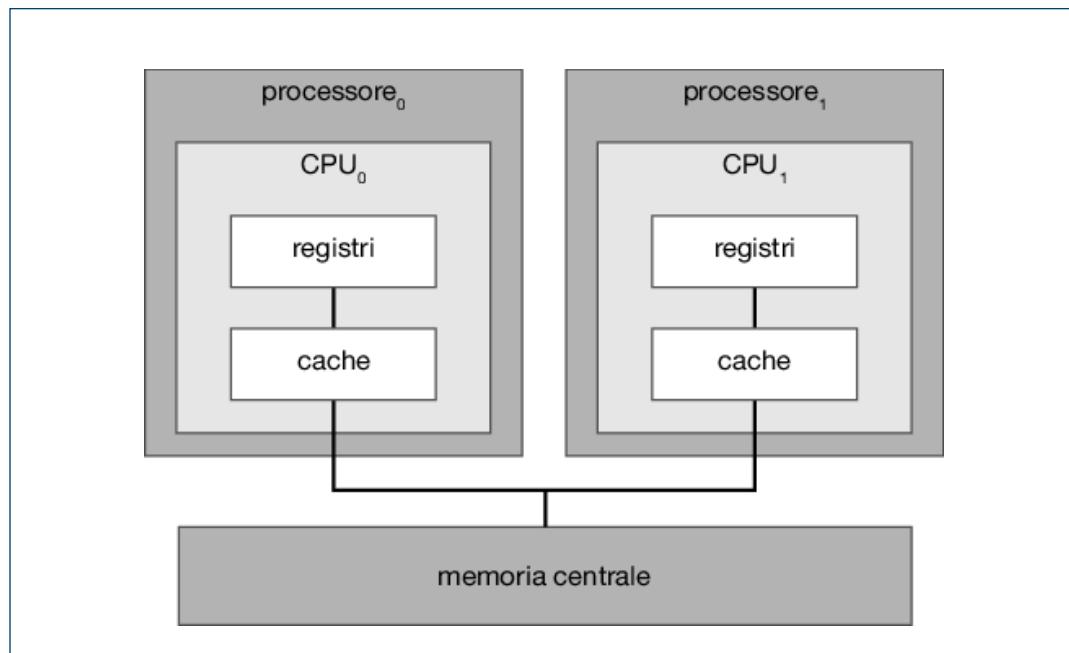


Figura 1.8 Architettura di multielaborazione simmetrica.

Il vantaggio offerto da questo modello è che molti processi sono eseguibili contemporaneamente (N processi se si hanno N cpu) senza causare un rilevante calo delle prestazioni. Tuttavia, poiché le unità di elaborazione sono separate, una potrebbe essere inattiva, mentre un'altra è sovraccarica e ciò determina un'inefficienza che sarebbe evitabile se le unità d'elaborazione condividessero alcune strutture dati. Un sistema multiprocessore di questo tipo permetterebbe infatti di condividere dinamicamente processi e risorse (per esempio la memoria) tra i vari processori, riducendo la varianza del carico di lavoro tra di essi. Come si vedrà nei Capitoli 5 e 6, un sistema di questo tipo deve essere implementato con molta cura.

La definizione di multiprocessore si è evoluta nel tempo e include ora i sistemi multicore, in cui più unità di calcolo (core) risiedono su un singolo chip. Questi sistemi possono essere più efficienti rispetto a più chip dotati di una singola unità di calcolo, perché la comunicazione all'interno di un singolo chip è più veloce rispetto a quella tra un chip e un altro. Inoltre, un chip dotato di diversi core usa molta meno potenza rispetto a diversi chip con un singolo core e ciò è particolarmente importante in dispositivi mobili e laptop.

COMPONENTI DI UN SISTEMA DI ELABORAZIONE

- **cpu**: componente hardware che esegue le istruzioni.
- **Processore**: chip che contiene una o più cpu.
- **Unità di calcolo (core)**: unità di elaborazione di base della cpu.
- **Multicore**: che include più unità di calcolo sulla stessa cpu.
- **Multiprocessore**: che include più processori.

Anche se praticamente tutti i sistemi sono ora multicore, usiamo il termine generico **cpu** quando ci riferiamo a una singola unità computazionale di un sistema elaborativo e unità di calcolo, o **core**, così come **multicore**, quando ci si riferisce espressamente a uno o più core sulla cpu.

Nella Figura 1.9 è illustrata un'architettura *dual core*, con due unità sullo stesso chip. In un'architettura di questo tipo ogni unità di calcolo ha il proprio insieme di registri e la propria cache, spesso nota come cache di livello 1 o L1. Si noti inoltre che una cache di livello 2 (L2) è locale al chip, ma è condivisa dai due core di elaborazione. La maggior parte delle architetture adotta questo approccio, utilizzando una combinazione di cache locali e condivise in cui le cache locali di livello inferiore sono generalmente più piccole e più veloci rispetto alle cache condivise di livello superiore. A prescindere da considerazioni architettoniche come la competizione per l'uso della cache, della memoria e del bus, queste cpu multicore appaiono al sistema operativo come N normali processori. Questa caratteristica mette sotto pressione i progettisti dei sistemi operativi e i programmati di applicazioni, alla ricerca di tecniche per sfruttare in maniera efficiente tutte le unità di elaborazione. Tratteremo la questione nel Capitolo 4. Praticamente tutti i sistemi operativi moderni, inclusi Windows, macos, Linux e i sistemi mobili Android e ios, supportano sistemi smp multicore.

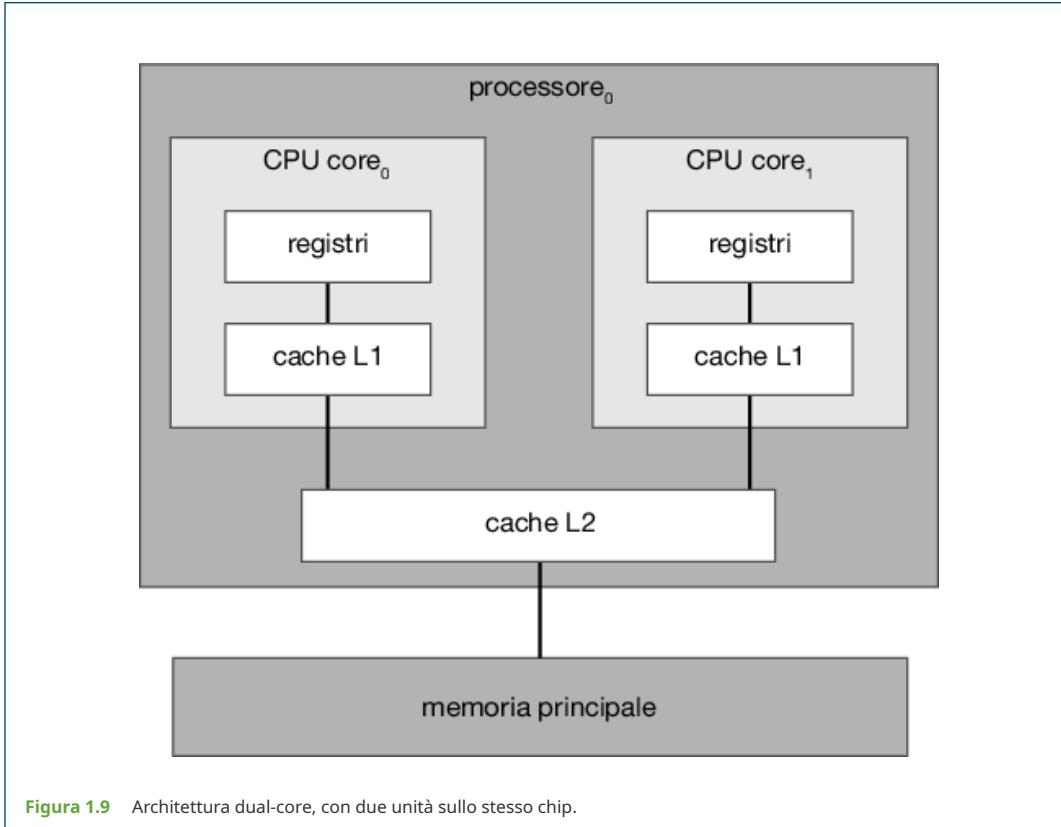
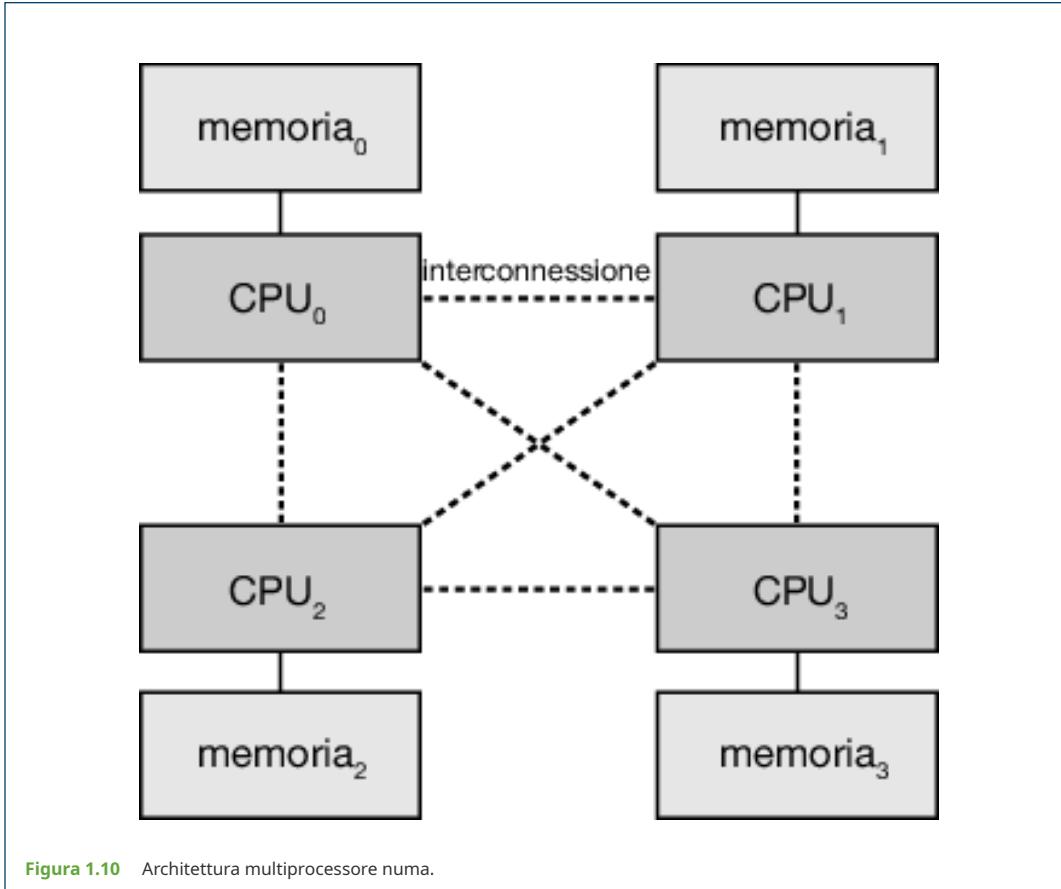


Figura 1.9 Architettura dual-core, con due unità sullo stesso chip.

Aggiungere nuove cpu a un multiprocessore ne aumenta la potenza di calcolo; tuttavia, come suggerito in precedenza, il concetto non scala molto bene e una volta aggiunte troppe cpu il conflitto per il bus di sistema diventa un collo di bottiglia e le prestazioni iniziano a peggiorare. Un approccio alternativo è quello di fornire a ciascuna cpu (o a ciascun gruppo di cpu) la propria memoria locale accessibile per mezzo di un bus locale piccolo e veloce. Le cpu sono collegate da un'interconnessione di sistema condivisa, in modo che tutte le cpu condividano uno spazio di indirizzamento fisico. Questo approccio, noto come accesso non uniforme alla memoria o numa, è illustrato nella Figura 1.10. Il vantaggio è che quando una cpu accede alla sua memoria locale, non solo è veloce, ma non vi è alcun conflitto sull'interconnessione di sistema. Pertanto, i sistemi numa possono scalare in modo più efficace con l'aggiunta di più processori.



Un potenziale svantaggio di un sistema numa è la maggior latenza quando una cpu deve accedere alla memoria remota attraverso l'interconnessione di sistema, con un conseguente possibile peggioramento delle prestazioni. In altre parole, per esempio, la cpu_0 non può accedere alla memoria locale della cpu_3 alla stessa velocità con cui può accedere alla propria memoria locale, rallentando così le prestazioni. I sistemi operativi possono minimizzare questo svantaggio del numa attraverso un attento scheduling della cpu e un'accurata gestione della memoria, come discusso nel Paragrafo 5.5.2 (Capitolo 5) e nel Paragrafo 10.5.4 (Capitolo 10). Poiché i sistemi numa possono scalare per ospitare un numero elevato di processori, stanno diventando sempre più diffusi nei server e nei sistemi di elaborazione ad alte prestazioni.

Infine, menzioniamo i cosiddetti server blade, che accolgono nello stesso contenitore fisico le schede del processore, dell'i/o e della rete. A differenza dei tradizionali sistemi multiprocessore, nei server blade ogni scheda dotata di processore avvia ed esegue in maniera indipendente il proprio sistema operativo. Alcune di queste schede, poi, possono essere a loro volta multiprocessore, il che rende più labile la distinzione tra questi diversi tipi di computer. Si può affermare, in sintesi, che tali server sono costituiti da svariati sistemi multiprocessore indipendenti.

1.3.3 Cluster di elaboratori

I cluster di elaboratori (*clustered systems*) o cluster sono un altro tipo di sistemi multiprocessore, basati sull'uso congiunto di più cpu, ma differiscono dai sistemi multiprocessore descritti nel paragrafo precedente per il fatto che sono composti di due o più calcolatori completi – detti nodi – collegati tra loro. Sistemi di questo tipo sono detti debolmente accoppiati (*loosely coupled*). Va osservato che per tali sistemi non esiste una definizione precisa; c'è un dibattito aperto tra i fornitori delle varie offerte commerciali su tale definizione e sul perché una soluzione sia migliore di un'altra. La definizione generalmente accettata è che si tratta di calcolatori che condividono la memoria di massa, connessi per mezzo di una rete locale (lan), come descritto nel Capitolo 19, o da connessioni più veloci come InfiniBand.

Di solito si adotta il clustering per offrire un'elevata disponibilità. Ciascun calcolatore esegue uno strato software di gestione del cluster; ogni nodo può tenere sotto controllo (attraverso la lan) uno o più degli altri nodi. Se il nodo controllato si guasta, il calcolatore che lo sta supervisionando può prendere il controllo della sua memoria e riavviare le applicazioni che erano in esecuzione. Gli utenti e i client delle applicazioni notano solo una breve interruzione del servizio.

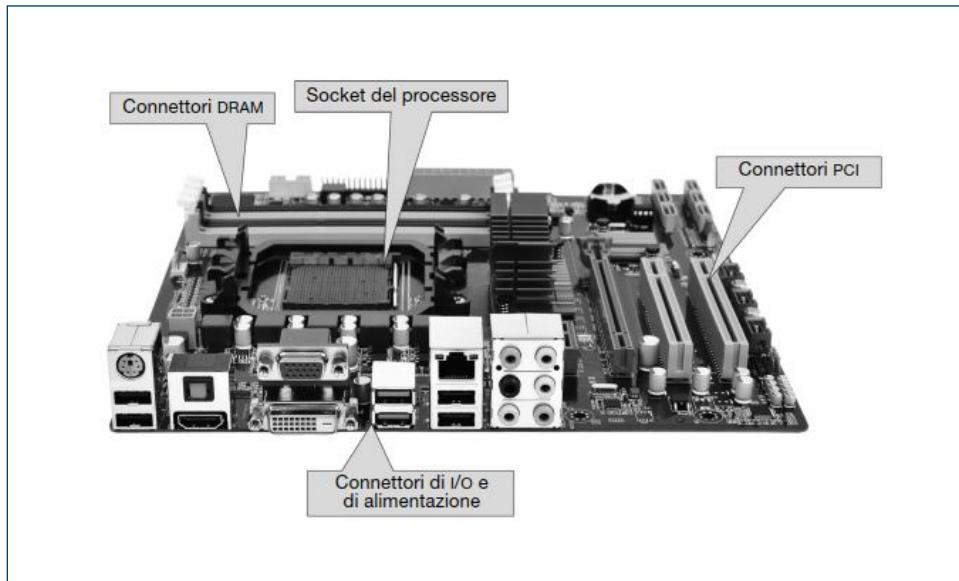
Per molte applicazioni una maggiore affidabilità dell'elaboratore è di importanza cruciale. La capacità di continuare a offrire un servizio proporzionale alla quantità di hardware ancora in funzione è detta degrado controllato (*graceful degradation*). Taluni sistemi si spingono oltre il degrado controllato e sono chiamati tolleranti ai guasti (*fault-tolerant*), perché continuano a funzionare

ugualmente nonostante il guasto di un qualunque singolo componente. La tolleranza ai guasti necessita di un meccanismo per il riconoscimento del danno, la sua diagnosi e, se è possibile, la correzione.

I cluster di elaboratori sono strutturabili in modo sia asimmetrico sia simmetrico. Nei cluster asimmetrici un calcolatore rimane nello stato di attesa attiva (*hot standby mode*) mentre l'altro esegue le applicazioni. Il calcolatore in *hot standby* non fa altro che monitorare il server attivo. Se questo presenta un problema, il calcolatore di controllo diventa il server attivo. Nei cluster simmetrici due o più calcolatori eseguono le applicazioni e allo stesso tempo si controllano reciprocamente; in questo modo si ottiene una maggiore efficienza, poiché si utilizzano meglio le risorse, ma ciò richiede che siano disponibili più applicazioni da eseguire.

LA SCHEDA MADRE DI UN PC

Osserviamo la scheda madre di un pc desktop con un singolo socket per il processore, mostrata di seguito.



Questa scheda diventa un computer perfettamente funzionante una volta che i suoi connettori (slot) vengono popolati.

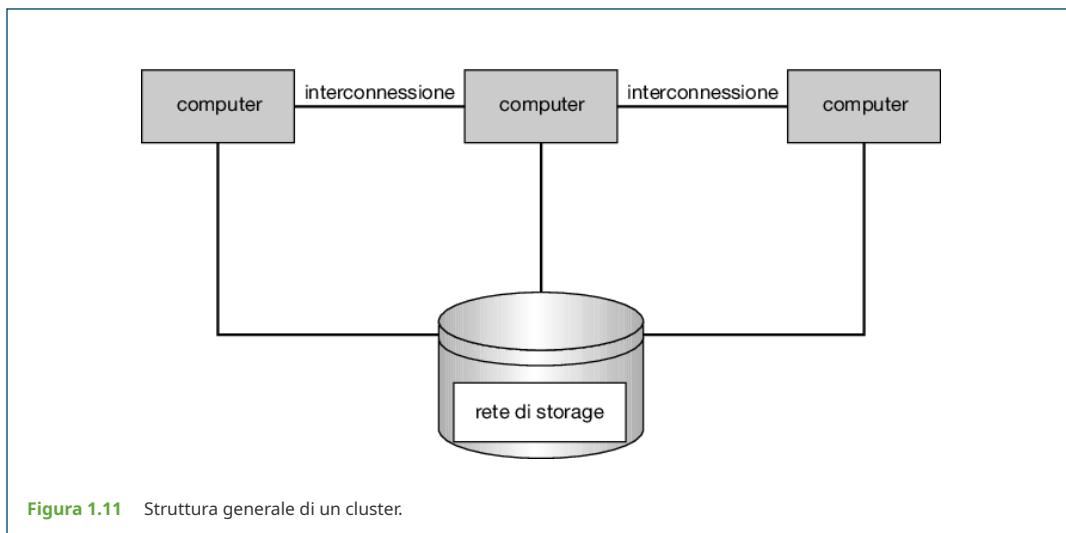
La scheda dispone di un socket del processore in grado di contenere una cpu, di alcuni connettori dram, di connettori pcie e di connettori i/o di vario tipo. Oggi, anche le cpu general-purpose più economiche contengono più unità di calcolo (core). Alcune schede madri contengono più socket del processore e i computer più avanzati consentono di avere più di una scheda di sistema, permettendo così di costruire sistemi numa.

I cluster, essendo formati da diversi sistemi di computer collegati in rete, possono anche essere utilizzati per ottenere ambienti di elaborazione ad alte prestazioni (*high-performance computing*). Sistemi di questo tipo offrono molta più potenza elaborativa rispetto a un monoprocesso o persino rispetto a sistemi smp, perché permettono l'esecuzione contemporanea di un'applicazione su tutti i computer del cluster. Tuttavia, le applicazioni devono essere scritte specificatamente in modo da trarre vantaggio dal cluster sfruttando una tecnica chiamata parallelizzazione (*parallelization*). Essa consiste nel suddividere il programma in componenti separate, eseguibili in parallelo su singoli computer all'interno del cluster. In genere tali applicazioni sono progettate in modo tale che, una volta che ogni nodo di elaborazione del cluster abbia risolto la sua porzione di problema, tutti i risultati vengano combinati in un'unica soluzione finale.

Altre forme di clustering sono i cluster paralleli e quelli di sistemi connessi attraverso reti geografiche (wan), come descritto nel Capitolo 19. I primi permettono a più calcolatori di accedere agli stessi dati nella memoria di massa condivisa. Poiché la maggior

parte dei sistemi operativi non supporta quest'accesso simultaneo ai dati da parte di più calcolatori, si ricorre a programmi specifici e particolari versioni delle applicazioni. Per esempio, l'*Oracle Real Application Cluster* è una versione del sistema di gestione delle basi di dati Oracle, progettata per funzionare su cluster paralleli. Ogni calcolatore esegue l'applicazione Oracle e uno strato software controlla l'accesso ai dischi condivisi, in questo modo ogni calcolatore del sistema ha accesso alla base di dati. Per ottenere questo accesso condiviso ai dati, il sistema deve anche prevedere il controllo dell'accesso e la mutua esclusione, in modo da evitare sul nascere i conflitti tra operazioni. Questa funzione, detta gestione distribuita dei lock (*distributed lock manager, dlm*), è fornita da alcune tecnologie di clustering.

La tecnologia in quest'ambito sta attraversando una fase di rapida evoluzione. Alcuni prodotti di questo genere ospitano dozzine di sistemi in un solo cluster e possono accoppare in un unico cluster nodi distanti chilometri. Tali progressi si devono in buona misura a reti di storage (*storage-area network, san*), illustrate nel Paragrafo 11.7.4, che permettono a molti sistemi di accedere a un'unica area di memorizzazione secondaria. Se le applicazioni e i relativi dati sono memorizzati in una san, il software del cluster può smistare l'applicazione verso una qualunque delle macchine che vi hanno accesso. Qualora venga meno una di loro, può subentrare qualsiasi altra macchina. Se una base di dati è implementata su un cluster, decine di macchine possono condividerne il contenuto, con notevole aumento delle prestazioni e dell'affidabilità. La Figura 1.11 rappresenta la struttura generale di un cluster.



1.4 Attività del sistema operativo

Avendo passato in rassegna l'organizzazione e l'architettura dei sistemi informatici siamo pronti ad affrontare i sistemi operativi. Il sistema operativo costituisce l'ambiente di esecuzione dei programmi. Sebbene la struttura dei sistemi operativi possa variare grandemente, vi sono alcuni aspetti comuni che esponiamo in questo paragrafo.

L'avviamento del sistema, conseguente all'accensione del calcolatore, così come il riavvio di un calcolatore già acceso, richiedono la presenza di uno specifico programma iniziale. Come già accennato, questo programma, detto programma di avviamento (*bootstrap program*), è di solito piuttosto semplice. Normalmente è memorizzato nel *firmware* che fa parte dell'hardware del calcolatore. La sua funzione consiste nell'inizializzare i diversi componenti del sistema, dai registri della cpu ai controllori dei diversi dispositivi, fino al contenuto della memoria centrale. Il programma di avviamento deve caricare il sistema operativo e avvarne l'esecuzione; a tale scopo deve individuare e caricare nella memoria il kernel del sistema operativo.

Una volta caricato e in esecuzione, il kernel può iniziare a offrire servizi al sistema e agli utenti. Alcuni servizi vengono forniti all'esterno del kernel da programmi di sistema caricati in memoria durante l'avviamento. Questi *processi di sistema*, o *demoni*, restano in esecuzione per tutto il tempo in cui è in esecuzione il kernel. In Linux il primo processo di sistema è "system", che avvia diversi altri demoni. Una volta che questa fase è completata il sistema risulta completamente inizializzato e attende che si verifichi qualche evento.

HADOOP

Hadoop è un ambiente open source utilizzato per l'elaborazione distribuita di insiemi di dati di grandi dimensioni (noti come **big data**) in un sistema cluster contenente componenti hardware semplici e a basso costo. Hadoop è progettato per scalare da un sistema singolo fino a un cluster contenente migliaia di nodi di calcolo. A ogni nodo nel cluster viene assegnata un'attività e Hadoop organizza la comunicazione tra i nodi per gestire il calcolo parallelo in grado di elaborare e fondere i risultati. Hadoop fornisce un servizio di calcolo distribuito efficiente e altamente affidabile, ed è in grado di rilevare e gestire i guasti nei nodi.

Hadoop è organizzato attorno alle tre componenti che seguono.

- Un file system distribuito che gestisce dati e file su nodi di calcolo distribuiti.
- Il framework yarn ("Yet Another Resource Negotiator"), che gestisce le risorse all'interno del cluster e pianifica le attività sui nodi nel cluster.
- Il sistema MapReduce, che consente l'elaborazione parallela dei dati tra i nodi del cluster.

Hadoop è progettato per funzionare su sistemi Linux; le applicazioni Hadoop possono essere scritte utilizzando diversi linguaggi di programmazione, tra cui linguaggi di scripting come PHP, Perl e Python. Una scelta frequente per lo sviluppo di applicazioni Hadoop è il linguaggio Java, perché Hadoop ha diverse librerie Java che supportano MapReduce. Maggiori informazioni su MapReduce e Hadoop sono disponibili su

<https://hadoop.apache.org/docs/r1.2.1/mapred.tutorial.html> e <https://hadoop.apache.org>

Se i dispositivi di i/o non richiedono alcun servizio, in assenza di processi da eseguire e di utenti a cui rispondere, il sistema operativo rimane inerte e attende che accada qualcosa. Un evento è quasi sempre segnalato da un'interruzione. Abbiamo descritto le interruzioni hardware nel Paragrafo 1.2.1. Un'altra forma di interruzioni è data dalle eccezioni (trap o exception) ossia interruzioni generate dal software causate da un errore (per esempio, una divisione per zero o l'accesso illegale alla memoria), o dalla richiesta di erogazione, da parte di un programma utente, di uno dei servizi del sistema operativo, realizzata mediante l'esecuzione di una speciale operazione detta chiamata di sistema (*system call*).

1.4.1 Multiprogrammazione e multitasking

Fra le principali caratteristiche dei sistemi operativi vi è la multiprogrammazione. In generale, un singolo programma non è in grado di tenere costantemente occupati la cpu e i dispositivi di i/o. Inoltre, gli utenti vogliono solitamente eseguire più programmi allo

stesso tempo. La multiprogrammazione, oltre a soddisfare questa esigenza degli utenti, consente di aumentare la percentuale di utilizzo della cpu, organizzando i programmi in modo tale da mantenerla in continua attività. In un sistema multiprogrammato un programma in esecuzione è chiamato *processo*.

L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale diversi processi (Figura 1.12). Il sistema operativo ne sceglie uno e inizia l'esecuzione: a un certo punto il processo potrebbe trovarsi nell'attesa di qualche evento, come il completamento di un'operazione di i/o. In questi casi, in un sistema non multiprogrammato, la cpu rimarrebbe inattiva. In un sistema con multiprogrammazione, invece, il sistema operativo passa semplicemente a un altro processo e lo esegue. Quando quest'ultimo deve a sua volta attendere, la cpu passa ancora a un altro processo. Quando il primo processo ha terminato l'attesa, la cpu ne riprende l'esecuzione. Finché c'è almeno un processo da eseguire, la cpu non è mai inattiva.

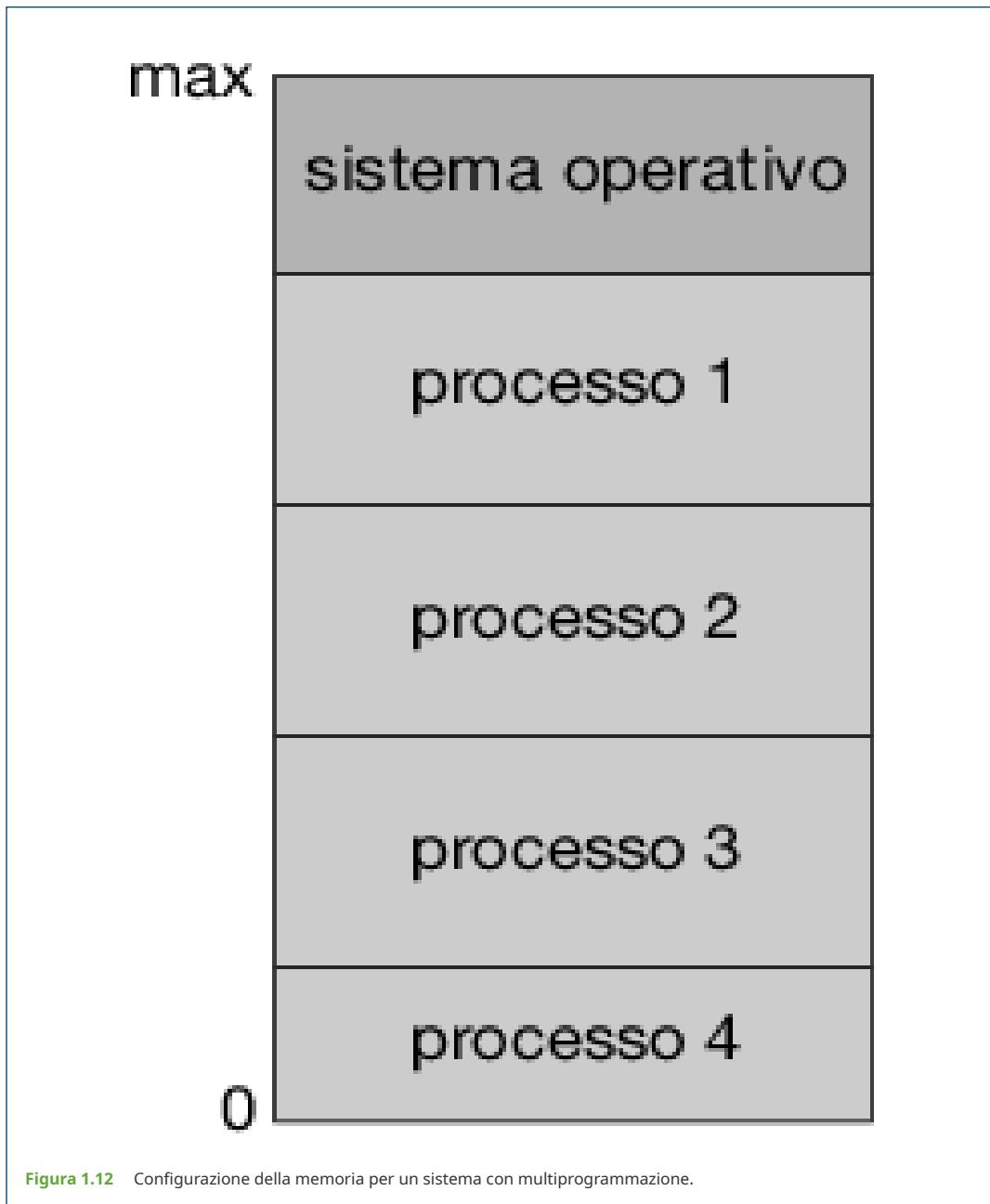


Figura 1.12 Configurazione della memoria per un sistema con multiprogrammazione.

Ritroviamo quest'idea anche in altre circostanze della vita comune. Un avvocato, per esempio, non lavora per un solo cliente alla volta: mentre un caso aspetta di essere dibattuto o si attende la stesura dei relativi documenti, l'avvocato può lavorare a un altro caso;

se ha abbastanza clienti, non sarà mai inattivo per mancanza di lavoro. (Gli avvocati inattivi tendono a trasformarsi in politici, quindi tenerli occupati ha un certo valore sociale).

Il multitasking è un'estensione logica della multiprogrammazione; la cpu esegue più lavori commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente di usufruire di tempi di risposta rapidi. Normalmente un processo, durante la sua esecuzione, impega la cpu per un breve periodo di tempo prima di terminare o di richiedere operazioni di i/o; tali operazioni possono essere interattive, cioè i risultati sono inviati a uno schermo a disposizione dell'utente, che a sua volta immette dati tramite una tastiera, un mouse o un touch-screen. La velocità delle operazioni di immissione dipende dai tempi tipici degli esseri umani; l'immissione di dati e comandi tramite una tastiera, per esempio, è limitata dalla velocità di battitura dell'utente: sette caratteri al secondo sono tanti per una persona, ma pochissimi per un calcolatore. Anziché lasciare inattiva la cpu, durante l'immissione interattiva il sistema operativo commuta rapidamente la cpu a un altro processo.

La coesistenza di un certo numero di programmi in memoria nello stesso lasso di tempo richiede una qualche forma di gestione della memoria, argomento esposto nei Capitoli 9 e 10. Inoltre, l'esistenza di diversi processi pronti per l'esecuzione nello stesso istante impone al sistema di scegliere quale processo viene eseguito per primo: il modo in cui viene effettuata questa decisione, ovvero lo scheduling della cpu, è descritto nel Capitolo 5. Infine, l'esecuzione concorrente di più processi rende necessario limitare le loro interferenze reciproche in ogni aspetto del funzionamento del sistema, compresi lo scheduling dei processi e la gestione del disco e della memoria. Tali problematiche sono affrontate di volta in volta nel libro.

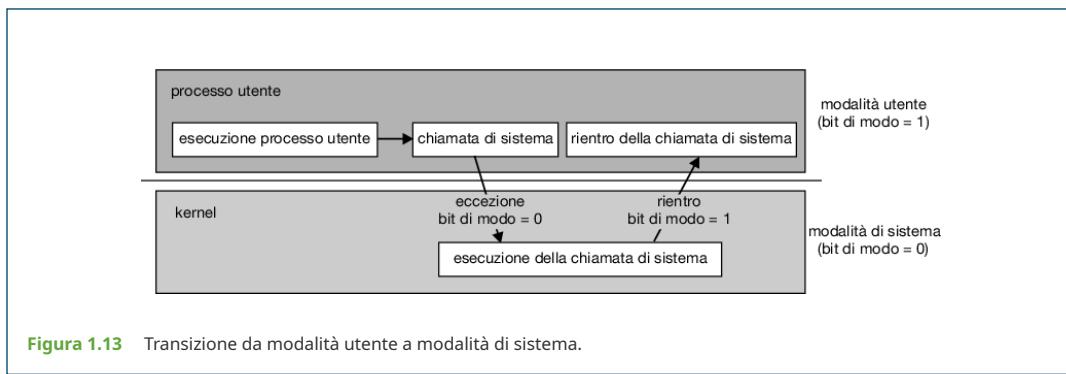
In un sistema multitasking, il sistema operativo deve garantire tempi di risposta accettabili. Un metodo comune per ottenere questo risultato è la memoria virtuale, tecnica che consente l'esecuzione di programmi anche non interamente caricati nella memoria (Capitolo 10). Il più evidente vantaggio della memoria virtuale è che i programmi possono avere dimensioni maggiori della memoria fisica; inoltre, essa astrae la memoria centrale in un grande e uniforme vettore, separando la memoria logica, vista dall'utente, dalla memoria fisica, sollevando i programmati dai problemi legati ai limiti della memoria.

I sistemi multiprogrammati e multitasking devono inoltre fornire un *file system* (Capitoli 13, 14 e 15) residente in una memoria secondaria, la quale a sua volta necessita di una gestione (Capitolo 11). Inoltre un sistema deve disporre di meccanismi per la protezione delle risorse rispetto a utilizzi non autorizzati (Capitolo 17). Per assicurare un'esecuzione disciplinata il sistema deve fornire meccanismi per la comunicazione e sincronizzazione dei processi (Capitoli 6 e 7) e garantire che i processi non si blocchino in una situazione di stallo, in un'indefinita attesa reciproca (Capitolo 8).

1.4.2 Duplice modalità di funzionamento

Dal momento che il sistema operativo e i suoi utenti condividono le risorse hardware e software del sistema, un sistema operativo progettato correttamente deve assicurarsi che un programma errato (o volutamente dannoso) non possa causare il malfunzionamento di altri programmi o del sistema operativo stesso. Al fine di garantire la corretta esecuzione del sistema dobbiamo essere in grado di distinguere tra l'esecuzione del codice del sistema operativo e l'esecuzione del codice utente. L'approccio adottato dalla maggior parte dei sistemi elaborativi è quello di fornire un supporto hardware che consenta la differenziazione tra le varie modalità di esecuzione.

Sono necessarie almeno due diverse modalità: modalità utente e modalità di sistema (detta anche *modalità kernel*, *modalità supervisore*, *modalità monitor* o *modalità privilegiata*). Per indicare quale sia la modalità attiva, l'architettura della cpu deve essere dotata di un bit, chiamato appunto bit di modalità: kernel (0) o user (1). Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente. Quando l'elaboratore agisce per conto di un'applicazione utente, il sistema è in modalità utente. Tuttavia, quando l'applicazione utente richiede un servizio al sistema operativo (tramite una chiamata di sistema), per soddisfare la richiesta il sistema deve passare dalla modalità utente alla modalità di sistema. Ciò è esemplificato dalla Figura 1.13. Come vedremo, questa miglioria architettonica agevola il funzionamento del sistema anche in altre circostanze.



All'avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all'esecuzione delle applicazioni utenti in modalità utente. Ogni volta che si verifica un'interruzione o un'eccezione, l'hardware passa dalla modalità utente a quella di sistema, cioè pone a 0 il bit di modo. Perciò, quando il sistema operativo riprende il controllo del calcolatore, esso si trova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

La duplice modalità di funzionamento (*dual mode*) consente la protezione del sistema operativo e degli altri utenti dagli errori di un utente. Questo livello di protezione si ottiene definendo come istruzioni privilegiate le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché l'hardware consente l'esecuzione di queste istruzioni soltanto nella modalità di sistema, se si tenta di far eseguire in modalità utente un'istruzione privilegiata, l'hardware non la esegue e passa il controllo con una eccezione al sistema operativo.

Un esempio di istruzione privilegiata è dato dall'istruzione per passare alla modalità kernel. Altri esempi si riferiscono al controllo dell'i/o, alla gestione dei timer e delle interruzioni. Come si vedrà in seguito, vi sono molte altre istruzioni privilegiate che verranno discusse nel testo.

Il concetto di modalità può essere esteso a più di due possibilità. Per esempio, i processori Intel hanno quattro livelli di protezione separati, detti protection ring, dove ring0 è la modalità kernel e ring3 è la modalità utente. (Sebbene ring1 e ring2 possano essere usati per vari servizi del sistema operativo, in pratica sono usati raramente). I sistemi armv8 hanno sette modalità. Le cpu che supportano la virtualizzazione (si veda il Paragrafo 18.1) hanno spesso una modalità distinta per indicare che il gestore della macchina virtuale (*virtual machine manager*, vmm) ha il controllo del sistema. In questa modalità, il vmm ha più privilegi dei processi utente, ma meno del kernel. Questo livello di privilegio è necessario per creare e gestire le macchine virtuali, modificando lo stato della cpu.

Siamo ora in grado di capire meglio il "ciclo di vita" dell'esecuzione delle istruzioni in un elaboratore. All'inizio il controllo appartiene al sistema operativo, dove le istruzioni sono eseguite in modalità kernel. Quando il controllo viene ceduto a un'applicazione utente si entra nella modalità utente. Alla fine, il controllo viene restituito al sistema operativo mediante un'interruzione, un'eccezione o una chiamata di sistema. I sistemi operativi recenti, come Microsoft Windows, Unix e Linux, sfruttano questa funzionalità dual-mode per offrire una maggior protezione.

Le chiamate di sistema sono gli strumenti con cui un programma utente richiede al sistema operativo di compiere operazioni a esso riservate, per conto del programma utente. Vi sono vari modi per generare una chiamata di sistema, a seconda delle funzionalità di cui dispone il processore. In ogni caso, però, le chiamate di sistema sono il mezzo utilizzato dai processi per richiedere un'azione al sistema operativo. Una chiamata di sistema è solitamente realizzata come un'eccezione che rimanda a un indirizzo specifico nel vettore delle interruzioni. A tale eccezione si può dare esecuzione con un'istruzione `trap` generica, sebbene alcuni sistemi abbiano un'istruzione `syscall` dedicata.

Quando un programma utente esegue una chiamata di sistema, questa è solitamente gestita dalla cpu come un'interruzione software. Il controllo passa, tramite il vettore delle interruzioni, all'apposita procedura di servizio nel sistema operativo e si pone il bit di modo in modalità kernel. La procedura di servizio della chiamata di sistema è parte integrante del sistema operativo. Il sistema esamina l'istruzione che ha causato la chiamata, al fine di stabilirne la natura, mentre un parametro di tale istruzione definisce il tipo di servizio richiesto dal programma utente. Ulteriori informazioni indispensabili per il completamento della richiesta si possono copiare nei registri, sullo stack o direttamente nella memoria centrale (in locazioni il cui indirizzo è memorizzato nei registri). Il sistema verifica correttezza e legalità dei parametri, soddisfa la richiesta e restituisce il controllo dell'esecuzione all'istruzione immediatamente seguente alla chiamata di sistema. Il Paragrafo 2.3 approfondisce il concetto di chiamata di sistema.

Una volta che la protezione hardware sia attiva gli errori di violazione della modalità sono rilevati dall'hardware stesso, e di norma sono gestiti dal sistema operativo. Se un programma utente causa un errore (per esempio tentando di eseguire un'istruzione illegale o di accedere a una zona di memoria che esula dallo spazio degli indirizzi dell'utente) l'hardware genera un'eccezione. L'eccezione cede il controllo, attraverso il vettore delle interruzioni, al sistema operativo, cioè segue una condotta analoga a quanto farebbe un'interruzione. Quando si imbatte nell'errore di un programma, il sistema operativo deve terminare il programma in maniera anomala. La cosa è gestita dal medesimo codice preposto alla terminazione anomala di un processo a seguito di una richiesta dell'utente: si genera un appropriato messaggio di errore, e si rilascia la memoria del programma incriminato. Il contenuto della memoria rilasciata è solitamente trascritto su un file (*memory dump*), perché l'utente o il programmatore possano esaminarlo ed eventualmente correggerlo in vista di un nuovo utilizzo del programma.

1.4.3 Timer

Occorre assicurare che il sistema operativo mantenga il controllo della cpu, che consiste nell'impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo. A tale scopo si può usare un timer, programmabile affinché invii un segnale d'interruzione alla cpu a intervalli di tempo specificati, che possono essere fissi (per esempio, di 1/60 di secondo) o variabili (per esempio, da un millisecondo a un secondo). Un timer variabile di solito si realizza mediante un clock a frequenza fissa e un contatore. Il sistema operativo assegna un valore al contatore, che si decrementa a ogni impulso e quando raggiunge il valore 0 si genera un segnale d'interruzione. Per esempio, un contatore di 10 bit con un clock con periodo di 1 millisecondo consente la generazione di interruzioni a intervalli compresi tra 1 e 1024 millisecondi, con incrementi di 1 millisecondo.

Prima di restituire all'utente il controllo dell'esecuzione, il sistema assegna un valore al timer. Se esso esaurisce questo intervallo genera un'interruzione che causa il trasferimento del controllo al sistema operativo, che può decidere se gestire l'interruzione come un errore fatale o concedere altro tempo al programma. Ovviamente, anche le istruzioni usate dal sistema per modificare il valore del timer si possono eseguire soltanto in modalità privilegiata.

LA TEMPORIZZAZIONE IN LINUX

Sui sistemi Linux il parametro di configurazione del kernel `Hz` specifica la frequenza delle interruzioni del timer. Un valore di `Hz` pari a 250 indica che il timer genera 250 interruzioni al secondo, ovvero un'interruzione ogni 4

millisecondi. Il valore di `Hz` dipende dalla configurazione del kernel, dal tipo di macchina e dall'architettura su cui è in esecuzione. Una variabile correlata ad `Hz` è `jiffies`, che rappresenta il numero di interruzioni del timer che si sono verificate dall'avvio del sistema. Un progetto di programmazione nel Capitolo 2 approfondirà la temporizzazione nel kernel di Linux.

1.5 Gestione delle risorse

Come abbiamo visto, il sistema operativo è un gestore di risorse che deve gestire, per esempio, la cpu del sistema, lo spazio di memoria, le memorie secondarie e le periferiche di i/o.

1.5.1 Gestione dei processi

Un programma fa qualcosa soltanto se la cpu esegue le istruzioni che lo costituiscono. Come abbiamo detto, un programma in esecuzione è un processo. Un programma utente, come un compilatore, eseguito in un ambiente time-sharing, è un processo; un programma d'elaborazione di testi eseguito da un pc per un singolo utente è un processo; così come lo è un servizio di sistema, per esempio l'invio di dati a una stampante. Per il momento ci si può limitare a considerare un processo come un lavoro d'elaborazione (*job*) o un programma eseguito in un ambiente time-sharing, anche se il concetto è più generale. Come si descrive nel Capitolo 3, si possono avere chiamate di sistema che permettono ai processi di creare sottoprocessi da eseguire in modo concorrente.

Per svolgere i propri compiti, un processo necessita di alcune risorse, tra cui tempo di cpu, memoria, file e dispositivi di i/o. Queste risorse si possono attribuire al processo al momento della sua creazione, oppure si possono assegnare durante l'esecuzione. Oltre alle diverse risorse fisiche e logiche assegnate a un processo durante la sua creazione, si possono considerare anche alcuni dati d'inizializzazione da passare di volta in volta al processo stesso. Per esempio, un processo che ha lo scopo di mostrare su uno schermo lo stato di un file riceve il nome del file ed esegue le appropriate istruzioni e chiamate di sistema che gli consentono di ottenere e mostrare sul terminale le informazioni desiderate; quando il processo termina, il sistema operativo riprende il controllo delle risorse impiegate dal processo.

Bisogna sottolineare che un programma di per sé *non* è un processo d'elaborazione; un programma è un'entità *passiva*, come il contenuto di un file memorizzato in un disco, mentre un processo è un'entità *attiva*. Un processo a singolo *thread* ha un contatore di programma che indica la successiva istruzione da eseguire (i thread sono trattati nel Capitolo 4). L'esecuzione di tale processo deve essere sequenziale: la cpu esegue le istruzioni del processo una dopo l'altra, finché il processo termina; inoltre in ogni istante si esegue al massimo un'istruzione del processo. Quindi, anche se due processi possono corrispondere allo stesso programma, si considerano in ogni caso due sequenze d'esecuzione separate. Un processo multithread possiede più contatori di programma, ognuno dei quali punta all'istruzione successiva da eseguire per un dato thread.

Il processo è l'unità di lavoro di un sistema. Un sistema è costituito di un gruppo di processi, alcuni dei quali del sistema operativo (eseguono codice di sistema), mentre altri sono processi utenti (eseguono codice utente). Tutti questi processi si possono potenzialmente eseguire in modo concorrente, per esempio avviciandosi nell'uso di una singola unità di calcolo o in parallelo su diverse unità.

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei processi:

- creazione e cancellazione dei processi utenti e di sistema;
- schedulazione di processi e thread sulle cpu;
- sospensione e ripristino dei processi;
- fornitura di meccanismi per la sincronizzazione dei processi;
- fornitura di meccanismi per la comunicazione tra processi.

Le tecniche di gestione dei processi sono trattate nei Capitoli dal 3 al 7.

1.5.2 Gestione della memoria

Come si è accennato nel Paragrafo 1.2.2, la memoria centrale è fondamentale per il funzionamento di un moderno sistema elaborativo. Si tratta di un vasto vettore di dimensioni che vanno dalle centinaia di migliaia ai miliardi di parole, ciascuna delle quali è dotata del proprio indirizzo. È un archivio di dati velocemente accessibile, condiviso dalla cpu e dai dispositivi di i/o. La cpu legge le istruzioni dalla memoria centrale durante il ciclo di prelievo delle istruzioni, oltre a leggere e scrivere i dati nella memoria centrale durante il ciclo d'accesso ai dati (su di un'architettura Von Neumann). Generalmente la memoria centrale è l'unico ampio dispositivo di memorizzazione a cui la cpu può far riferimento e accedere in modo diretto. Per esempio, affinché la cpu possa gestire i dati di un disco, occorre che essi siano prima trasferiti nella memoria centrale attraverso le richieste di i/o generate dalla cpu. In modo analogo, la cpu può eseguire le istruzioni solo se si trovano nella memoria.

Per eseguire un programma è necessario che questo sia associato a indirizzi assoluti e sia caricato nella memoria. Durante l'esecuzione del programma, questo accede alle proprie istruzioni e ai dati provenienti dalla memoria, generando i suddetti indirizzi assoluti. Quando il programma termina, si rende disponibile il suo spazio di memoria; a questo punto si può caricare ed eseguire il programma successivo.

Per migliorare l'utilizzo della cpu e la rapidità con la quale il calcolatore risponde ai propri utenti i computer general-purpose devono tenere molti programmi in memoria. Esistono diversi schemi di gestione della memoria; l'efficacia di ogni algoritmo dipende dalla situazione specifica. La scelta di un particolare schema di gestione della memoria dipende da molti fattori, principalmente dal tipo di *architettura hardware* del sistema.

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione della memoria centrale:

- tenere traccia di quali parti di memoria sono attualmente usate e da chi;
- assegnare e revocare lo spazio di memoria secondo le necessità;
- decidere quali processi (o parti di essi) e dati debbano essere caricati in memoria centrale o trasferiti sulla memoria di massa.

Le tecniche di gestione della memoria sono trattate nei Capitoli 9 e 10.

1.5.3 Gestione dei file

Per facilitare gli utenti, il sistema operativo fornisce un'interfaccia logica uniforme per la memorizzazione delle informazioni. Esso, cioè, prescinde dalle caratteristiche fisiche dei dispositivi di memorizzazione, definendo un'unità logica di archiviazione, il file. Il sistema operativo associa i file a supporti fisici, e vi accede tramite i dispositivi di memorizzazione delle informazioni.

La gestione dei file è uno dei componenti più visibili di un sistema operativo. I calcolatori possono registrare le informazioni su molti mezzi fisici diversi. Ciascuno ha caratteristiche proprie e una propria organizzazione fisica, ed è controllato da un dispositivo, come un'unità a disco, avente anch'esso caratteristiche proprie. Queste proprietà comprendono velocità, capacità, rapidità nel trasferimento dei dati, metodi d'accesso (diretto o sequenziale).

Un file è una raccolta d'informazioni correlate definita dal suo creatore. Comunemente, i file rappresentano programmi, sia sorgente sia oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari; la loro forma può essere libera, come nei file di testo, oppure rigidamente formattata, per esempio in campi fissi come nel caso dei file mp3. Il concetto di file è quindi molto generale.

Il sistema operativo realizza il concetto astratto di file gestendo i mezzi di memoria di massa e i dispositivi che li controllano. I file sono generalmente organizzati in directory, che ne facilitano l'uso. Infine, se più utenti hanno accesso ai file, si potrebbe voler controllare chi ha la possibilità di accedervi e in che modo (per esempio, lettura, scrittura, aggiunta).

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei file:

- creazione e cancellazione di file;
- creazione e cancellazione di directory;
- fornitura delle funzioni fondamentali per la gestione di file e directory;
- associazione dei file ai dispositivi di memoria secondaria;
- creazione di copie di riserva (*backup*) dei file su dispositivi di memorizzazione non volatili.

Le tecniche di gestione dei file sono trattate nei Capitoli 13, 14 e 15.

1.5.4 Gestione della memoria di massa

Come si è visto, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. La maggior parte dei moderni sistemi elaborativi impiega dischi magnetici e nvm come principali mezzi di memorizzazione secondaria, sia per i programmi sia per i dati. I programmi si servono del disco sia come sorgente sia come destinazione delle loro elaborazioni. Per tale ragione è fondamentale che la registrazione nella memoria secondaria sia gestita correttamente. Il sistema operativo è responsabile delle seguenti attività connesse alla gestione della memoria di massa:

- montare (*mounting*) e smontare (*unmounting*) le unità di memoria;
- gestione dello spazio libero;
- assegnazione dello spazio;
- scheduling del disco;
- partizionamento;
- protezione.

Il frequente uso della memoria secondaria impone una sua gestione efficiente. Infatti, l'efficienza complessiva di un calcolatore può dipendere dalla velocità del sottosistema di gestione dei dischi e dagli algoritmi che lo gestiscono.

Vi sono però svariati frangenti in cui tornano utili dispositivi di memorizzazione più lenti, meno costosi e talora più capienti della memoria secondaria. Le copie di riserva dei dischi di sistema, i dati usati raramente e gli archivi a lungo termine sono alcuni esempi. Le unità a nastro magnetico, i cd, i dvd e i Blu-ray sono tipici dispositivi di memoria terziaria.

La memoria terziaria ha scarso impatto sulle prestazioni del sistema, ma deve pur essere gestita. Alcuni sistemi si assumono tale onere direttamente, mentre altri lo delegano a programmi applicativi. Tra le funzioni che i sistemi possono fornire citiamo l'installazione e la rimozione dei media dei dispositivi, l'allocazione dei dispositivi ai processi che ne richiedano l'uso esclusivo e il trasferimento alla memoria terziaria dei dati provenienti da quella secondaria.

Le tecniche di gestione della memoria secondaria e terziaria saranno approfondate nel Capitolo 11.

1.5.5 Gestione della cache

Il concetto di cache è un principio importante di un sistema elaborativo. Di norma le informazioni sono mantenute in un sistema di memoria come la memoria centrale; al momento del loro uso si copiano temporaneamente in un'unità più veloce: la cache. Quando si deve accedere a una particolare informazione, innanzitutto si controlla se è già presente all'interno della cache; in tal caso si adopera direttamente la copia contenuta nella cache, altrimenti la si preleva dalla memoria centrale e la si copia nella cache, poiché si suppone che questa informazione presto servirà ancora.

Inoltre, i registri programmabili presenti all'interno della cpu, come i registri indice, rappresentano per la memoria centrale una cache ad alta velocità. Il programmatore (o il compilatore) implementa gli algoritmi di assegnazione e aggiornamento dei registri in modo da stabilire quali informazioni mantenere nei registri e quali nella memoria centrale. Esistono anche cache che sono interamente gestite dall'hardware del sistema: per esempio la maggior parte dei sistemi è dotata di una cache per la memorizzazione delle istruzioni che presumibilmente saranno eseguite dopo l'istruzione corrente. Senza di essa, la cpu dovrebbe attendere parecchi cicli prima che un'istruzione sia prelevata dalla memoria. Per motivi analoghi, la maggior parte dei sistemi è dotata di una o più cache di dati nella gerarchia delle memorie. In questo testo non ci si occupa di tali dispositivi, poiché sono componenti non controllabili dal sistema operativo.

Data la capacità limitata di questi dispositivi, la gestione della cache è un importante problema di progettazione. Da un'attenta selezione delle dimensioni e dei criteri di aggiornamento della cache può conseguire un notevole incremento delle prestazioni del

sistema. La Figura 1.14 presenta un confronto tra le prestazioni di vari dispositivi di memorizzazione in piccoli server e potenti stazioni di lavoro. La necessità delle memorie cache emerge chiaramente. Nel Capitolo 10 si discutono alcuni algoritmi per la sostituzione degli elementi contenuti nelle cache controllabili via software.

Livello	1	2	3	4	5
Nome	registri	cache	memoria centrale	disco a stato solido	disco magnetico
Dimensione tipica	< 1 kb	< 16 mb	< 64 gb	< 1 tb	< 10 tb
Tecnologia	memoria dedicata con porte multiple (cmos)	cmos sram (on-chip o off-chip)	cmos dram	memoria flash	disco magnetico
Tempo d'accesso (ns)	0,25 – 0,5	0,5 – 25	80 – 250	25.000-50.000	5.000.000
Ampiezza di banda (mb/s)	20.000 – 100.000	5000 – 10.000	1000 – 5000	500	20 – 150
Gestito da	compilatore	hardware	sistema operativo	sistema operativo	sistema operativo
Supportato da	cache	memoria centrale	disco	disco	disco o nastro

Figura 1.14 Caratteristiche di varie forme di archiviazione dei dati.

Il movimento delle informazioni tra i vari livelli della gerarchia può essere quindi sia implicito sia esplicito, a seconda dell'hardware e del sistema operativo che lo controlla. Per esempio, il trasferimento dei dati tra la cache e i registri della cpu è di solito svolto dall'hardware del sistema senza alcun intervento del sistema operativo. Invece il trasferimento dei dati dai dischi alla memoria è di solito gestito dal sistema operativo.

In una struttura gerarchica come quella appena introdotta può accadere che gli stessi dati siano mantenuti contemporaneamente in diversi livelli del sistema di memorizzazione. Per esempio, si supponga di dover incrementare di 1 il valore di un numero intero A contenuto in un file B, registrato in un disco magnetico. L'operazione d'incremento prevede innanzitutto l'esecuzione di un'operazione di i/o per copiare nella memoria centrale il blocco di disco contenente il valore di A. Questa operazione è seguita dalla copiatura di A all'interno della cache, e di qui in uno dei registri interni della cpu. Quindi, esistono diverse copie di A memorizzate nei vari dispositivi: nel disco magnetico, nella memoria centrale, nella cache e in un registro interno della cpu (Figura 1.15). Dopo l'incremento del valore della copia contenuta nel registro interno, il valore di A sarà diverso da quello assunto dalle altre copie della stessa variabile. Le diverse copie di A avranno lo stesso valore solamente dopo che il nuovo valore sarà stato riportato dal registro interno della cpu alla copia di A residente nel disco.

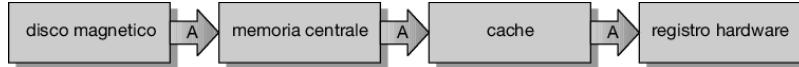


Figura 1.15 Migrazione di un intero A da un disco a un registro.

In un ambiente elaborativo che ammette l'esecuzione di un solo processo alla volta, questo modo di operare non pone particolari difficoltà, poiché ogni accesso ad A coinvolgerà sempre la copia posta al livello più alto della gerarchia. Viceversa, nei sistemi multitasking, in cui il controllo della cpu passa da un processo all'altro, è necessario prestare una particolare attenzione al fine di assicurare che ogni processo che desidera accedere ad A ottenga dal sistema il valore della variabile aggiornato più di recente.

La situazione si complica ulteriormente negli ambienti multiprocessore, dove ciascuna cpu, oltre ai registri interni, contiene anche una cache locale (si riveda la Figura 1.8). Nei sistemi del genere possono esistere più copie simultanee di A mantenute in cache differenti. Poiché le diverse unità d'elaborazione possono operare in parallelo, è necessario che l'aggiornamento del valore di una qualsiasi delle copie di A contenute nelle cache locali si rifletta immediatamente in tutte le cache in cui A risiede. Questa situazione, nota come coerenza della cache, di solito si risolve a livello dell'hardware del sistema, quindi a un livello più basso di quello del sistema operativo.

In un sistema distribuito la situazione diventa ancora più complessa, poiché può accadere che più copie (chiamate in questo caso repliche) dello stesso file siano mantenute in differenti calcolatori, dislocati in luoghi fisici diversi. Essendoci la possibilità di accedere e modificare in modo concorrente queste repliche, è necessario fare in modo che ogni modifica a una qualunque replica si rifletta quanto prima sulle altre. Nel Capitolo 19 si discutono diversi metodi che consentono di soddisfare questo requisito.

1.5.6 Gestione dell'i/o

Uno degli scopi di un sistema operativo è nascondere all'utente le peculiarità degli specifici dispositivi. In unix, per esempio, le caratteristiche dei dispositivi di i/o sono nascoste alla maggior parte dello stesso sistema operativo dal sottosistema di i/o, che è composto delle seguenti parti:

- un componente di gestione della memoria comprendente la gestione dei buffer di i/o, la gestione delle cache e la gestione delle aree di memoria per l'i/o asincrono (*spooling*);
- un'interfaccia generale per i driver dei dispositivi;
- i driver per gli specifici dispositivi.

Soltanto il driver del dispositivo conosce le peculiarità dello specifico dispositivo cui è assegnato.

In precedenza è stato presentato l'uso dei gestori delle interruzioni e dei driver dei dispositivi per la costruzione di sottosistemi di i/o efficienti. Nel Capitolo 12 si discute per esteso il modo in cui il sottosistema di i/o interagisce con gli altri componenti del sistema, come gestisce i dispositivi, trasferisce i dati e individua il completamento delle operazioni di i/o.

1.6 Sicurezza e protezione

Se diversi utenti usufruiscono dello stesso elaboratore che consente l'esecuzione concorrente di processi multipli, l'accesso ai dati dovrà essere disciplinato da regole. A tale scopo vi sono meccanismi che assicurano che i file, i segmenti di memoria, la cpu e le altre risorse possano essere manipolati solo dai processi che abbiano ottenuto apposita autorizzazione dal sistema operativo. Per esempio, l'hardware per l'indirizzamento della memoria garantisce il fatto che un processo sia eseguito solo entro il proprio spazio degli indirizzi. Il timer impedisce che un processo, dopo aver conquistato il controllo della cpu, non lo restituisca più al sistema operativo. I registri di controllo dei dispositivi non sono accessibili agli utenti, quindi l'integrità delle varie periferiche viene protetta.

Per protezione, quindi, si intende ciascun meccanismo di controllo dell'accesso alle risorse possedute da un elaboratore, da parte di processi o utenti. Le strategie di protezione devono fornire le specifiche dei controlli da attuare e gli strumenti per la loro effettiva applicazione.

La protezione può migliorare l'affidabilità rilevando errori nascosti alle interfacce tra i componenti dei sottosistemi. Il beneficio che spesso deriva da una tempestiva rilevazione degli errori nelle interfacce è di prevenire la contaminazione di un sottosistema sano a opera di un altro sottosistema infetto. Non proteggere una risorsa significa lasciare via libera al suo utilizzo (o all'utilizzo scorretto) da parte di utenti incompetenti o non autorizzati. Un sistema dotato di strategie di protezione offre i mezzi per distinguere l'uso autorizzato da quello non autorizzato, come si vedrà nel Capitolo 17.

Pur essendo dotato di protezione adeguata, un sistema può rimanere esposto agli accessi abusivi, rischiando malfunzionamenti. Si consideri un'utente le cui informazioni di autenticazione siano state trafugate. I suoi dati rischiano di essere copiati o cancellati, anche se è attiva la protezione dei file e della memoria. È compito della sicurezza difendere il sistema da attacchi provenienti dall'interno o dall'esterno. La varietà di minacce conosciute è enorme: si va da virus e worm, agli attacchi *denial-of-service* che paralizzano il servizio (appropriandosi di tutte le risorse del sistema e dunque escludendo da esso i legittimi utenti), a furto d'identità e sottrazione del servizio (uso non autorizzato di un sistema). Per alcuni sistemi l'attività di prevenzione da alcuni di questi attacchi è considerata una funzione del sistema, mentre ve ne sono altri che delegano la difesa a regole operative o a programmi aggiuntivi. A causa dell'allarmante incremento di incidenti legati alla sicurezza, le problematiche della sicurezza del sistema operativo costituiscono un settore che vede crescere rapidamente la ricerca e la sperimentazione. La sicurezza è analizzata nel Capitolo 16.

Protezione e sicurezza presuppongono che il sistema sia in grado di distinguere tra tutti i propri utenti. Nella maggior parte dei sistemi operativi è disponibile un elenco di nomi degli utenti e dei loro identificatori utente (user id). Nel gergo Windows, si parla di id di sicurezza (sid). Si tratta di id numerici che identificano univocamente l'utente. Quando un utente si collega al sistema, la fase di autenticazione determina l'id utente corretto. Tale id utente è associato a tutti i processi e i thread del soggetto in questione. Se l'utente ha necessità di leggere un id, il sistema lo riconverte in forma di nome utente grazie al suo elenco dei nomi degli utenti.

In certe circostanze è preferibile distinguere tra gruppi di utenti invece che tra utenti singoli. Per esempio, il proprietario di un file su un sistema unix potrebbe aver titolo a effettuare qualsiasi operazione su quel file, mentre un gruppo selezionato di utenti può essere abilitato soltanto alla lettura del file. Per ottenere questo, dobbiamo attribuire un nome al gruppo e identificare gli utenti che vi appartengono. La funzionalità è realizzabile creando un elenco, a livello di sistema, dei nomi dei gruppi e dei relativi identificatori di gruppo. Un utente può fare parte di uno o più gruppi, a seconda delle scelte compiute in sede di progettazione del sistema operativo. L'identificatore di gruppo è incluso in tutti i processi e i thread a esso relativi.

Durante il normale utilizzo del sistema, all'utente sono sufficienti un id utente e un id del gruppo. Tuttavia, gli utenti devono talvolta *scalare i privilegi*, ossia ottenere permessi ausiliari per certe attività; per esempio, un dato utente potrebbe aver bisogno di accedere a un dispositivo il cui uso è riservato. Vi sono vari metodi che permettono agli utenti di scalare i privilegi. In unix, per esempio, se è presente l'attributo `setuid` in un programma, esso potrà essere eseguito con l'id del proprietario del file, piuttosto che con quello dell'utente attuale. Il processo esegue con questo id effettivo finché non rinunci a questo privilegio, o termini.

1.7 Virtualizzazione

La virtualizzazione è una tecnica che permette di astrarre l'hardware di un singolo computer (la cpu, la memoria, le unità disco, le schede di interfaccia di rete e così via) in diversi ambienti di esecuzione, creando così l'illusione che ogni distinto ambiente sia in esecuzione sul suo proprio computer. Questi ambienti possono essere visti come diversi sistemi operativi (per esempio, Windows e unix) in esecuzione contemporaneamente e in grado di interagire l'uno con l'altro. Un utente di una macchina virtuale può passare da un sistema operativo a un altro esattamente come può commutare tra processi in esecuzione contemporaneamente in un singolo sistema operativo.

La virtualizzazione permette ai sistemi operativi di funzionare come applicazioni all'interno di altri sistemi operativi. Può sembrare, a prima vista, che ci siano poche ragioni per utilizzare questa tecnologia, ma il mercato della virtualizzazione è grande e in continua crescita, a testimonianza dell'utilità e dell'importanza di questa tecnologia.

In senso lato, la virtualizzazione appartiene a una tipologia di software che include anche l'emulazione. Quest'ultima tecnica viene utilizzata quando il tipo di cpu origine è diverso dal tipo di cpu destinazione. Per esempio, quando Apple è passata dalle cpu ibm Power alle cpu Intel x86 per i propri desktop e portatili, ha fornito un emulatore chiamato "Rosetta" per permettere di eseguire le applicazioni compilate per cpu ibm con i nuovi processori Intel. Lo stesso concetto può essere esteso per permettere a un intero sistema operativo scritto per una piattaforma di essere eseguito su una piattaforma diversa. L'emulazione tuttavia ha un costo piuttosto alto, poiché ogni istruzione che può essere eseguita nativamente sul sistema sorgente deve essere tradotta nell'equivalente funzione da eseguire sul sistema di destinazione e ciò richiede spesso l'utilizzo di diverse istruzioni di destinazione. Se la cpu origine e la cpu destinazione hanno prestazioni simili il codice emulato viene eseguito molto più lentamente rispetto al codice nativo.

Con la virtualizzazione invece un sistema operativo compilato per una particolare architettura viene eseguito all'interno di un altro sistema operativo progettato per la stessa cpu. La virtualizzazione fu inizialmente utilizzata su mainframe ibm come metodo per permettere a diversi utenti di eseguire task in maniera concorrente. L'esecuzione di diverse macchine virtuali permise (e permette tuttora) di eseguire task di diversi utenti su un sistema progettato per un solo utente. Più tardi, in risposta ad alcuni problemi nell'esecuzione contemporanea di più applicazioni Windows su cpu Intel x86, vmware creò una nuova tecnologia di virtualizzazione per Windows. Si trattava di un'applicazione che eseguiva una o più copie ospiti (guest) di Windows o altri sistemi operativi per x86, ognuna in grado di eseguire le sue applicazioni (si veda la Figura 1.16). Windows era l'applicazione ospitante (host) e l'applicazione vmware era il gestore della macchina virtuale (vmm). Compito del vmm era eseguire i sistemi operativi ospiti, gestire le loro risorse e proteggere ciascun ospite dagli altri.

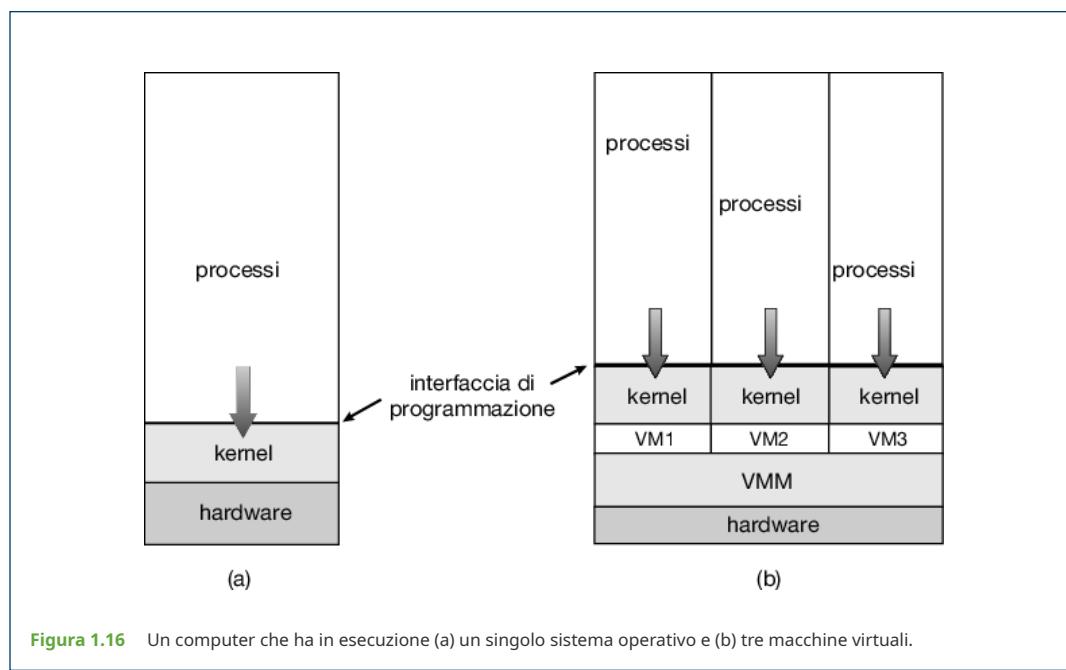


Figura 1.16 Un computer che ha in esecuzione (a) un singolo sistema operativo e (b) tre macchine virtuali.

Anche se i moderni sistemi operativi sono in grado di eseguire più applicazioni in maniera affidabile, l'uso della virtualizzazione è in continua crescita. Un vmm permette all'utente di installare su un computer portatile o desktop più sistemi operativi e di eseguire applicazioni scritte per sistemi operativi diversi da quello installato nativamente. Per esempio, un portatile Apple con un sistema macos in esecuzione su cpu x86 può mandare in esecuzione un sistema ospite Windows 10 per permettere l'esecuzione di applicazioni Windows. Le società che sviluppano software per diversi sistemi operativi possono utilizzare la virtualizzazione per eseguire tutti questi sistemi operativi su un unico server fisico destinato allo sviluppo, al test e al debugging. Nei data center la

virtualizzazione è diventata un metodo diffuso per eseguire e gestire gli ambienti elaborativi. Oggi i vmm come vmware, esx e Citrix XenServer non sono più soltanto programmi in esecuzione sul sistema host, ma svolgono essi stessi il ruolo di host, fornendo servizi e gestione delle risorse ai processi delle macchine virtuali. Maggiori dettagli sugli strumenti per la virtualizzazione e sulla sua implementazione si trovano nel Capitolo 18.

1.8 Sistemi distribuiti

Per sistema distribuito si intende un insieme di elaboratori fisicamente separati e con caratteristiche spesso eterogenee, interconnessi da una rete per consentire agli utenti l'accesso alle varie risorse dei singoli sistemi. L'accesso a una risorsa condivisa aumenta la velocità di calcolo, la funzionalità, la disponibilità dei dati e il grado di affidabilità. Alcuni sistemi operativi gestiscono l'accesso alla rete come una forma di accesso ai file, demandando i dettagli dell'accesso alla rete al driver dell'interfaccia fisica con la rete. Altri sistemi, invece, permettono agli utenti di invocare funzioni specifiche della rete. Generalmente nei sistemi si ha una combinazione delle due modalità: per esempio ftp e nfs. I protocolli che danno vita a un sistema distribuito possono determinarne l'utilità e la diffusione in misura considerevole.

Una rete si può considerare, in parole semplici, come un canale di comunicazione tra due o più sistemi. I sistemi distribuiti si basano sulle reti per realizzare le proprie funzioni. Le reti differiscono per i protocolli usati, per le distanze tra i nodi e per il mezzo attraverso il quale avviene la comunicazione. Il più diffuso protocollo di comunicazione è il tcp/ip. Esso fornisce l'architettura fondamentale di Internet. La maggior parte dei sistemi operativi, inclusi tutti i sistemi general-purpose, impiega il tcp/ip. Alcuni sistemi dispongono di propri protocolli che soddisfano esigenze specifiche. Affinché un sistema operativo possa gestire un protocollo di rete è necessaria la presenza di un dispositivo d'interfaccia – un adattatore di rete, per esempio – con un driver per gestirlo, oltre al software per la gestione dei dati. Questi concetti verranno descritti nel corso del testo.

Le reti si classificano secondo le distanze tra i loro nodi: una rete locale (lan) collega nodi all'interno della stessa stanza, edificio o campus; una rete geografica (wan) si estende a gruppi di edifici, città, o al territorio di una regione o di uno stato. Una società multinazionale, per esempio, potrebbe disporre di una rete wan per connettere i propri uffici nel mondo. Queste reti possono funzionare con uno o più protocolli e il continuo sviluppo di nuove tecnologie fa sì che si definiscano nuovi tipi di reti. Le reti metropolitane (man) per esempio possono collegare gli edifici all'interno di una città; i dispositivi BlueTooth e 802.11 comunicano a brevi distanze, dell'ordine delle decine di metri, creando essenzialmente una rete personale (pan, *personal area network*) tra un telefono e gli auricolari o tra uno smartphone e un computer desktop.

I mezzi di trasmissione che si impiegano nelle reti sono altrettanto vari: fili di rame, fibre ottiche, trasmissioni via satellite, sistemi a microonde e sistemi radio; anche il collegamento dei dispositivi di calcolo ai telefoni cellulari crea una rete, così come, per creare una rete, si può usare anche la capacità di comunicazione a brevissima distanza dei dispositivi a raggi infrarossi. In breve, ogni volta che comunicano, i calcolatori usano o creano reti, che ovviamente si differenziano per prestazioni e affidabilità.

Taluni sistemi operativi hanno interpretato l'idea delle reti e dei sistemi distribuiti secondo un'ottica più vasta rispetto alla semplice fornitura della connettività di rete. Un sistema operativo di rete è dotato di caratteristiche quali la condivisione dei file attraverso la rete e di un modello di comunicazione per i processi attivi su elaboratori diversi, che possono così scambiarsi messaggi. Un computer che funziona con un sistema operativo di rete agisce in autonomia rispetto a tutti gli altri computer della rete, benché sia consci della presenza della rete e, dunque, possa interagire con gli altri computer che ne fanno parte. Un sistema operativo distribuito offre un ambiente meno autonomo: la stretta comunicazione che si instaura tra i diversi elaboratori partecipanti tende a dare l'impressione che vi sia un solo sistema operativo che controlla la rete.

Il Capitolo 19 è dedicato alle reti di calcolatori e ai sistemi distribuiti.

1.9 Strutture dati del kernel

Passiamo ora a un argomento di cruciale importanza per l'implementazione dei sistemi operativi: come i dati sono strutturati in un sistema. In questo paragrafo descriviamo brevemente diverse strutture dati fondamentali, utilizzate diffusamente dai sistemi operativi. I lettori che volessero avere maggiori dettagli su queste strutture sono invitati a consultare la bibliografia alla fine di questo capitolo.

1.9.1 Liste, stack e code

Un array è una semplice struttura dati in cui ogni elemento è direttamente accessibile. La memoria principale, per esempio, è costruita come un array. Quando un dato memorizzato è più grande di un byte possono essere allocati più byte per lo stesso dato e il dato sarà accessibile utilizzando il suo numero progressivo moltiplicato per la sua dimensione. Come procedere però in caso di dati di dimensione variabile? Come si può rimuovere un elemento quando si deve preservare la posizione relativa degli elementi rimanenti? In queste situazioni gli array devono lasciare il posto ad altre strutture dati.

Le più importanti strutture dati dopo gli array sono probabilmente le liste. Mentre ogni elemento di un array è accessibile direttamente, i dati presenti in una lista sono accessibili solo in un particolare ordine. Una lista, in altre parole, rappresenta una collezione di valori in sequenza. Il metodo più comune per implementare questo tipo di struttura è la lista concatenata, in cui gli elementi sono collegati tra di loro. Esistono diversi tipi di liste concatenate.

- In una lista semplicemente concatenata ogni elemento punta all'elemento successivo, come mostrato nella Figura 1.17.

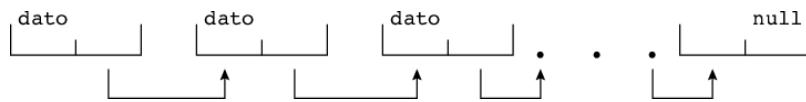


Figura 1.17 Lista semplicemente concatenata.

- In una lista doppiamente concatenata ogni elemento contiene riferimenti sia al suo successore sia al suo predecessore, come mostrato nella Figura 1.18.

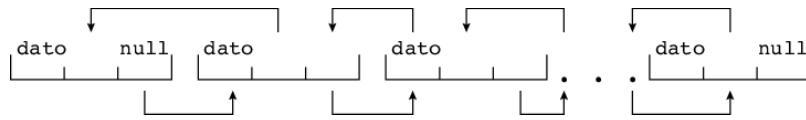


Figura 1.18 Lista doppiamente concatenata.

- In una lista circolare l'ultimo elemento punta al primo elemento piuttosto che a un valore null, come è mostrato nella Figura 1.19.

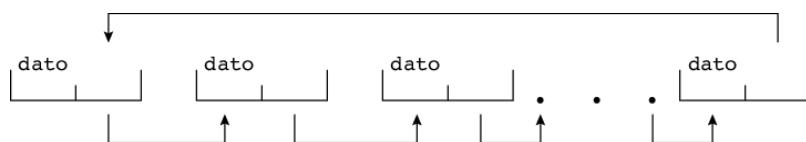


Figura 1.19 Lista circolare.

Le liste concatenate possono contenere dati di diversa dimensione e permettono di inserire e rimuovere elementi in maniera semplice. Un potenziale svantaggio nell'utilizzo di liste è che il tempo per prelevare un fissato elemento in una lista di dimensione n è lineare – $O(n)$ –, perché potrebbe essere necessario, nel caso peggiore, attraversarne tutti gli elementi. Le liste sono talvolta utilizzate direttamente dagli algoritmi del kernel, ma più frequentemente le si utilizza per costruire strutture dati più potenti, come pile e code.

Uno stack (chiamato anche pila) è una struttura dati dotata di un ordine sequenziale che utilizza una politica di tipo lifo (*last in first out*) per l'inserimento e la cancellazione degli elementi: l'ultimo elemento inserito è il primo a essere rimosso. Le operazioni di inserimento e cancellazione su uno stack si chiamano *push* e *pop*, rispettivamente. Un sistema operativo utilizza spesso uno stack per gestire una chiamata di funzione. In questo caso, al momento della chiamata i parametri, le variabili locali e l'indirizzo di ritorno vengono inseriti, mediante operazioni di push, nello stack. Al ritorno dalla funzione gli elementi inseriti nello stack vengono rimossi mediante operazioni di pop.

Una coda, a differenza dello stack, è una struttura dati ordinata sequenzialmente che adotta una politica di accesso di tipo fifo (*first in first out*): gli elementi vengono rimossi da una coda nell'ordine in cui sono stati inseriti. Troviamo diversi esempi di code nella vita quotidiana, tra cui le persone in fila alla cassa di un negozio o le auto al semaforo in attesa del verde. Le code sono anche piuttosto comuni nei sistemi operativi, per esempio nel caso dei documenti inviati a una stampante che vengono di solito processati nell'ordine in cui sono stati ricevuti. Come vedremo nel Capitolo 5 i task in attesa di essere eseguiti su una cpu sono spesso organizzati in code.

1.9.2 Alberi

Un albero è una struttura dati utilizzabile per rappresentare i dati in maniera gerarchica. Gli elementi di un albero sono strutturati secondo una relazione padre-figlio. In un generico albero un padre può avere un numero illimitato di figli, mentre in un albero binario un padre ha al massimo due figli, chiamati figlio sinistro e figlio destro. Un albero di ricerca binario soddisfa un requisito aggiuntivo: i figli sono ordinati in modo che *figlio_sinistro <= figlio_destro*. La Figura 1.20 rappresenta un esempio di albero di ricerca binario. Quando si cerca un elemento in un albero binario di ricerca le prestazioni nel caso peggiore sono $O(n)$ (lasciamo al lettore di capire quando si verifica il caso peggiore). Per rimediare a questa situazione possiamo utilizzare un algoritmo per la creazione di un albero binario di ricerca bilanciato. Un tale albero con n elementi ha al più $\log n$ livelli, assicurando così prestazioni di $O(\log n)$ nel caso peggiore. Come vedremo nel Paragrafo 5.7.1 Linux utilizza un albero binario di ricerca bilanciato nel suo algoritmo di scheduling della cpu.

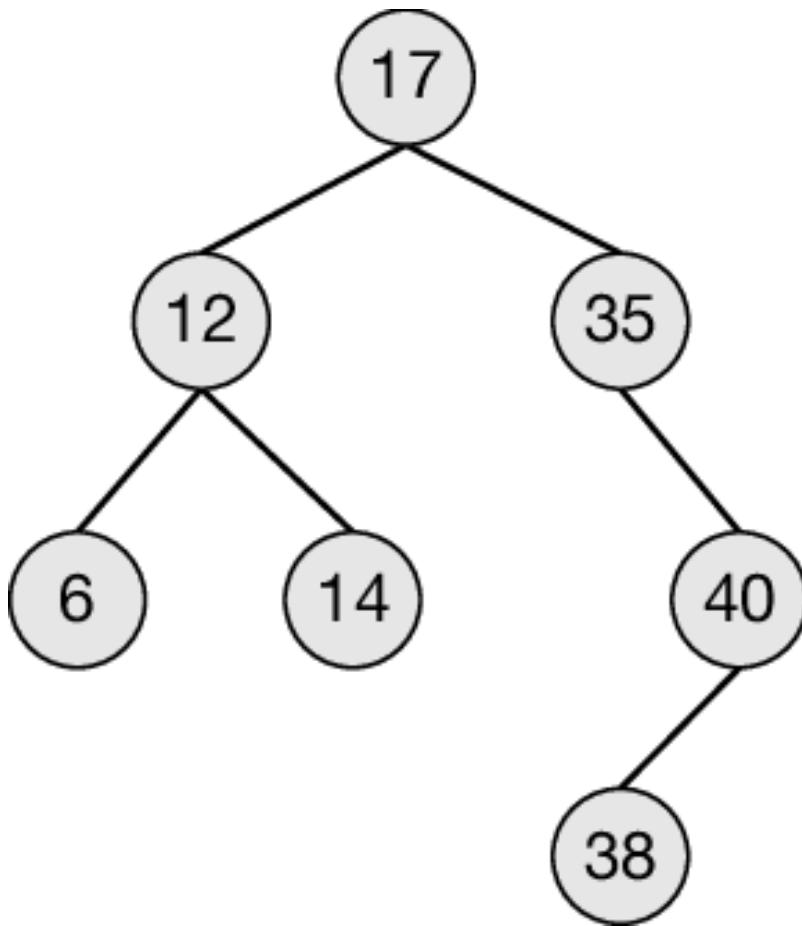


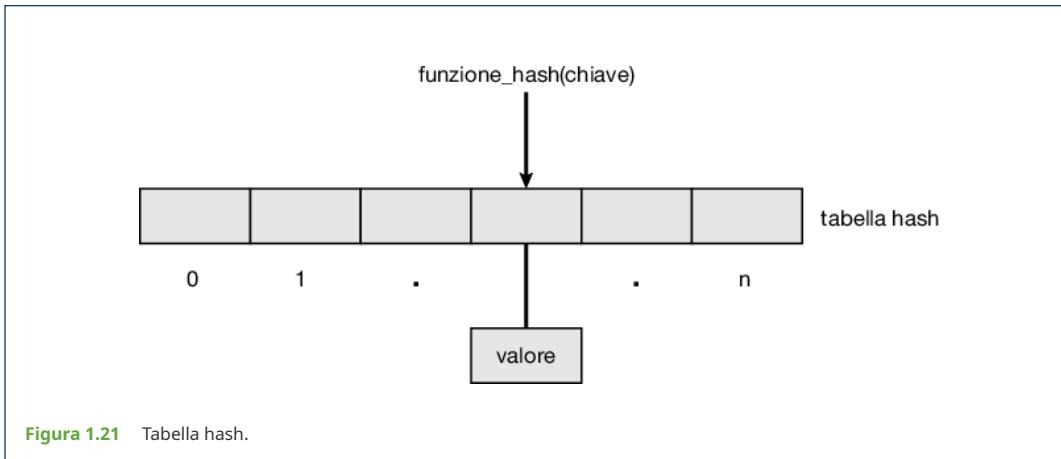
Figura 1.20 Albero binario di ricerca.

1.9.3 Funzioni e mappe hash

Una funzione hash riceve dati in input, realizza operazioni numeriche sui dati e restituisce un valore numerico. Questo valore può essere utilizzato come indice in una tabella (di solito un array) per recuperare velocemente il dato. Mentre la ricerca di un elemento in una lista di dimensione n può richiedere fino a $O(n)$ confronti nel caso peggiore, utilizzando una funzione hash per il recupero di un dato si può arrivare anche a $O(1)$ nel caso peggiore, a seconda dei dettagli di implementazione. Grazie a queste prestazioni le funzioni hash sono molto utilizzate dai sistemi operativi.

Un potenziale problema delle funzioni hash è il fatto che su input differenti si può produrre lo stesso valore di output, ossia i due input possono corrispondere alla stessa locazione nella tabella. Questo problema di collisione può essere risolto inserendo nella locazione una lista concatenata contenente tutti gli elementi con lo stesso valore hash. Ovviamente maggiore è il numero di collisioni, meno efficiente è la funzione hash.

Una funzione hash può essere utilizzata per creare una tabella hash (o mappa hash) che associa (o mappa) coppie [chiave:valore]. Possiamo per esempio mappare la chiave *operativo* al valore *sistema*. Una volta stabilita la mappa, è possibile applicare la funzione hash alla chiave per ottenere dalla tabella hash il valore (Figura 1.21). Si supponga, per esempio, che il valore di una username sia mappato in una password. Per l'autenticazione si procede come segue: l'utente inserisce username e password; viene applicata la funzione hash alla username, in modo da recuperare la password dalla tabella; la password viene confrontata con quella inserita dall'utente per l'autenticazione.



LE STRUTTURE DATI DEL KERNEL DI LINUX

Le strutture dati del kernel di Linux sono disponibili nel codice sorgente. Il file `<linux/list.h>` fornisce i dettagli delle liste concatenate usate dal kernel. In Linux una coda è conosciuta come `kfifo` e la sua implementazione si trova in `kfifo.c`, nella directory `kernel` del codice sorgente. Linux mette a disposizione anche un'implementazione di un albero binario di ricerca bilanciato che utilizza gli **R-B alberi** (*alberi Red-Black*). I dettagli si trovano nel file `<linux/rbtree.h>`.

1.9.4 Bitmap

Una bitmap è una stringa di n caratteri binari utilizzabile per rappresentare lo stato di n elementi. Per esempio, si supponga di avere diverse risorse la cui disponibilità è indicata dal valore di un bit: 0 significa che la risorsa è disponibile e 1 significa che la risorsa è occupata (o viceversa). Il valore della i -esima posizione nella bitmap è associato alla i -esima risorsa. Si consideri la bitmap 001011101. I suoi bit indicano che le risorse 2, 4, 5, 6 e 8 non sono disponibili, mentre le risorse 0, 1, 3 e 7 lo sono.

La potenzialità delle bitmap risulta evidente quando consideriamo la loro efficienza in termini di spazio utilizzato. Se infatti utilizzassimo un valore di 8 bit al posto di un singolo bit la struttura dati risultante sarebbe 8 volte più grande. Per questa ragione le bitmap sono spesso utilizzate quando si ha la necessità di rappresentare la disponibilità di un gran numero di risorse. I dischi ci offrono un ottimo esempio. Un disco di medie dimensioni può essere diviso in diverse migliaia di unità, chiamate blocchi. Una bitmap può essere utilizzata per indicare la disponibilità di ognuno dei blocchi.

In sintesi, le strutture dati pervadono l'implementazione dei sistemi operativi. Rivedremo quindi le strutture dati appena descritte, insieme ad altre strutture, nel corso del testo, quando studieremo gli algoritmi del kernel e le loro implementazioni.

1.10 Ambienti d'elaborazione

Abbiamo finora descritto brevemente diversi aspetti dei sistemi elaborativi e dei sistemi operativi che li gestiscono. Passiamo ora a discutere come i sistemi operativi siano utilizzati in ambienti d'elaborazione diversi.

1.10.1 Elaborazione tradizionale

Con l'evoluzione delle tecniche d'elaborazione i confini tra molti ambienti d'elaborazione tradizionale diventano sempre più sfumati. Si consideri per esempio un tipico ambiente d'ufficio: solo pochi anni fa consisteva di pc connessi in rete, con server che fornivano servizi di accesso ai file e di stampa. L'accesso remoto era difficoltoso e la portabilità si otteneva grazie ai laptop.

Oggi le tecnologie del Web e la crescita della velocità delle wan stanno estendendo i confini dell'elaborazione tradizionale. Le aziende realizzano portali che permettono l'accesso tramite il Web ai propri server interni. I network computer (detti anche thin client) sono essenzialmente terminali adatti all'elaborazione basata sul Web e vengono utilizzati al posto delle tradizionali workstation quando è richiesta una sicurezza maggiore o una manutenzione più semplice. I dispositivi mobili possono sincronizzarsi con i pc per consentire un uso estremamente portatile delle informazioni aziendali e permettono la connessione a reti wireless e a reti cellulari per accedere al portale web dell'azienda (e alle tantissime altre risorse del Web).

A casa, la maggior parte degli utenti aveva un solo calcolatore con una lenta connessione via modem all'ufficio, alla rete Internet, o a entrambi. Le connessioni di rete veloci, un tempo possibili a costi molto alti, sono ora disponibili in molti luoghi a prezzi abbastanza contenuti e permettono l'accesso a maggiori quantità di dati. Queste connessioni veloci consentono ai calcolatori di casa di trasformarsi in server web e di formare reti con stampanti, pc client e server. Alcuni ambienti d'elaborazione domestici sono dotati anche di firewall (*barriere anti-intrusione*) che proteggono dagli attacchi informatici esterni. I firewall limitano la comunicazione tra i dispositivi su una rete.

Nella seconda metà del '900 le risorse elettroniche di calcolo erano relativamente scarse (e prima ancora, non esistevano!). C'è stato un periodo di tempo in cui i sistemi erano o a lotti (batch) oppure interattivi. I sistemi a lotti elaboravano i processi all'ingrosso, per così dire, con un input predeterminato (da file o da altre fonti). I sistemi interattivi aspettavano di ricevere i dati in ingresso dagli utenti. Per ottenere la massima resa dall'elaboratore, diversi utenti utilizzavano questi sistemi in time-sharing. I sistemi time-sharing impiegavano un timer e algoritmi di scheduling per assegnare rapidamente i processi alla cpu, attribuendo a ogni utente una parte delle risorse.

Attualmente, i tradizionali sistemi time-sharing sono rari. Le stesse strategie di scheduling sono tuttora usate da computer desktop e portatili, dai server e anche dai dispositivi mobili, ma spesso tutti i processi fanno capo allo stesso utente (o a un utente singolo e al sistema operativo). I processi dell'utente, e i processi del sistema che forniscono servizi all'utente, sono gestiti in modo da ricevere ciascuno, frequentemente, una fetta del tempo a disposizione. Ciò si può notare, per esempio, osservando le finestre esistenti durante il lavoro di un utente al calcolatore, e notando che ciascuna di esse può eseguire nel contempo una diversa operazione. Anche un browser web può essere formato da più processi, uno per ogni pagina che si sta visitando, con la condivisione di tempo applicata a ogni processo.

1.10.2 Mobile computing

Con il termine mobile computing si fa riferimento all'elaborazione su smartphone e tablet. Questi dispositivi hanno in comune alcune caratteristiche fisiche che li contraddistinguono: sono portatili e leggeri. In origine, a confronto con desktop e computer portatili, i sistemi mobili dovevano fare rinunce nelle dimensioni dello schermo, nella capacità di memoria e nelle funzionalità complesse per poter offrire servizi come l'accesso alla posta elettronica e la navigazione sul Web su un dispositivo portatile. Negli ultimi anni le funzionalità disponibili sui dispositivi mobili si sono così arricchite da rendere indistinguibili un computer portatile e un tablet. In alcuni casi le funzioni di un moderno dispositivo mobile non sono disponibili, o non sono facilmente realizzabili, su computer desktop e portatili.

I sistemi mobili sono oggi utilizzati, oltre che per la posta elettronica e l'accesso al Web, anche per riprodurre musica e video, per leggere libri, per fotografare e registrare filmati in alta definizione. L'ampia gamma di applicazioni disponibili per questi dispositivi è dunque soggetta a un continuo sviluppo. Diversi sviluppatori progettano applicazioni in grado di sfruttare le caratteristiche peculiari dei dispositivi mobili, come il gps, l'accelerometro e il giroscopio. Il chip gps permette al dispositivo mobile di utilizzare i satelliti per determinare la sua posizione in maniera accurata. Questo strumento è particolarmente utile per sviluppare applicazioni che offrono servizi di navigazione, per esempio per dire all'utente quali strade percorrere e dove andare per raggiungere determinati servizi, come un ristorante. Un accelerometro permette a un dispositivo mobile di determinare il suo orientamento rispetto al piano e di rilevare alcuni movimenti. In molti giochi che utilizzano gli accelerometri i giocatori interagiscono con il sistema senza mouse né tastiera, ma attraverso movimenti del dispositivo, per esempio ruotandolo o scuotendolo. Un utilizzo pratico degli strumenti appena citati si trova anche nelle applicazioni di realtà aumentata, che sovrappongono determinate informazioni a un display dell'ambiente circostante. È difficile immaginare simili applicazioni in esecuzione su computer tradizionali.

Per offrire accesso ai servizi on-line i dispositivi mobili utilizzano di solito lo standard wireless 802.11 o le reti cellulari. La capacità di memoria e la velocità del processore dei dispositivi mobili sono limitate rispetto a quelle dei pc. Uno smartphone o un tablet possono avere 256 gb di memoria secondaria, mentre non è insolito trovare anche 8 tb di spazio su computer desktop. Analogamente, a causa dell'importanza di mantenere bassi i consumi, i dispositivi mobili utilizzano spesso processori più piccoli, più lenti e dotati di meno core rispetto ai processori desktop e portatili.

Il mercato mobile è attualmente dominato da due sistemi operativi: Apple ios e Google Android. ios è progettato per essere eseguito sui dispositivi iPhone e iPad di Apple, mentre Android viene eseguito su dispositivi di diversi produttori. Esamineremo questi due sistemi operativi più nel dettaglio nel Capitolo 2.

1.10.3 Elaborazione client-server

Un'architettura di rete contemporanea realizza un sistema in cui alcuni server soddisfano le richieste dei sistemi client. Questa particolare variante dei sistemi distribuiti, che prende il nome di sistema client-server, ha la struttura generale rappresentata nella Figura 1.22.

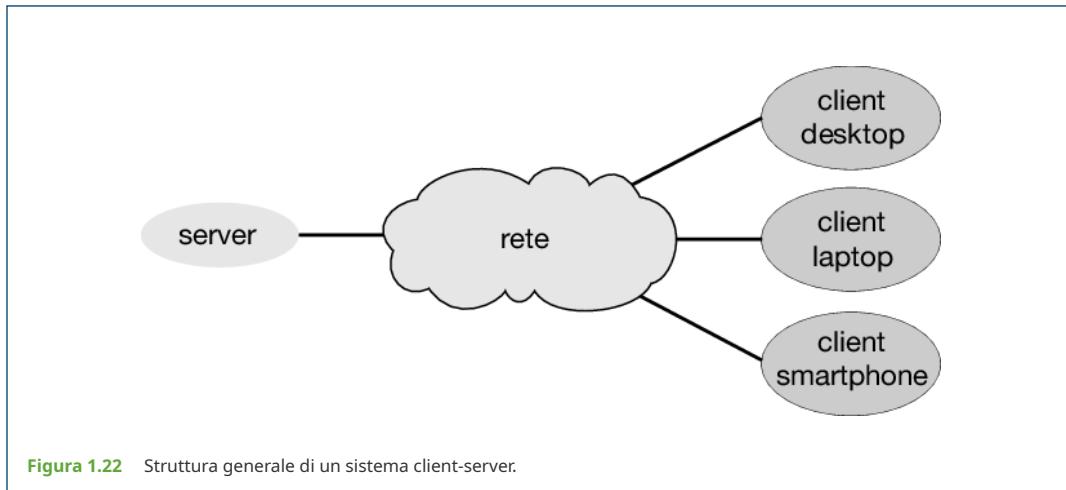


Figura 1.22 Struttura generale di un sistema client-server.

Schematicamente possiamo suddividere i sistemi server come server elaborativi e file server.

- I server elaborativi forniscono un'interfaccia a cui i client possono inviare una richiesta (per esempio, la lettura di alcuni dati); in risposta, il server esegue l'azione richiesta e restituisce i risultati al client. Un server che ospiti una base di dati a cui i client possono attingere costituisce un esempio di tale sistema.
- I file server offrono un'interfaccia di file system che consente al client creazione, aggiornamento, lettura e cancellazione dei file. Un esempio di questo sistema è dato da un server web che trasferisce i file richiestigli dai browser dei client. Il contenuto effettivo dei file può variare molto, andando dalle tradizionali pagine web fino a contenuti multimediali avanzati come video ad alta definizione.

1.10.4 Elaborazione peer-to-peer

Un'altra architettura di sistema distribuito è il modello di sistema “da pari a pari”, o peer-to-peer (P2P). In tale modello cade la distinzione tra client e server; infatti, tutti i nodi all’interno del sistema sono su un piano di parità, e ciascuno può fungere ora da client, ora da server, a seconda che stia richiedendo o fornendo un servizio. Questi sistemi offrono un vantaggio rispetto ai sistemi client-server tradizionali: infatti, in un sistema client-server un server può diventare un collo di bottiglia, mentre in un sistema peer-to-peer, uno stesso servizio può essere fornito da uno qualunque dei vari nodi distribuiti nella rete.

Per entrare a far parte di un sistema peer-to-peer, un nodo deve in primo luogo unirsi alla rete degli altri sistemi. Una volta entrato a far parte della rete, esso può iniziare a fornire i servizi agli altri nodi che risiedono nella rete e, a sua volta, ottenerli dagli altri nodi. Vi sono due modalità generali per stabilire quali servizi siano disponibili.

- Al momento di unirsi a una rete, un nodo iscrive il proprio servizio in un registro centralizzato di consultazione della rete. Quando un nodo vuole ottenere un servizio, esso contatta in via preliminare il registro centralizzato, per verificare quali nodi lo forniscono. Il resto della comunicazione ha luogo tra il client e il fornitore del servizio.
- Uno schema alternativo non utilizza alcun servizio di consultazione centralizzato. Un nodo che operi come client deve innanzitutto accertare quale nodo fornisca il servizio desiderato, inoltrando la propria richiesta di servizio a tutti gli altri nodi della rete. I nodi che possono fornire tale servizio rispondono al nodo da cui è partita la richiesta. Questa procedura richiede un *protocollo di scoperta*, che deve permettere ai nodi di scoprire i servizi forniti dagli altri nodi della rete. La Figura 1.23 illustra un tale scenario.

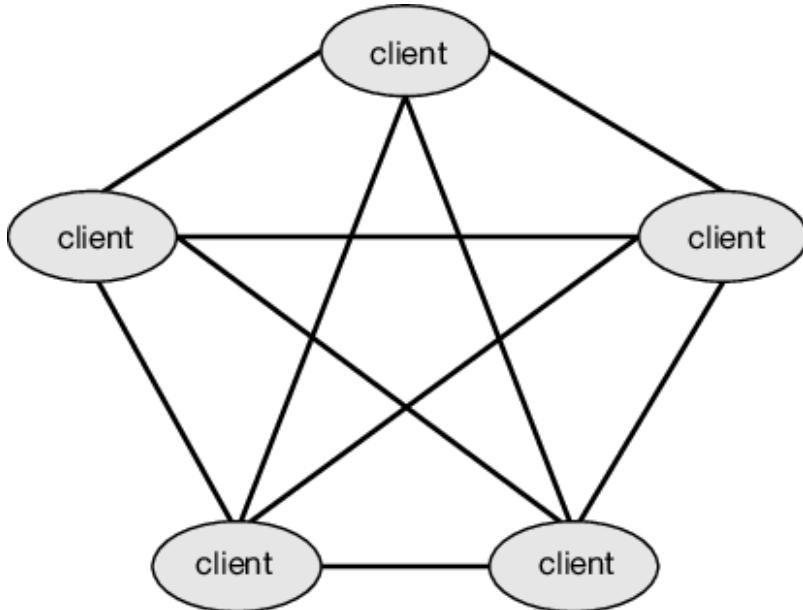


Figura 1.23 Sistema peer-to-peer senza servizi centralizzati.

Le reti peer-to-peer hanno avuto larga diffusione dalla fine degli anni '90 grazie a diversi servizi di condivisione dei file, quali Napster e Gnutella, che permettevano agli utenti lo scambio reciproco di file. Il sistema Napster utilizzava una modalità operativa simile al primo tipo sopra descritto: un server centrale conservava il registro di tutti i file prelevabili dai nodi della rete Napster, e lo scambio effettivo dei file avveniva tra i nodi stessi. Il sistema Gnutella adottava una tecnica simile al secondo tipo descritto: un client faceva pervenire le richieste di file agli altri nodi del sistema e i nodi che potevano soddisfare la richiesta rispondevano direttamente al client.

Il futuro dello scambio di file resta incerto, perché le reti peer-to-peer possono essere utilizzate per lo scambio di materiale protetto da diritti d'autore (la musica, per esempio) in forma anonima, contravvenendo a leggi sulla distribuzione di materiale protetto. Napster, per esempio, ha avuto problemi legali per violazione di copyright ed è stato chiuso nell'anno 2001.

Skype è un altro esempio di applicazione peer-to-peer, in questo caso per permettere agli utenti di effettuare chiamate voce e videochiamate e di inviare messaggi attraverso Internet, sfruttando una tecnologia nota come VoIP (Voice over IP). Skype utilizza un approccio peer-to-peer ibrido: è dotato di un server di autenticazione centralizzato, ma dispone anche di nodi decentralizzati e permette a due nodi di comunicare direttamente.

1.10.5 Cloud computing

Il cloud computing è una tecnica che permette di fornire capacità elaborativa, storage e persino applicazioni come servizi di rete. In un certo senso si tratta di un'estensione logica della virtualizzazione, perché utilizza la virtualizzazione come strumento base per offrire le sue funzionalità. Per esempio, ec2 (*elastic compute cloud*) di Amazon dispone di migliaia di server, milioni di macchine virtuali e petabyte di spazio dati a disposizione di ogni utente connesso a Internet. Gli utenti pagano una tariffa mensile che dipende dalla quantità di risorse che utilizzano.

Esistono diverse tipologie di cloud computing, tra cui le seguenti.

- Cloud pubblico: un cloud disponibile attraverso Internet a chiunque si abboni al servizio.
- Cloud privato: un cloud gestito da un'azienda per l'utilizzo al proprio interno.
- Cloud ibrido: un cloud che comprende componenti pubbliche e private.
- SaaS (software as a service): una o più applicazioni (per esempio un word processor o un foglio di calcolo) fruibili via Internet.
- PaaS (platform as a service): un ambiente software predisposto per usi applicativi via Internet (per esempio un database server).
- IaaS (Infrastructure as a Service): server o storage disponibili attraverso Internet (per esempio storage disponibile per eseguire copie di backup dei dati di produzione).

La divisione tra queste tipologie di cloud non è netta: una piattaforma cloud può offrire una combinazione di tipologie diverse di servizi. Per esempio, una società può offrire sia SaaS sia IaaS come servizio disponibile pubblicamente.

Ci sono certamente sistemi operativi tradizionali dentro molti tipi di infrastrutture cloud. Al di sopra di questi sistemi ci sono i vmm che gestiscono le macchine virtuali su cui vengono eseguiti i processi utente. A un livello più alto gli stessi vmm sono controllati da strumenti per la gestione del cloud, come vmware vCloud Director e l'open-source Eucalyptus. Questi sistemi gestiscono le risorse

all'interno di un dato cloud e forniscono interfacce verso i componenti cloud, offrendo così buoni argomenti per considerarli un nuovo tipo di sistema operativo.

La Figura 1.24 mostra un cloud pubblico che offre il servizio IaaS. Si noti che sia i servizi cloud sia l'interfaccia utente sono protetti da firewall.

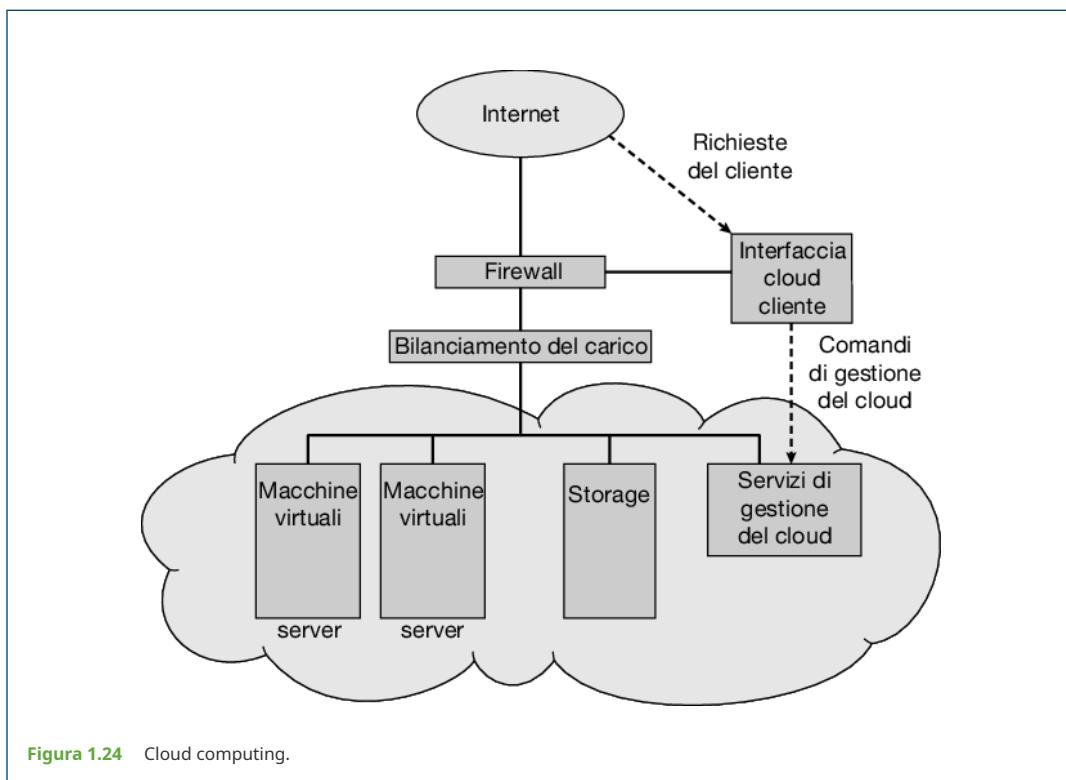


Figura 1.24 Cloud computing.

1.10.6 Sistemi embedded real-time

In termini quantitativi, i computer facenti parte di altri sistemi (*embedded computer*) attualmente costituiscono la tipologia predominante di elaboratore. Questi dispositivi si ritrovano dappertutto, dai motori delle auto ai robot industriali, dai lettori di dvd ai forni a microonde. Di solito, hanno compiti molto precisi: i sistemi su cui sono installati sono spesso rudimentali, per cui offrono funzionalità limitate. Essi presentano un'interfaccia utente scarsamente sviluppata, se non assente, dato che sono spesso concepiti per la sorveglianza e la gestione di dispositivi meccanici, quali i motori delle automobili e i bracci dei robot.

Ciò che contraddistingue i sistemi embedded è la loro grande variabilità. Talvolta possono essere elaboratori general-purpose con sistemi operativi standard – come Linux – che sfruttano applicazioni create appositamente per implementare una funzionalità. Altri sono dispositivi hardware che ospitano un sistema operativo special purpose, che consente di ottenere proprio la funzionalità desiderata. Inoltre, esistono dispositivi hardware che hanno al loro interno circuiti integrati per applicazioni specifiche (asic), capaci di svolgere il loro lavoro senza un sistema operativo.

La diffusione dei sistemi embedded è in continua espansione. Indubbiamente, sono destinate a crescere anche le potenzialità di questi congegni, sia come unità indipendenti sia in qualità di membri delle reti e di Internet. È già oggi possibile l'automatizzazione di intere abitazioni, cosicché un computer centrale – sia che si tratti di un computer general-purpose o di un sistema embedded – possa essere in grado di controllare il riscaldamento, l'illuminazione o i sistemi di allarme. Attraverso la rete si può comandare a distanza il riscaldamento della propria casa prima di rientrarvi. Un giorno, il frigorifero potrà forse prendere autonomamente l'iniziativa di chiamare il negozio di alimentari per il rifornimento del latte.

I sistemi embedded funzionano quasi sempre con sistemi operativi real-time. Essi si utilizzano quando siano stati imposti rigidi vincoli di tempo alle funzioni del processore o al flusso dei dati: per questa ragione sono spesso adoperati come dispositivi di controllo per applicazioni dedicate. I sensori trasmettono i dati al computer, che deve analizzarli e, a volte, prendere le misure adatte per il controllo del sistema. I sistemi adibiti al controllo di esperimenti scientifici, i sistemi di controllo industriale e taluni sistemi per la visualizzazione, sono sistemi real-time. Alcuni motori a iniezione di benzina, i sistemi d'allarme delle case e i sistemi d'arma sono, anch'essi, real-time.

Questo tipo di sistema ha vincoli di tempo fissi e definiti con precisione. L'elaborazione *dove* avvenire entro i limiti prestabiliti: in caso contrario, il sistema andrà in crisi. Per esempio, non serve a nulla che il braccio di un robot riceva l'ordine di fermarsi solo *dopo*

essersi scontrato con l'automobile che era impegnato a costruire. Un sistema real-time funziona correttamente solo se esso genera il risultato corretto entro precise scadenze.

Il Capitolo 5 esamina la tipologia di scheduling da adottare affinché un sistema operativo possa eseguire applicazioni real-time. Il Capitolo 20 (reperibile sulla piattaforma MyLab) è dedicato ai componenti real-time di Linux.

1.11 Sistemi operativi liberi e open-source

Lo studio dei sistemi operativi è stato facilitato dalla disponibilità di un vasto numero di programmi liberi e open-source. Sia i sistemi operativi liberi sia i sistemi operativi open-source sono disponibili in formato sorgente anziché come codice binario compilato. Si noti, tuttavia, che software libero e software open-source sono due idee diverse sostenute da diversi gruppi di persone (si veda <http://gnu.org/philosophy/open-source-misses-the-point.html> per una discussione sul tema). Il software libero (chiamato anche *free/libre software*) non solo rende il codice sorgente disponibile, ma è anche dotato di una licenza che consente l'uso, la ridistribuzione e la modifica senza costi. Il software open-source non offre necessariamente tale licenza. Pertanto, sebbene tutto il software libero sia open-source, alcuni software open-source non sono liberi. gnu/Linux è il sistema operativo open-source più famoso, con alcune distribuzioni libere e altre solo open-source (<http://www.gnu.org/distros/>). Microsoft Windows è un ben noto esempio dell'approccio opposto (*closed-source*). Windows è un software proprietario, Microsoft lo possiede, ne limita l'uso e ne protegge attentamente il codice sorgente. Il sistema operativo macos di Apple adotta un approccio ibrido: contiene un kernel open-source chiamato Darwin, ma include anche componenti chiuse e proprietarie.

Avere a disposizione il codice sorgente permette al programmatore di produrre il codice binario, eseguibile da un sistema. Il processo inverso, chiamato processo di reverse-engineering, che permette di ricavare il codice sorgente partendo dal binario, è molto più oneroso; molti elementi utili, per esempio i commenti, non possono essere ripristinati. Apprendere il funzionamento dei sistemi operativi esaminandone il codice sorgente originale può essere molto utile. Avendo a disposizione il codice sorgente, uno studente può modificare il sistema operativo per poi compilare ed eseguire il codice, verificando così i cambiamenti che vi ha apportato. Procedere in questo modo è sicuramente di notevole aiuto per l'apprendimento. Questo libro include degli esercizi che richiedono la modifica del codice sorgente di un sistema operativo. Alcuni algoritmi, inoltre, sono descritti ad alto livello, per essere sicuri di coprire tutti gli argomenti importanti che riguardano i sistemi operativi. Nel libro si possono inoltre trovare riferimenti a esempi di codice open-source, per eventuali approfondimenti.

I vantaggi dei sistemi operativi open-source sono molti. Tra questi vi è la presenza di una comunità di programmatore interessati (e spesso non retribuiti) che contribuisce allo sviluppo aiutando a verificare la presenza di eventuali errori nel codice, ad analizzarlo, a dare assistenza e a suggerire dei cambiamenti. Si può sostenere che i programmi open-source siano più sicuri di quelli a sorgente chiuso, perché molti più occhi sono puntati sul codice. Certo, anche i codici open-source hanno dei bachi ma, argomentano i fautori dell'open-source, questi bachi vengono scoperti ed eliminati molto più velocemente proprio grazie al gran numero di utilizzatori. Le società che traggono profitto dalla vendita dei loro programmi sono spesso riluttanti a rendere accessibili i loro sorgenti, anche se le aziende che stanno procedendo in questa direzione, come Red Hat e molte altre, dimostrano di trarne benefici commerciali, anziché soffrirne. Per tali società il profitto deriva, per esempio, da contratti di assistenza e dalla vendita di hardware sul quale il software funziona.

1.11.1 Storia

Ai primordi dell'informatica moderna (ovvero negli anni '50 del secolo scorso) gran parte del software era disponibile in formato open-source. Gli hacker di allora (gli appassionati dei computer) al Tech Model Railroad Club del mit lasciavano i loro programmi nei cassetti affinché altri potessero lavorarci. Gruppi di utenti "casalinghi" ("Homebrew") scambiavano il codice durante i loro incontri. Qualche tempo dopo, gruppi di utenti legati a specifiche società, come la Digital Equipment Corporation, accettarono contributi di programmi in codice sorgente e li raccolsero su nastri per poi distribuirli ai membri interessati. Nel 1970 i sistemi operativi della Digital venivano distribuiti come codice sorgente, senza alcuna restrizione né avvisi sul copyright.

Successivamente le società informatiche cercarono di limitare l'utilizzo del loro software a computer autorizzati e clienti paganti. Riuscirono a raggiungere questo obiettivo rendendo disponibili solo i file in binario compilati a partire dal codice sorgente, ma non il codice stesso. Esse protessero così allo stesso tempo il proprio codice e le proprie idee dai concorrenti. Anche se il gruppo di utenti Homebrew negli anni '70 si scambiava codice sorgente durante i meeting, i sistemi operativi (come il cpm) per pc amatoriali erano proprietari. Fino agli anni '80 il software proprietario era il più comune.

1.11.2 Sistemi operativi liberi

Per contrastare la decisione di limitare l'uso e la ridistribuzione del software, nel 1984 Richard Stallman iniziò a sviluppare un sistema operativo libero compatibile con unix chiamato gnu (acronimo ricorsivo che sta per "gnu's Not Unix!"). Per Stallman, il termine "libero" si riferisce alla libertà di utilizzo, non al prezzo. Il movimento per il software libero non si oppone al commercio delle copie, ma sostiene che gli utenti abbiano diritto a quattro libertà: (1) eseguire liberamente il programma, (2) studiare e modificare il codice sorgente, e regalarne o vendere delle copie (3) con o (4) senza modifiche. Nel 1985 Stallman pubblicò il Manifesto gnu, in cui si sosteneva che tutto il software avrebbe dovuto essere libero, e fondò la Free Software Foundation (fsf) con l'obiettivo di incoraggiare l'uso e lo sviluppo del software libero.

La fsf utilizza i copyright sui suoi programmi per implementare "copyleft", una forma di licenza inventata da Stallman. Rilasciare diritti di tipo Copyleft a un'opera offre a chiunque possieda una copia del lavoro le quattro libertà essenziali che rendono il lavoro libero, alla condizione che la ridistribuzione debba preservare queste libertà. La gnu General Public License (gpl) è una licenza diffusa che viene utilizzata per rilasciare il software libero. In sostanza, la gpl richiede che il codice sorgente sia distribuito con qualsiasi binario e che tutte le copie (incluso le versioni modificate) vengano rilasciate con la stessa licenza gpl. Anche la licenza Creative Commons "Attribution Sharealike" è una licenza copyleft; si tratta infatti di un altro modo di affermare l'idea del copyleft.

1.11.3 gnu/Linux

Consideriamo gnu/Linux come esempio di sistema operativo libero e open-source. Nel 1991 il sistema operativo gnu era quasi completo. Il progetto gnu produsse molti strumenti, inclusi compilatori, editor, utilità, librerie e giochi, ma non rilasciò mai un kernel. Nel 1991 uno studente finlandese, Linus Torvalds, rilasciò un kernel rudimentale simile a unix utilizzando compilatori e strumenti di

gnu e invitò gli interessati a contribuire allo sviluppo. Con l'avvento di Internet, chiunque era interessato al progetto poteva scaricare il codice sorgente, modificarlo e sottoporre i cambiamenti a Torvalds. Il rilascio settimanale di aggiornamenti permise a questo sistema operativo, il cosiddetto Linux, di crescere rapidamente, avvalendosi delle migliorie apportate da migliaia di programmatore. Nel 1991 Linux non era un software libero, in quanto la sua licenza consentiva solo la ridistribuzione non commerciale. Nel 1992, tuttavia, Torvalds rilasciò nuovamente Linux sotto licenza gpl, rendendolo un software libero (e anche, per usare un termine coniato negli anni successivi, open-source).

Il sistema operativo gnu/Linux che ne è risultato (il termine corretto per il solo kernel è Linux, ma il sistema operativo completo, compresi gli strumenti gnu, va chiamato gnu/Linux) ha creato centinaia di singole distribuzioni, ossia versioni personalizzate, del sistema. Le principali distribuzioni includono Red Hat, suse, Fedora, Debian, Slackware e Ubuntu. Le distribuzioni differiscono nelle funzionalità, nelle applicazioni installate, nel supporto hardware, nell'interfaccia e negli obiettivi. Per esempio, Red Hat Enterprise Linux è indirizzato al grande uso commerciale. pclinuxos è un Livecd, un sistema operativo che può essere avviato ed eseguito da un cd-rom senza essere installato sul disco fisso. Una variante di pclinuxos, "pclinuxos Supergamer dvd", è un Livedvd che include driver per la grafica e giochi. Un giocatore può farlo funzionare su qualsiasi sistema compatibile semplicemente avviandolo dal dvd; al termine del gioco il riavvio del sistema ripristina il sistema operativo originario.

I seguenti passi permettono di eseguire Linux, gratuitamente, su un sistema Windows.

1. Scaricate il software di virtualizzazione gratuito "Virtualbox vmm tool" all'indirizzo
2. <https://www.virtualbox.org/>
3. e installatelo sul vostro sistema.
4. Scegliete di installare un sistema operativo da zero a partire, per esempio, da un cd di installazione, oppure scegliete da un sito come
5. <http://virtualboxes.org/images/>
6. un'immagine del sistema operativo preconfigurata che può essere installata ed eseguita più velocemente. Queste immagini hanno preinstallati sistemi operativi e applicazioni e includono diverse varianti di gnu/Linux.
7. Avviate la macchina virtuale all'interno di Virtualbox.

Un'alternativa a Virtualbox è il programma libero Qemu (<http://wiki.qemu.org/Download/>) che permette, attraverso il comando `qemu-img`, di convertire le immagini Virtualbox in immagini Qemu per importarle facilmente.

In allegato a questo testo viene fornita un'immagine con la distribuzione Ubuntu di gnu/Linux. L'immagine contiene il codice sorgente gnu/Linux e alcuni strumenti per lo sviluppo software. Nel testo sono contenuti esempi che riguardano questa immagine e, nel Capitolo 20 (reperibile sulla piattaforma MyLab), un caso di studio dettagliato.

1.11.4 unix bsd

Rispetto a Linux, unix bsd ha una storia più lunga e complicata. La sua creazione, derivata dallo unix di at&t, risale al 1978 e le sue prime versioni vennero distribuite dall'Università della California a Berkeley (ucb) in codice sorgente e in formato binario, ma non erano open-source, perché era necessaria una licenza della at&t. Lo sviluppo di unix bsd venne rallentato nei successivi anni da una querela della at&t, ma alla fine una versione completa e open-source del sistema, la 4.4bsd-lite, venne rilasciata nel 1994.

Esattamente come nel caso di Linux, ci sono diverse distribuzioni di unix bsd, tra le quali Freebsd, Netbsd, Openbsd e Dragonflybsd. Per esaminare nel dettaglio il codice sorgente di Freebsd è sufficiente scaricare l'immagine della versione desiderata e caricarla in Virtualbox, come descritto in precedenza per Linux. Il codice sorgente è allegato alla distribuzione e lo si può trovare in `/usr/src`. Il codice sorgente del kernel si trova in `/usr/src/sys`. Per esaminare, per esempio, il codice che implementa la memoria virtuale nel kernel di Freebsd, è sufficiente andare in `/usr/src/sys/vm`. In alternativa, è possibile visualizzare il codice sorgente on-line all'indirizzo <https://svnweb.freebsd.org/>.

Come per molti progetti open-source, questo codice sorgente è archiviato e controllato da un sistema di controllo di versione (*version control system*), in questo caso "subversion" (<https://subversion.apache.org/source-code>). I sistemi di controllo di versione consentono all'utente di prelevare (*pull*) l'intera struttura del codice sorgente sul proprio computer e di sottoporre (*push*) eventuali modifiche nel repository, affinché altre persone possano a loro volta prelevarle. Questi sistemi forniscono anche altre funzionalità, tra cui l'intera cronologia di ciascun file e una funzione di risoluzione dei conflitti nel caso in cui lo stesso file venga modificato contemporaneamente da più utenti. Un altro sistema di controllo di versione è git, utilizzato per gnu/Linux e per molti altri programmi (<http://www.git-scm.com>).

Darwin, il componente kernel fondamentale di macos, è basato su unix bsd ed è anch'esso open-source. Il sorgente è disponibile all'indirizzo <http://www.opensource.apple.com/>. Lo stesso sito contiene tutte le componenti open-source delle distribuzioni di macos. Il nome del pacchetto contenente il kernel comincia con "xnu". Apple fornisce anche diversi strumenti per sviluppatori, documentazione e supporto all'indirizzo <http://developer.apple.com>.

LO STUDIO DEI SISTEMI OPERATIVI

Lo studio dei sistemi operativi non è mai stato così interessante come al giorno d'oggi, e non è mai stato così facilitato.

Con la diffusione del movimento open-source molti sistemi operativi, tra cui Linux, unix bsd, Solaris e una parte di macos sono diventati disponibili sia in formato sorgente sia in formato binario (eseguibile). Disponendo del codice sorgente è possibile studiare i sistemi operativi partendo dal loro interno e rispondere a domande che richiedevano in precedenza lo studio della documentazione o l'osservazione del comportamento di un sistema operativo.

Anche quei sistemi operativi non più attuali dal punto di vista commerciale sono spesso disponibili in versione open-source, il che permette di studiarne il funzionamento su sistemi con cpu lenta e poche risorse di memoria.

All'indirizzo http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/ è reperibile un'ampia lista, seppur non esaustiva, di progetti di sistemi operativi open-source.

Inoltre, l'incremento della virtualizzazione quale funzione principale (e spesso gratuita) del computer permette di far funzionare più sistemi operativi su un unico sistema (*core system*). Per esempio vmware (<http://www.vmware.com>) fornisce un programma gratuito per Windows sul quale possono girare centinaia di applicazioni virtuali gratuite.

Virtualbox (<http://www.virtualbox.com>) mette a disposizione un gestore di macchine virtuali per diversi sistemi operativi, gratuito e open-source. Grazie a questi strumenti gli studenti possono sperimentare centinaia di sistemi operativi senza bisogno di un hardware dedicato.

In alcuni casi sono disponibili anche simulatori di hardware specifico che consentono al sistema operativo di funzionare sull'hardware nativo, anche quando si stanno utilizzando un computer e un sistema operativo moderni. Ad esempio, un simulatore decsystem-20 attivo su macos può avviare tops-20, caricare nastri e modificare e compilare un nuovo kernel tops-20. Se interessato, lo studente può cercare in Internet i manuali e i documenti originali che descrivono il sistema operativo.

L'avvento dei sistemi operativi open-source accorcia la distanza tra studenti e sviluppatori di sistemi operativi. Con qualche conoscenza, un po' d'impegno e una connessione Internet, lo studente può persino creare una nuova distribuzione di un sistema operativo! Solo pochi anni fa era difficile, se non impossibile, accedere al codice sorgente, mentre al giorno d'oggi l'unica limitazione consiste nel tempo, nello spazio su disco e nell'interesse di cui uno studente dispone.

1.11.5 Solaris

Solaris è il sistema operativo commerciale basato su unix della Sun Microsystems. In origine il sistema operativo della Sun, il Sunos, si basava su unix bsd. A partire dal 1991 Sun iniziò a utilizzare come base del suo sistema operativo unix System V di at&t. Nel 2005 la Sun, nell'ambito del progetto OpenSolaris, rese open-source gran parte del codice di Solaris. Nel 2009 l'acquisto di Sun da parte di Oracle ha reso incerto lo stato di questo progetto.

Diversi gruppi interessati all'utilizzo di OpenSolaris sono partiti da questo codice e hanno espanso le sue funzionalità. Il loro progetto si chiama Illumos, è partito dalla base di OpenSolaris introducendo ulteriori funzionalità e vuole essere la base per una serie di prodotti. Illumos è disponibile all'indirizzo <http://wiki.illumos.org>.

1.11.6 Sistemi open-source come strumenti didattici

Il movimento a sostegno dei software gratuiti spinge legioni di programmati a creare migliaia di progetti open-source, inclusi sistemi operativi. Siti come <http://freshmeat.net/> e <http://distrowatch.com/> offrono portali per molti di questi progetti. Come abbiamo detto, i progetti open-source permettono agli studenti di utilizzare il codice sorgente come strumento di apprendimento, dando loro l'opportunità di modificare i programmi, testarli, contribuire all'individuazione di bachi e alla loro eliminazione, oltre a esplorare sistemi operativi maturi e completi, con i relativi compilatori, strumenti, interfacce utente e altri programmi. La disponibilità del codice sorgente per progetti storici, come Multics, può aiutare gli studenti a comprendere quei progetti e a costruire conoscenze utili per nuovi progetti.

Un ulteriore vantaggio dei sistemi operativi open-source è la loro diversità. gnu/Linux e unix bsd sono sistemi operativi open-source, ma ognuno ha obiettivi, funzionalità, licenze e scopi propri. A volte le licenze non si escludono reciprocamente e danno vita a una sorta di impollinazione incrociata che contribuisce a un rapido miglioramento dei progetti riguardanti i sistemi operativi. Per esempio, diverse componenti di rilievo di OpenSolaris sono state trasferite su unix bsd. I vantaggi del software gratuito e dell'open-source possono portare a un aumento del numero e della qualità dei progetti open-source, comportando un incremento del numero di individui e società che li utilizzano.

1.12 Sommario

- Un sistema operativo è il software che gestisce l'hardware di un calcolatore, e fornisce un ambiente all'interno del quale sono eseguibili le applicazioni.
- Le interruzioni sono uno strumento fondamentale utilizzato dall'hardware per interagire con il sistema operativo. Un dispositivo attiva un'interruzione inviando un segnale alla cpu per avvisarla che alcuni eventi richiedono attenzione. L'interruzione è gestita dal gestore delle interruzioni.
- Per essere eseguiti, i programmi devono risiedere nella memoria centrale del calcolatore. Questa è infatti la sola area di memoria di grandi dimensioni direttamente accessibile dalla cpu
- La memoria centrale è un dispositivo volatile, poiché perde il proprio contenuto quando manca l'alimentazione elettrica.
- La memoria non volatile è un'estensione della memoria centrale ed è in grado di memorizzare in modo permanente grandi quantità di dati.
- Il più comune dispositivo di memoria secondaria è il disco magnetico che può memorizzare sia dati sia programmi.
- I sistemi di memorizzazione di un calcolatore si possono organizzare in modo gerarchico secondo la velocità e il costo. I livelli più alti rappresentano i dispositivi più rapidi, ma più costosi. Scendendo nella gerarchia, il costo per bit generalmente decresce, mentre di solito aumentano i tempi d'accesso.
- Le moderne architetture degli elaboratori sono sistemi multiprocessore in cui ogni cpu contiene diverse unità di calcolo (core).
- Per utilizzare al meglio la cpu, i sistemi operativi moderni impiegano la multiprogrammazione, grazie alla quale diversi processi possono occupare contemporaneamente la memoria, assicurando che la cpu non resti mai inattiva.
- Con i sistemi multitasking il concetto di multiprogrammazione è stato ulteriormente esteso per mezzo di algoritmi di scheduling della cpu che commutano rapidamente tra un processo e l'altro, offrendo così agli utenti tempi di risposta rapidi.
- Per evitare che i programmi utenti interferiscano tra di loro e col sistema operativo, la cpu ha due modalità di funzionamento: la modalità utente e la modalità di sistema.
- Diverse istruzioni sono privilegiate e si possono eseguire solamente in modalità di sistema. Tra queste vi sono le istruzioni per il passaggio alla modalità di sistema e le istruzioni per il controllo dell'i/o e per la gestione dei timer e delle interruzioni.
- Un processo è l'unità fondamentale di lavoro in un sistema operativo. La gestione dei processi comprende aspetti come la loro creazione e cancellazione, nonché la messa a punto di meccanismi per la comunicazione reciproca e la sincronizzazione dei processi.
- Un sistema operativo gestisce la memoria mantenendo traccia di quali parti di essa vengono usate e da chi. È sempre al sistema operativo, inoltre, che spetta l'allocazione dinamica e il rilascio dello spazio di memoria.
- Il sistema operativo gestisce l'archiviazione dei dati: ciò comprende la realizzazione del file system per i file e le directory, e la gestione dello spazio sui dispositivi per la memorizzazione di massa.
- I sistemi operativi forniscono meccanismi di protezione e per la sicurezza del sistema e degli utenti. La protezione controlla l'accesso, da parte dei processi o degli utenti, alle risorse che il sistema mette a disposizione.
- La virtualizzazione consiste nell'astrazione dell'hardware di un computer in molteplici distinti ambienti di esecuzione.
- I sistemi operativi utilizzano diverse strutture dati, tra cui: liste, pile, code, alberi e bitmap.
- Vi sono diversi tipi di ambienti elaborativi, che includono l'elaborazione tradizionale, il mobile computing, i sistemi client-server, i sistemi peer-to-peer, il cloud computing e i sistemi operativi embedded real-time.
- I sistemi operativi liberi e open-source sono resi disponibili come codice sorgente. Il software libero ha una licenza d'uso che permette il suo utilizzo gratuito, la sua ridistribuzione e la sua modifica. gnu/Linux, Freebsd e Solaris sono esempi diffusi di sistemi operativi open-source.

Esercizi di ripasso

1.1 Quali sono i tre scopi principali di un sistema operativo?

1.2 Abbiamo sottolineato come il sistema operativo sia volto all'uso efficiente dell'hardware. Quando è opportuno che il sistema operativo rinunci a questo principio e "sprechi" risorse? Perché un sistema simile non può essere considerato davvero inefficiente?

1.3 Qual è la difficoltà principale che deve superare un programmatore nello scrivere un sistema operativo per un ambiente real-time?

1.4 Considerate le varie definizioni di sistema operativo. Valutate se sia opportuno che il sistema operativo includa o meno applicazioni quali browser e programmi di posta elettronica. Argomentate entrambe le possibilità, fornendo delle motivazioni.

1.5 Come funziona la distinzione tra modalità di sistema (modalità kernel) e modalità utente quale rudimentale forma di protezione (sicurezza) del sistema?

1.6 Quale delle seguenti istruzioni dovrebbe essere privilegiata?

- a. Impostare il timer.
- b. Leggere il clock.
- c. Cancellare la memoria.
- d. Invocare un'istruzione trap.
- e. Disattivare le interruzioni.
- f. Modificare le informazioni nella tabella che indica lo status dei dispositivi.
- g. Passare da modalità utente a modalità di sistema.
- h. Accedere a un dispositivo i/o.

1.7 Alcuni dei primi computer proteggevano il sistema operativo posizionandolo in una partizione della memoria che non poteva essere modificata né dall'utente né dal sistema operativo. Descrivete due difficoltà che secondo voi potrebbero sorgere da uno schema simile.

1.8 Alcune cpu offrono più di due modalità di operazione. Quali sono due possibili impieghi di queste modalità multiple?

1.9 I timer potrebbero essere utilizzati anche per calcolare l'ora corrente. Spiegate brevemente come.

1.10 Fornite due ragioni per l'utilizzo delle cache. Quali problemi risolvono? Quali problemi creano? Se una cache potesse essere costruita grande quanto il dispositivo per il quale lavora (per esempio una cache grande come un disco), perché non costruirla così grande ed eliminare il dispositivo?

1.11 Confrontate i modelli di sistemi distribuiti di tipo client-server e di tipo peer-to-peer.

Esercizi

1.12 Quali differenze presentano i cluster di elaboratori rispetto ai sistemi multiprocessore? Che cosa è necessario perché due macchine appartenenti a un cluster cooperino in modo da offrire un servizio altamente affidabile?

1.13 Ipotizziamo che sui due nodi di un cluster di elaboratori sia attiva una base di dati. Descrivete due modalità con cui il software per la gestione del cluster può regolare l'accesso ai dati sul disco, analizzando i pro e i contro di ognuna.

1.14 Qual è lo scopo delle interruzioni? Quali differenze vi sono tra un'eccezione e un'interruzione? Un programma utente può generare un'eccezione di proposito? In caso affermativo, con quale scopo?

1.15 Spiegate come le variabili `hz` e `jiffies` del kernel di Linux siano utilizzabili per stabilire il numero di secondi di esecuzione trascorsi dall'avvio del sistema.

1.16 L'accesso diretto alla memoria (dma) è usato per dispositivi i/o ad alta velocità per evitare di sovraccaricare la cpu.

- In che modo la cpu si interfaccia con il dispositivo per coordinare il trasferimento?
- In che modo la cpu apprende che il trasferimento in memoria è completo?
- La cpu è abilitata all'esecuzione di altri programmi mentre il controllore dma procede al trasferimento dei dati. Può tale trasferimento interferire con la corretta esecuzione dei programmi utenti? Se la risposta è positiva, descrivete in quale forma può sorgere l'interferenza.

1.17 L'hardware di alcuni sistemi elaborativi non possiede una modalità hardware di funzionamento riservata al sistema operativo. Per questi calcolatori è possibile realizzare sistemi operativi sicuri? Fornite motivazioni in favore e contro questa possibilità.

1.18 Molti sistemi smp hanno livelli distinti di cache: un livello interno a ogni core e un livello condiviso tra tutti i core. Perché i sistemi di cache sono progettati in questo modo?

1.19 Ordinate i seguenti sistemi di memorizzazione dal più lento al più veloce:

- Dischi magnetici
- Registri
- Disco ottico
- Memoria centrale
- Memoria non volatile
- Nastri magnetici
- Cache

1.20 Considerate un sistema smp simile a quello della Figura 1.8. Illustrate con un esempio come i dati presenti nella memoria potrebbero avere un valore differente in ognuna delle cache locali.

1.21 Illustrate, con l'ausilio di esempi, come si manifesta il problema della coerenza dei dati memorizzati nella cache nei seguenti ambienti di elaborazione:

- sistemi a processore unico;
- sistemi multiprocessore;
- sistemi distribuiti.

1.22 Descrivete un meccanismo di protezione della memoria grazie al quale sia possibile impedire a un programma la modifica della memoria di pertinenza di altri programmi.

1.23 Quale configurazione di rete – lan o wan – si adatta meglio alle seguenti situazioni?

- Un edificio in un campus universitario.
- Diversi campus localizzati all'interno di una stessa regione.
- Il quartiere di una città.

1.24 Descrivete alcune problematiche da affrontare nel progetto di sistemi operativi per dispositivi mobili in confronto ai pc tradizionali.

- 1.25 Descrivete alcuni vantaggi dei sistemi peer-to-peer rispetto ai sistemi client-server.
- 1.26 Descrivete alcune applicazioni distribuite adatte a un sistema peer-to-peer.
- 1.27 Identificate vantaggi e svantaggi dei sistemi operativi open-source. Discutete anche le tipologie di persone che potrebbero definire determinati aspetti come vantaggi oppure svantaggi.

CAPITOLO 2

Strutture dei sistemi operativi

I sistemi operativi forniscono l'ambiente in cui si eseguono i programmi. Essendo organizzati secondo criteri che possono essere assai diversi, la struttura interna che li caratterizza può variare molto. La progettazione di un nuovo sistema operativo è un compito complesso, perciò è necessario definirne in modo chiaro gli scopi. Il tipo di sistema desiderato definisce i criteri di scelta delle politiche e degli algoritmi utilizzati.

Un sistema operativo si può considerare da diverse angolazioni: secondo i servizi che esso fornisce o l'interfaccia messa a disposizione degli utenti e dei programmatori, oppure secondo i suoi componenti e le relative interconnessioni. In questo capitolo vengono analizzati questi tre aspetti, mostrando il punto di vista dell'utente, del programmatore e del progettista. Si esaminano i servizi offerti da un sistema operativo e la modalità e i metodi da adottare per la sua progettazione. Infine si descrive la creazione e l'inizializzazione del sistema operativo.

2.1 Servizi di un sistema operativo

Un sistema operativo offre un ambiente in cui eseguire i programmi e fornire servizi. Naturalmente, i servizi specifici variano secondo il sistema operativo, ma si possono identificare alcune classi di servizi comuni. Il loro scopo è facilitare il compito dei programmatori di applicazioni. La Figura 2.1 fornisce una panoramica dei servizi del sistema operativo e delle loro relazioni.

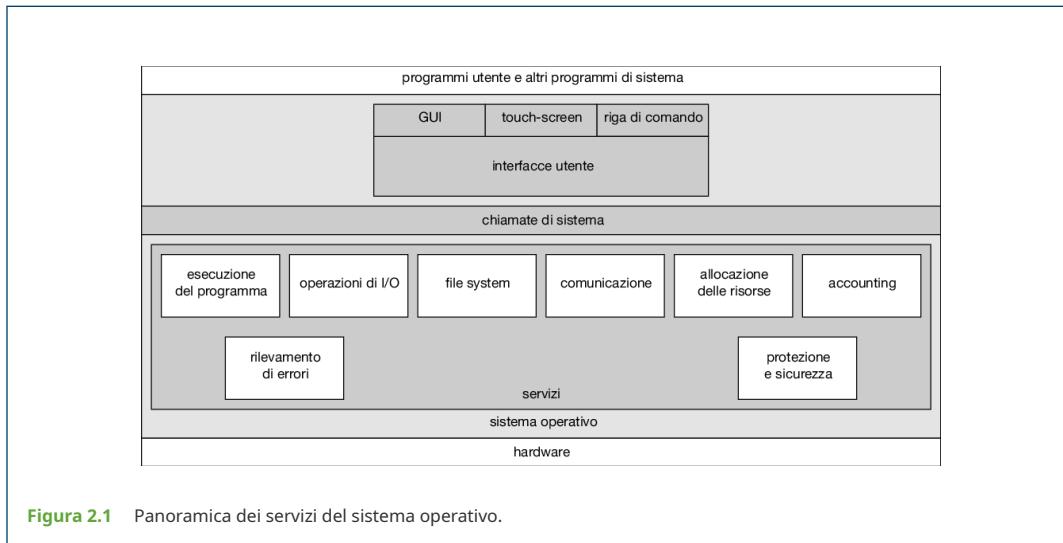


Figura 2.1 Panoramica dei servizi del sistema operativo.

Un primo insieme di servizi offre funzionalità utili all'utente.

- Interfaccia con l'utente. Quasi tutti i sistemi operativi hanno un'interfaccia con l'utente (ui). Essa può assumere diverse forme, ma la forma senz'altro più diffusa è un'interfaccia utente grafica (gui), ossia un sistema a finestre dotato di un dispositivo puntatore (per esempio, il mouse) per comandare operazioni di i/o e selezionare opzioni dai menu, insieme a una tastiera per inserire del testo. I sistemi mobili come smartphone e tablet offrono un'interfaccia touch-screen che consente agli utenti di far scorrere le dita o premere pulsanti sullo schermo per effettuare delle scelte. Un'altra possibilità è l'interfaccia a riga di comando (cli), che utilizza comandi di testo e un mezzo per inserirli (per esempio, una tastiera con cui inserire comandi in un dato formato e con opzioni specifiche). Certi sistemi offrono alcune o anche tutte queste soluzioni.
- Esecuzione di un programma. Il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anomalo (indicando l'errore).
- Operazioni di i/o. Un programma in esecuzione può richiedere un'operazione di i/o che implica l'uso di un file o di un dispositivo di i/o. Per particolari dispositivi possono essere necessarie funzioni speciali, come la registrazione su un cd o dvd, oppure la cancellazione di uno schermo. Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di i/o, quindi il sistema operativo deve offrire mezzi adeguati.
- Gestione del file system. Il file system riveste un interesse particolare. I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file con un certo nome e disporre di informazioni relative al file stesso. Alcuni sistemi operativi, infine, gestiscono i permessi di accesso ai file sulla base della proprietà del file interessato. Molti sistemi operativi offrono all'utente la scelta di file system diversi con funzionalità e prestazioni specifiche.
- Comunicazioni. In molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una memoria condivisa, che permette a due o più processi di leggere e scrivere in una porzione di memoria che condividono, o attraverso lo scambio di messaggi, in questo caso il sistema operativo trasferisce pacchetti d'informazioni in un formato predefinito tra i vari processi.
- Rilevamento di errori. Il sistema operativo deve essere sempre capace di rilevare e correggere eventuali errori che possono verificarsi nella cpu e nei dispositivi di memoria (come un errore di memoria o un guasto all'alimentazione elettrica), nei dispositivi di i/o (come un errore di parità in un nastro, il guasto di una connessione di rete, la mancanza di carta nella stampante) e in un programma utente (come una divisione per zero, un tentativo d'accesso a una locazione di memoria illegale, un uso eccessivo del tempo di cpu). Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore. Talvolta l'unica scelta possibile è l'arresto del sistema, altre volte è possibile terminare il processo che è causa d'errore o restituire un codice d'errore a un processo in modo che da solo cerchi di rilevare e correggere l'errore.

Un secondo gruppo di funzioni del sistema operativo non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso. Sistemi con più utenti possono guadagnare in efficienza condividendo le risorse del calcolatore tra i diversi utenti.

- Allocazione delle risorse. Se sono attivi più utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di cpu, la memoria centrale e la memoria del file system, possono avere un software di gestione molto specifico, mentre altre, come i dispositivi di

i/o, possono avere routine di richiesta e di rilascio più generali. Per esempio, per determinare come utilizzare al meglio la cpu, i sistemi operativi impiegano le procedure di scheduling della cpu, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili e di altri fattori. Esistono anche procedure per l'assegnazione di stampanti, driver di memorizzazione usb e altre periferiche.

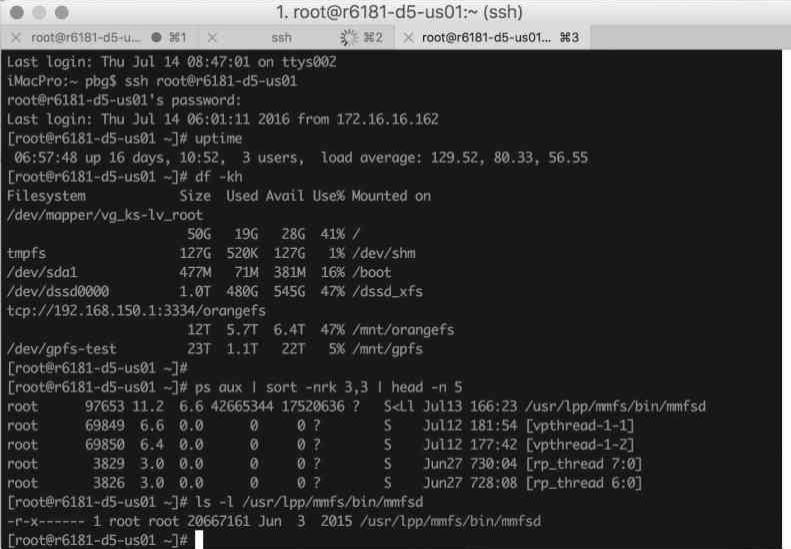
- Logging. Vogliamo mantenere traccia di quali programmi usano il calcolatore, segnalando quali e quante risorse impiegano. Questo tipo di registrazione si può usare per contabilizzare l'uso delle risorse, in modo da addebitare il costo agli utenti, oppure per redigere statistiche; queste ultime possono essere un valido strumento per gli amministratori di sistema che desiderano riconfigurare il sistema per migliorarne i servizi di calcolo.
- Protezione e sicurezza. I proprietari di informazioni memorizzate in un sistema elaborativo multiutente o in rete possono voler controllare l'uso di tali informazioni. Quando più processi separati sono in esecuzione concorrente essi non devono influenzarsi o interferire con il sistema operativo. La protezione assicura che l'accesso alle risorse del sistema sia controllato. È importante anche proteggere il sistema dagli estranei. La sicurezza di un sistema comincia con la richiesta d'identificazione da parte di ciascun utente, di solito attraverso password, per permettere l'accesso alle risorse; si estende fino a difendere i dispositivi di i/o (compresi gli adattatori di rete) dai tentativi d'accesso illegali e provvede al loro rilevamento. Se un sistema deve essere protetto e sicuro, al suo interno devono esistere precauzioni ovunque. La forza di una catena è solo quella del suo anello più debole.

2.2 Interfaccia con l'utente del sistema operativo

Vi sono due modi fondamentali per gli utenti di comunicare con il sistema operativo. Uno si basa su un'interfaccia a riga di comando o interprete dei comandi, che permette agli utenti di inserire direttamente le istruzioni che il sistema deve eseguire. L'altro sfrutta un'interfaccia grafica con l'utente o gui, che serve da tramite tra utente e sistema.

2.2.1 Interprete dei comandi

Nella maggior parte dei sistemi operativi, tra cui Linux, unix e Windows, l'interprete dei comandi è considerato un programma speciale, che si avvia all'avvio di un job o non appena un utente effettua il logon (nel caso di sistemi interattivi). Quando i sistemi consentono la scelta tra molteplici interpreti dei comandi, questi vengono definiti shell. In unix e Linux, per esempio, l'utente può scegliere tra svariate shell differenti, come la C, la Bourne-again, la Korn, e così via. Sono anche disponibili shell di terze parti e shell gratuite scritte dagli utenti. Nella maggior parte dei casi, le shell forniscono funzionalità simili e la scelta di un utente è solitamente dovuta alle preferenze personali. La Figura 2.2 illustra la shell Bourne-again (bash), l'interprete dei comandi utilizzato da macos.



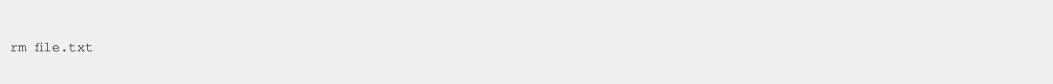
```
1. root@r6181-d5-us01:~ (ssh)
x root@r6181-d5-u... ❶ ssh ❷ ❸ ❹ x root@r6181-d5-us01... ❺
Last login: Thu Jul 14 08:47:01 on ttys002
iMacPro:~ pbgs$ ssh root@r6181-d5-us01
root@r6181-d5-us01's password:
Last login: Thu Jul 14 06:01:11 2016 from 172.16.16.162
[root@r6181-d5-us01 ~]# uptime
 06:57:48 up 16 days, 10:52,  3 users,  load average: 129.52, 80.33, 56.55
[root@r6181-d5-us01 ~]# df -kh
Filesystem      Size  Used Avail Use% Mounted on
/dev/mapper/vg_ks-lv_root
                  50G   19G   28G  41% /
tmpfs          127G  520K  127G  1% /dev/shm
/dev/sda1       477M   71M   381M  16% /boot
/dev/dssd0000   1.0T  480G  545G  47% /dssd_xfs
tcp://192.168.150.1:3334/orangefs
                  12T  5.7T  6.4T  47% /mnt/orangefs
/dev/gpfs-test   23T  1.1T   22T  5% /mnt/gpfs
[root@r6181-d5-us01 ~]#
[root@r6181-d5-us01 ~]# ps aux | sort -nrk 3,3 | head -n 5
root    97653 11.2  6.6 42665344 17520636 ?  S<Ll Jul13 16:23 /usr/lpp/mmfs/bin/mmfsd
root    69849  6.6  0.0     0  0 ?  S  Jul12 18:154 [vpthread-1-1]
root    69850  6.4  0.0     0  0 ?  S  Jul12 17:42 [vpthread-1-2]
root    3829  3.0  0.0     0  0 ?  S  Jun27 730:04 [rp_thread 7:0]
root    3826  3.0  0.0     0  0 ?  S  Jun27 728:08 [rp_thread 6:0]
[root@r6181-d5-us01 ~]# ls -l /usr/lpp/mmfs/bin/mmfsd
-rwx----- 1 root root 20667161 Jun  3 2015 /usr/lpp/mmfs/bin/mmfsd
[root@r6181-d5-us01 ~]#
```

Figura 2.2 La shell bash, l'interprete dei comandi utilizzato in macOS.

La funzione principale dell'interprete dei comandi consiste nel raccogliere ed eseguire il successivo comando impartito dall'utente. A questo livello, la maggioranza dei comandi riguarda la gestione dei file: creazione, cancellazione, elenco, stampa, copia, esecuzione, e così via. Le varie shell di unix funzionano in questo modo. I comandi si possono implementare in due modi.

Nel primo, lo stesso interprete dei comandi contiene il codice per l'esecuzione del comando. Il comando di cancellazione di un file, per esempio, può causare un salto dell'interprete dei comandi a una sezione del suo stesso codice che imposta i parametri e invoca le idonee chiamate di sistema; in questo caso, poiché ogni comando richiede il proprio segmento di codice, il numero dei comandi che si possono impartire determina le dimensioni dell'interprete dei comandi.

L'altro metodo, usato per esempio nel sistema operativo unix, implementa la maggior parte dei comandi per mezzo di programmi di sistema; in questo caso l'interprete dei comandi non "capisce" il significato del comando, ma ne impiega semplicemente il nome per identificare un file da caricare in memoria per l'esecuzione. Quindi, il comando unix per cancellare un file



```
rm file.txt
```

cerca un file chiamato `rm`, lo carica in memoria e lo esegue con il parametro `file.txt`. La funzione corrispondente al comando `rm` è interamente definita dal codice del file `rm`. In questo modo i programmati possono aggiungere nuovi comandi al sistema.

semplicemente creando nuovi file con il nome appropriato. Il codice dell'interprete dei comandi, che può quindi essere abbastanza piccolo, non necessita di alcuna modifica quando s'introducono nuovi comandi.

2.2.2 Interfaccia grafica con l'utente

Un'interfaccia grafica con l'utente o gui rappresenta una seconda modalità di comunicazione con il sistema operativo, più *user-friendly*. Infatti, invece di obbligare gli utenti a digitare direttamente i comandi, nella gui l'interfaccia è costituita da una o più finestre e dai relativi menu, entro cui muoversi con il mouse. La gui utilizza la metafora della scrivania (desktop) in cui, spostando il puntatore con il mouse, si selezionano immagini o icone sullo schermo (il desktop): queste rappresentano programmi, file, directory e funzioni del sistema. A seconda della posizione del puntatore, cliccando un pulsante del mouse si può invocare un programma, selezionare un file o una directory – notti in questo contesto come cartella (folder) – o far apparire un menu a tendina contenente comandi.

Le interfacce grafiche si affacciarono sulla scena, da principio, per effetto delle ricerche condotte nei primi anni '70 dai laboratori di ricerca Xerox parc. La prima gui apparve sul computer Xerox Alto nel 1973. Tuttavia, una maggiore diffusione delle interfacce grafiche si ebbe con l'avvento dei computer Apple Macintosh negli anni '80. L'interfaccia utente con il sistema operativo Macintosh ha subito, nel corso degli anni, diverse modifiche, la più significativa delle quali è stata l'adozione dell'interfaccia *Aqua* per il macos. La prima versione di Microsoft Windows, cioè la 1.0, era basata sull'aggiunta di un'interfaccia gui al sistema operativo ms-dos. I successivi sistemi Windows hanno apportato ritocchi cosmetici significativi all'aspetto della gui e una serie di miglioramenti sul piano della funzionalità.

Nei sistemi unix, tradizionalmente, le interfacce a riga di comando hanno avuto un ruolo preponderante. Sono tuttavia disponibili diverse interfacce gui: un impulso determinante nel loro sviluppo è giunto da vari progetti open-source come il kde (*K desktop environment*) e il desktop gnome del progetto gnu. Ambedue i desktop, kde e gnome, sono compatibili con Linux e con vari sistemi unix, e sono regolati da licenza open-source, vale a dire che il loro codice sorgente è reso disponibile per consultazioni e per modifiche soggette a specifiche condizioni di licenza.

2.2.3 Interfaccia touch-screen

Dato che un'interfaccia a linea di comando o un sistema composto da mouse e tastiera non risultano pratici per la maggior parte dei sistemi mobili, gli smartphone e i tablet utilizzano in genere un'interfaccia touch-screen, in cui gli utenti interagiscono mediante gesti (*gesture*) sullo schermo, per esempio premendo e scorrendo le dita. Anche se in passato gli smartphone erano dotati di una tastiera fisica, attualmente la maggior parte degli smartphone e dei tablet simula una tastiera sul touch-screen. La Figura 2.3 mostra il touch-screen di un iPhone. Sia l'iPad sia l'iPhone utilizzano un'interfaccia touch-screen chiamata Springboard.



Figura 2.3 Il touch-screen di un iPhone.

2.2.4 Scelta dell'interfaccia

La scelta di un'interfaccia a riga di comando piuttosto che gui dipende in buona misura dalle preferenze personali. In linea di massima, gli amministratori di sistema e gli utenti più esperti optano spesso per le interfacce a riga di comando, per loro più efficienti in quanto forniscono un accesso più veloce alle attività che tali tipologie di utenti devono effettuare. In molti sistemi, in effetti, solo un sottosistema delle funzionalità del sistema è disponibile attraverso la gui e le funzioni meno comuni sono accessibili solo tramite riga di comando. Le interfacce a riga di comando semplificano l'esecuzione di comandi ripetuti, perché sono programmabili. Per esempio, se un task viene eseguito di frequente ed è costituito da più comandi è possibile registrare la sequenza di comandi in un unico file ed eseguire il file esattamente come si fa con un programma. Questo programma non viene compilato, ma è interpretato dall'interfaccia a riga di comando. Questi shell script sono molto comuni su sistemi orientati alla riga di comando come Unix e Linux.

Molti utenti Windows, d'altro canto, sono soddisfatti dell'ambiente Windows gui, e per questo motivo non usano quasi mai la shell. Le versioni recenti del sistema operativo Windows forniscono sia una gui standard per desktop e laptop tradizionali sia una interfaccia touch-screen per tablet. I sistemi operativi Macintosh, con i molti cambiamenti subiti, costituiscono un utile caso di studio da confrontare alla situazione di unix e Windows. Fino a tempi recenti, il Mac os non disponeva di un'interfaccia a riga di comando, e

vincolava l'interazione degli utenti con il sistema alla propria interfaccia gui. Tuttavia, con l'introduzione di macos (realizzato, in parte, sfruttando il kernel unix), il sistema operativo contiene ora sia l'interfaccia grafica Aqua sia un'interfaccia a riga di comando. La Figura 2.4 mostra una schermata dell'interfaccia grafica di macos.



Figura 2.4 Interfaccia grafica di macOS.

Sebbene esistano app per i sistemi mobili ios e Android che forniscono un'interfaccia a riga di comando, queste vengono utilizzate raramente e quasi tutti gli utenti dei sistemi mobili interagiscono con i dispositivi utilizzando l'interfaccia touch-screen.

L'interfaccia con l'utente può cambiare da sistema a sistema e persino da utente a utente all'interno dello stesso sistema; in genere è ben distinta dalla struttura fondamentale del sistema. La progettazione di un'interfaccia utile e intuitiva per l'utente non è, pertanto, intrinsecamente legata al sistema operativo. In questo libro vengono evidenziati i problemi correlati alla prestazione di un servizio adeguato ai programmi utenti: dal punto di vista del sistema operativo non si applicherà alcuna distinzione tra programmi utenti e programmi del sistema.

2.3 Chiamate di sistema

Le chiamate di sistema (*system call*) costituiscono un'interfaccia per i servizi resi disponibili dal sistema operativo. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in c o c++, sebbene per alcuni compiti di basso livello, come quelli che comportano un accesso diretto all'hardware, possa essere necessario il ricorso al linguaggio assembly.

2.3.1 Esempio

Prima di illustrare come le chiamate di sistema vengano rese disponibili da parte del sistema operativo, consideriamo un esempio del loro uso: la scrittura di un semplice programma che legga i dati da un file e li trascriva in un altro. La prima informazione di cui il programma necessita è costituita dai nomi dei due file: il file in ingresso e il file in uscita. Questi nomi si possono indicare in molti modi diversi, secondo la struttura del sistema operativo. Un primo metodo consiste nel passare i nomi dei due file come argomenti di un comando, per esempio del comando unix `cp`:

```
cp in.txt out.txt
```

Questo comando copia il file in ingresso `in.txt` nel file in uscita `out.txt`. Un secondo approccio prevede che il programma chieda all'utente i nomi dei due file. In un sistema interattivo questa operazione necessita di una sequenza di chiamate di sistema, innanzitutto per scrivere un messaggio di richiesta sullo schermo e quindi per leggere dalla tastiera i caratteri che compongono i nomi dei due file. Nei sistemi basati su mouse e finestre in genere appare in una finestra un menu contenente i nomi dei file. L'utente può usare il mouse per scegliere il nome del file di origine, dopodiché è possibile aprire un'altra finestra in cui specificare il nome del file di destinazione. Come vedremo, questa sequenza richiede molte chiamate di sistema di i/o.

Una volta ottenuti i nomi, il programma deve aprire il file in ingresso e creare il file di destinazione. Ciascuna di queste operazioni richiede un'altra chiamata di sistema e può andare incontro a condizioni d'errore che richiedono ulteriori chiamate. Per esempio, quando il programma tenta di aprire il file in ingresso, può scoprire che non esiste alcun file con quel nome, oppure che l'accesso al file è protetto. In questi casi il programma deve scrivere un messaggio nello schermo della console (altra sequenza di chiamate di sistema) e quindi terminare in maniera anomala la propria elaborazione (ulteriore chiamata di sistema). Se il file in ingresso esiste, è necessario creare il file di destinazione. È possibile che esista già un file col nome indicato per il file di destinazione; questa situazione potrebbe causare l'interruzione del programma (una chiamata di sistema) o la cancellazione del file esistente (un'altra chiamata di sistema) e la creazione di uno nuovo (ancora un'altra chiamata di sistema). Un'ulteriore possibilità, in un sistema interattivo, è quella di richiedere all'utente (attraverso una sequenza di chiamate di sistema per emettere il messaggio di richiesta e per leggere la risposta dal terminale) se si debba sostituire il file già esistente o terminare l'esecuzione del programma.

Una volta predisposti i due file, si entra in un ciclo che legge dal file in ingresso (una chiamata di sistema) e scrive nel file di destinazione (altra chiamata di sistema). Ciascuna lettura (`read`) e scrittura (`write`) devono riportare informazioni di stato relative alle possibili condizioni d'errore. Quando effettua l'input, il programma può rilevare che è stata raggiunta la fine del file, oppure che nella lettura si è riscontrato un errore hardware, per esempio un errore di parità. Nella fase di scrittura si possono verificare vari errori, la cui natura dipende dal dispositivo impiegato (per esempio l'esaurimento dello spazio su disco).

Infine, una volta copiato tutto il file, il programma può chiuderlo entrambi (altre chiamate di sistema), inviare un messaggio alla console o alla finestra (più chiamate di sistema) e terminare normalmente (ultima chiamata di sistema). Tale sequenza di chiamate di sistema è illustrata nella Figura 2.5.

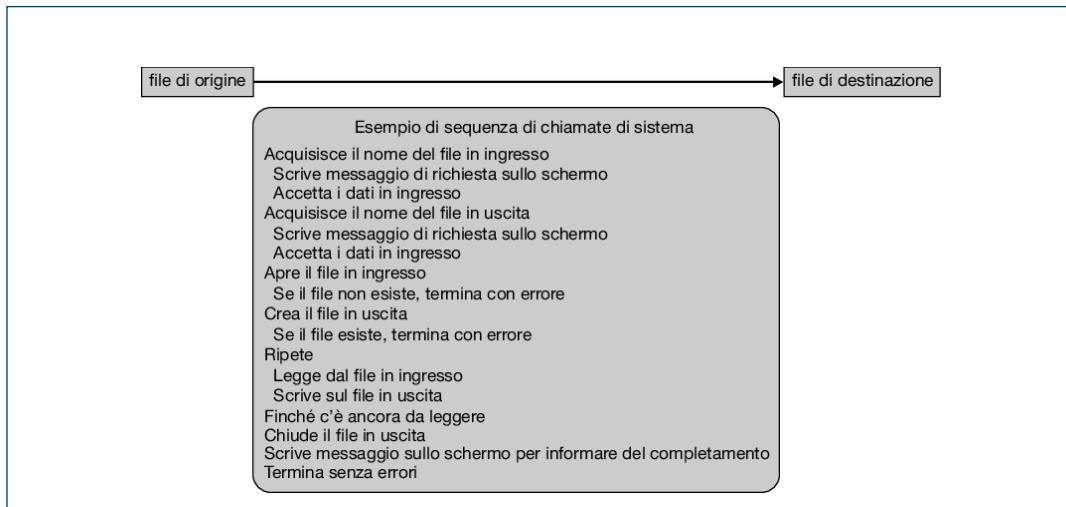


Figura 2.5 Esempio d'uso delle chiamate di sistema.

2.3.2 Interfaccia per la programmazione di applicazioni (API)

Come si è visto, anche programmi molto semplici possono fare un intenso uso del sistema operativo. Non è raro che un sistema esegua migliaia di chiamate di sistema al secondo. La maggior parte dei programmatore, tuttavia, non si dovrà mai preoccupare di questi dettagli: infatti, gli sviluppatori di applicazioni usano in genere un'interfaccia per la programmazione di applicazioni (api, *application programming interface*). Essa specifica un insieme di funzioni a disposizione del programmatore e dettaglia i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti. Tre delle interfacce più diffuse disponibili ai programmatore di applicazioni sono la api di Windows, la api posix per i sistemi basati sullo standard posix (il che include praticamente tutte le versioni di unix, Linux e macos) e la api Java per applicazioni eseguite dalla macchina virtuale Java. Un programmatore ha accesso a un'api tramite una libreria di codice fornita dal sistema operativo. Per programmi in C in ambienti Unix e Linux tale libreria si chiama libc. Si noti che (se non diversamente specificato) i nomi delle chiamate di sistema che ricorrono in questo libro sono esempi generici; un dato sistema operativo adotterà nomi suoi propri per ogni system call.

Dietro le quinte, le funzioni fornite da un'api invocano le chiamate di sistema per conto del programmatore. Per esempio la funzione Windows `CreateProcess()`, che ovviamente serve a generare un nuovo processo, invoca in effetti `NTCreateProcess()`, una chiamata di sistema del kernel di Windows.

ESEMPIO DI API STANDARD

Come esempio di api standard consideriamo la funzione `read()` disponibile in Unix e Linux. L'api per questa funzione si può ottenere digitando

- man read

da riga di comando. Una descrizione di questa api è la seguente:

```
#include <unistd.h>
ssize_t      read(int fd, void *buf, size_t count)
```

valore nome parametri
restituito della funzione

Un programma che utilizza la `read()` deve includere il file `unistd.h` che, tra le altre cose, definisce i tipi di dato `ssize_t` e `size_t`. I parametri passati alla `read()` sono i seguenti:

- `int fd` — il descrittore del file da leggere
- `void *buf` — un buffer nel quale vengono messi i dati letti
- `size_t count` — il massimo numero di byte da leggere e inserire nel buffer

Quando una `read()` è completata con successo viene restituito il numero di byte letti. La `read()` restituisce 0 in caso di fine del file e -1 quando si è verificato un errore.

Ci sono molte ragioni per cui è preferibile, per un programmatore, sfruttare l'intermediazione della api piuttosto che invocare direttamente le chiamate di sistema. Una di queste è legata alla portabilità delle applicazioni: ci si può aspettare che un programma sviluppato sulla base di una certa api possa venire compilato ed eseguito su qualunque sistema che la metta a disposizione (anche se le differenze architettoniche possono rendere questa operazione non del tutto indolare). Inoltre, le chiamate di sistema sono spesso più dettagliate e difficili da usare di una api. Bisogna però dire che vi è spesso una stretta correlazione tra le funzioni di una api e le associate chiamate di sistema all'interno del kernel. In effetti, molte funzioni delle api posix e Windows sono simili alle chiamate di sistema fornite dai sistemi operativi unix, Linux e Windows.

Un altro fattore importante nella gestione delle chiamate di sistema è l'ambiente di esecuzione al run-time (rte, *run-time environment*), la suite completa di programmi necessaria per eseguire applicazioni scritte in un determinato linguaggio di programmazione, che include i compilatori o gli interpreti e altri software, come librerie e loader. L'rte fornisce un'interfaccia alle chiamate di sistema (*system call interface*) che collega il linguaggio alle system call rese disponibili dal sistema operativo. L'interfaccia intercetta le chiamate a funzioni nella api e invoca le relative system call. Di solito, ogni chiamata di sistema è codificata da un numero. L'interfaccia alle chiamate di sistema mantiene una tabella delle chiamate e invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, restituendo al chiamante lo stato della chiamata.

Il chiamante non ha alcuna necessità di conoscere l'implementazione della chiamata di sistema o i dettagli della sua esecuzione: gli è sufficiente essere conforme alla specifica della api e conoscere l'effetto dell'esecuzione della chiamata di sistema operativo. Ne consegue che la gran parte dei dettagli relativi alle chiamate di sistema è nascosta al programmatore dalla api e gestita dal sistema di supporto all'esecuzione. Le relazioni fra una api, l'interfaccia alle chiamate di sistema e il sistema operativo sono illustrate nella Figura 2.6, ove si mostra come il sistema operativo tratti l'invocazione della chiamata di sistema `open()` da parte di un'applicazione.

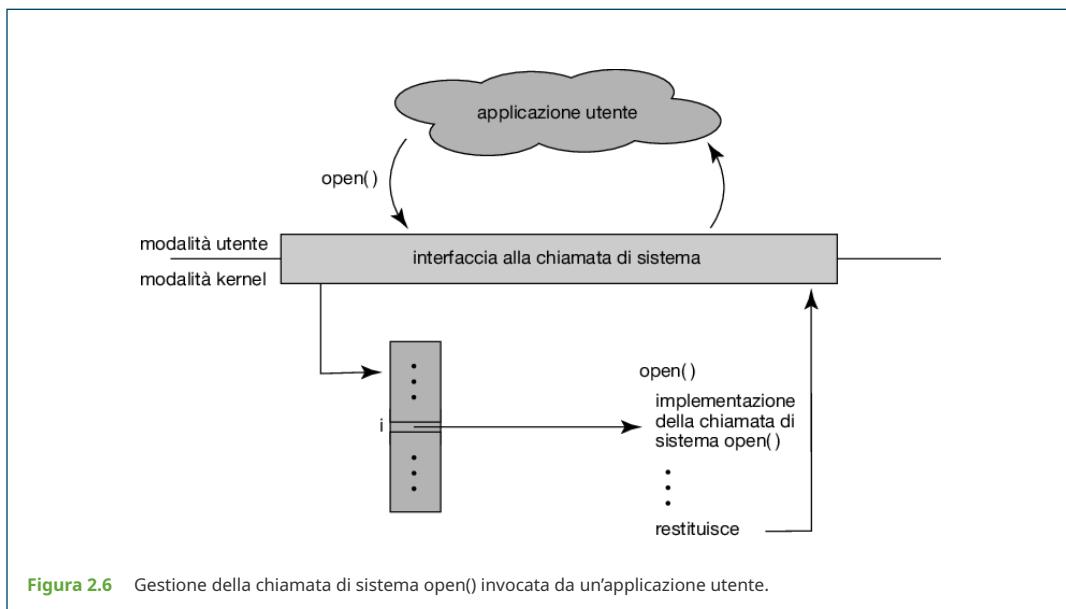


Figura 2.6 Gestione della chiamata di sistema `open()` invocata da un'applicazione utente.

Le system call si presentano in modi diversi, secondo il calcolatore in uso. Spesso sono richieste maggiori informazioni oltre alla semplice identità della chiamata di sistema desiderata. Il tipo e la quantità delle informazioni variano secondo lo specifico sistema operativo e la specifica chiamata di sistema. Per ottenere l'immissione di un dato, per esempio, può essere necessario specificare il file o il dispositivo da usare come sorgente e anche l'indirizzo e la dimensione del buffer in memoria in cui depositare i dati letti. Naturalmente, il dispositivo o il file e la dimensione possono essere impliciti nella chiamata di sistema.

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in *registri*: si possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un *blocco* o tabella di memoria e si passa l'indirizzo del blocco, come parametro, in un registro (Figura 2.7). Linux fa uso di una combinazione di questi approcci. Se sono presenti al massimo cinque parametri vengono utilizzati i registri, altrimenti viene utilizzato il metodo del blocco. Un programma può anche collocare (*push*) i parametri nello stack da cui sono prelevati (*pop*) dal sistema operativo. Alcuni sistemi operativi preferiscono i metodi del blocco o dello stack, poiché non limitano il numero o la lunghezza dei parametri da passare.

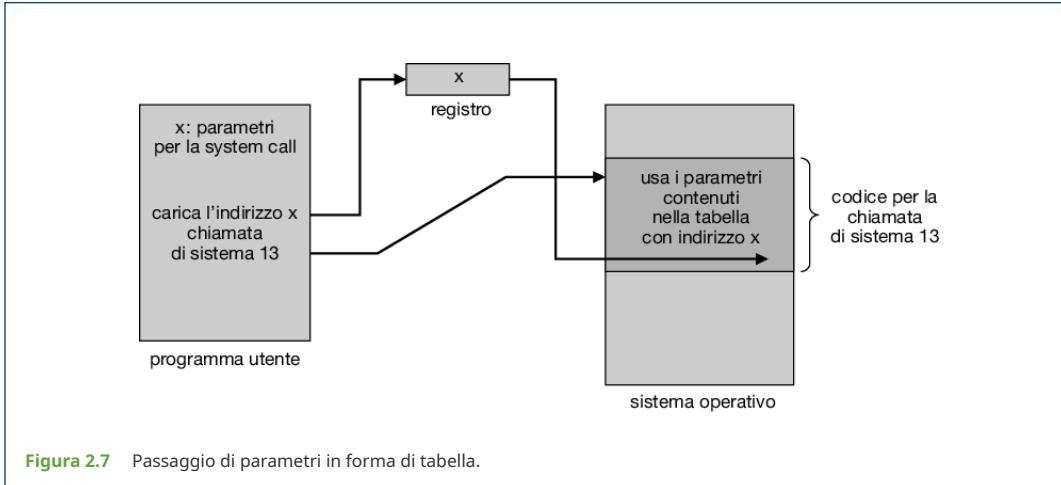


Figura 2.7 Passaggio di parametri in forma di tabella.

2.3.3 Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in sei categorie principali: controllo dei processi, gestione dei file, gestione dei dispositivi, gestione delle informazioni, comunicazioni e protezione. Di seguito sono illustrati brevemente i tipi di chiamate di sistema forniti da un sistema operativo. La maggior parte di queste chiamate di sistema implica o presuppone concetti e funzioni trattati in capitoli successivi. La Figura 2.8 riassume i tipi di chiamate di sistema forniti normalmente da un sistema operativo. Come già detto, in questo testo solitamente facciamo riferimento alle chiamate di sistema con denominazioni generiche. In tutto il libro, tuttavia, forniamo esempi delle controparti reali delle chiamate per i sistemi unix, Linux e Windows.

- Controllo dei processi
 - creazione e arresto di un processo
 - caricamento, esecuzione
 - terminazione normale e anomala
 - esame e impostazione degli attributi di un processo
 - attesa per il tempo indicato
 - attesa e segnalazione di un evento
 - assegnazione e rilascio di memoria
- Gestione dei file
 - creazione e cancellazione di file
 - apertura, chiusura
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
 - richiesta e rilascio di un dispositivo
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un dispositivo
 - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
 - esame e impostazione dell'ora e della data
 - esame e impostazione dei dati del sistema
 - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
 - creazione e chiusura di una connessione
 - invio e ricezione di messaggi
 - informazioni sullo stato di un trasferimento
 - inserimento ed esclusione di dispositivi remoti
- Protezione
 - visualizzazione dei permessi di un file
 - impostazione dei permessi di un file

Figura 2.8 Tipi di chiamate di sistema.

2.3.3.1 Controllo dei processi

Un programma in esecuzione deve potersi fermare in modo sia normale (`end()`) sia anomalo (`abort()`). Talvolta, se si ricorre a una chiamata di sistema per terminare in modo anomalo un programma in esecuzione, oppure se il programma incontra un problema e segnala l'errore con un'eccezione, un'immagine del contenuto della memoria viene copiata in un file (*dump*) e viene generato un messaggio d'errore. Il programmatore può usare uno specifico programma di ricerca e correzione di errori o bug (*debugger*) per esaminare tali informazioni e determinare le cause del problema. Sia in condizioni normali sia anomale il sistema operativo deve trasferire il controllo all'interprete dei comandi che legge il comando successivo. In un sistema interattivo l'interprete dei comandi continua semplicemente a interpretare il comando successivo; si suppone che l'utente invii un comando idoneo per rispondere a qualsiasi errore. In un sistema a interfaccia gui una finestra avverte l'utente dell'errore e richiede indicazioni. In un sistema a lotti l'interprete dei comandi generalmente abortisce il lavoro corrente e prosegue con il successivo. Quando si presenta un errore, alcuni sistemi possono permettere specifiche azioni di recupero. Se il programma scopre un errore nei dati ricevuti e intende terminare in modo anomalo, può anche definire un livello d'errore. Più grave è l'errore, più alto è il livello del parametro che lo individua. È quindi possibile indicare in maniera uniforme una terminazione normale e una terminazione anomala, definendo la terminazione normale come errore di livello 0. L'interprete dei comandi o un programma successivo possono usare questo livello d'errore per determinare l'azione da intraprendere.

ESEMPIO DI CHIAMATE DI SISTEMA DI WINDOWS E UNIX

WindowsUNIX

```
ControlloCreateProcess() fork()  
  
dei processiExitProcess() exit()  
  
WaitForSingleObject() wait()  
  
GestioneCreateFile() open()  
  
dei fileReadFile() read()  
  
WriteFile() write()  
  
CloseHandle() close()  
  
Gestione SetConsoleMode() ioctl()  
  
dei dispositiviReadConsole() read()  
  
WriteConsole() write()  
  
Gestione delleGetCurrentProcessID() getpid()  
  
informazioniSetTimer() alarm()  
  
Sleep() sleep()  
  
ComunicazioneCreatePipe() pipe()  
  
CreateFileMapping() shm_open()  
  
MapViewOfFile() mmap()
```

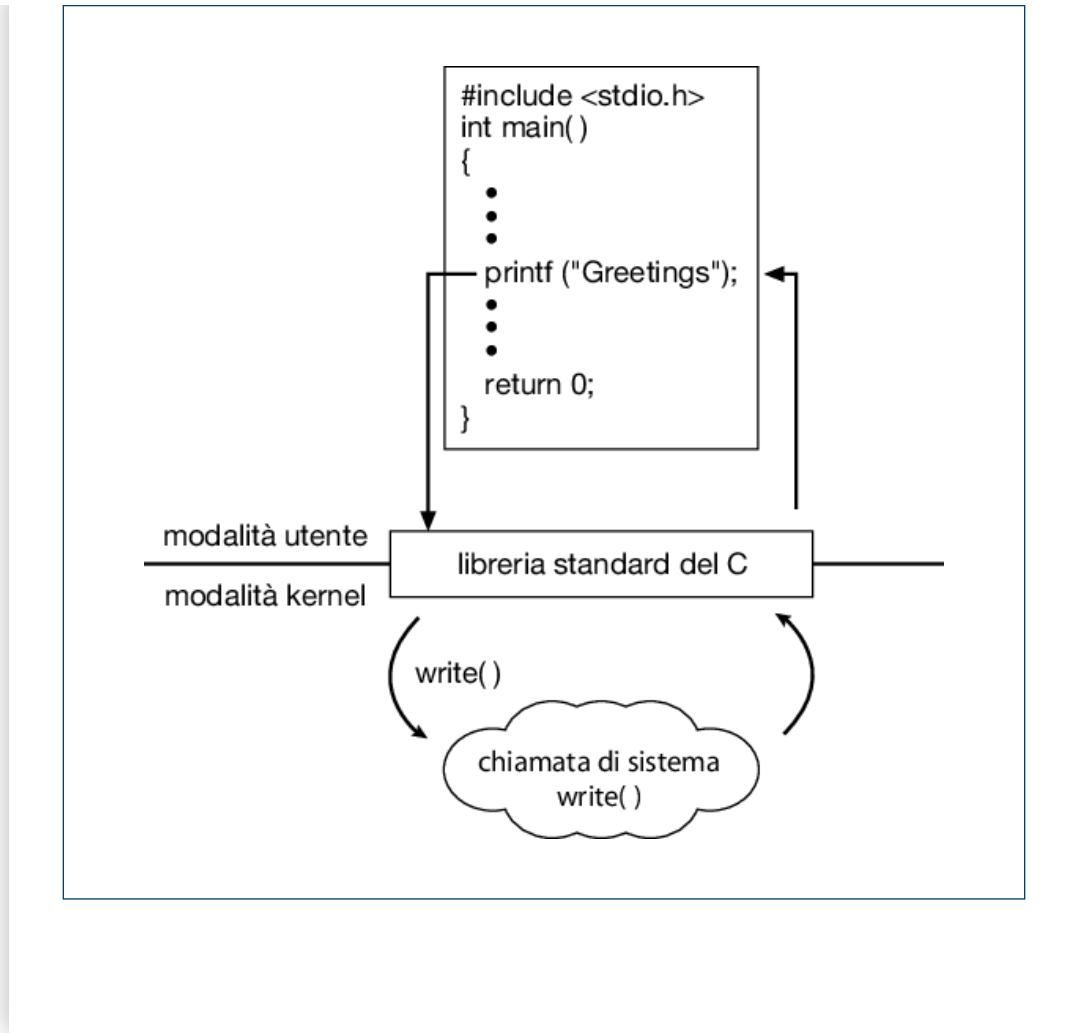
```
ProtezioneSetFileSecurity() chmod()  
  
InitializeSecurityDescriptor() umask()  
  
SetSecurityDescriptorGroup() chown()
```

Un processo che esegue un programma può richiedere di caricare (`load()`) ed eseguire (`execute()`) un altro programma. In questo modo l'interprete dei comandi esegue un programma in seguito a una richiesta impartita, per esempio, da un comando utente, oppure dal clic di un mouse o da un comando batch. È interessante chiedersi dove si debba restituire il controllo una volta terminato il programma caricato. La questione è legata alle eventualità che il programma attuale sia terminato, sia stato sospeso oppure che abbia continuato l'esecuzione in modo concorrente con il nuovo programma.

Se al termine del nuovo programma il controllo rientra nel programma attuale, si deve salvare l'immagine della memoria del programma attuale, creando così effettivamente un meccanismo con cui un programma può richiamare un altro programma. Se entrambi i programmi continuano l'esecuzione in modo concorrente, si è creato un nuovo processo da eseguire in multiprogrammazione. A questo scopo spesso si ha una chiamata di sistema specifica (`create_process()`).

LA LIBRERIA STANDARD DEL LINGUAGGIO C

La libreria standard del linguaggio C fornisce una parte dell'interfaccia alle chiamate di sistema per molte versioni di unix e Linux. Come esempio, supponiamo che un programma C invochi la funzione `printf()`. La libreria C intercetta la funzione e invoca le necessarie chiamate di sistema: in questo caso, la chiamata `write()`. La libreria riceve il valore restituito da `write()` e lo passa al programma utente.



Quando si crea un nuovo processo, o anche un insieme di processi, è necessario mantenerne il controllo; ciò richiede la capacità di determinare e reimpostare gli attributi di un processo, compresi la sua priorità, il suo tempo massimo d'esecuzione e così via (`get_process_attributes()` e `set_process_attributes()`). Inoltre, può essere necessario terminare un processo creato, se si riscontra che non è corretto o se la sua esecuzione non è più utile (`terminate_process()`).

Una volta creati, può essere necessario attendere che i processi terminino la loro esecuzione. Quest'attesa si può impostare per un certo periodo di tempo (`wait_time()`), ma è più probabile che si preferisca attendere che si verifichi un dato evento (`wait_event()`). In tal caso i processi devono segnalare il verificarsi di quell'evento (`signal_event()`).

Molto spesso due o più processi possono condividere dati. Per assicurare l'integrità dei dati che vengono condivisi, spesso i sistemi operativi forniscono chiamate di sistema che consentono a un processo di bloccare (lock) dati condivisi, di modo che nessun altro processo possa accedere ai dati fino al rilascio del blocco. In genere tali chiamate di sistema includono `acquire_lock()` e `release_lock()`. Chiamate di sistema di questo tipo, che trattano cioè il coordinamento di processi concorrenti, sono esaminate in profondità nei Capitoli 6 e 7.

Il controllo dei processi presenta così tanti aspetti e varianti che per chiarire questi concetti conviene ricorrere a due esempi, riguardanti uno un sistema monoprogrammato e l'altro un sistema multitasking. Arduino è una semplice piattaforma hardware composta da un microcontrollore e da sensori di ingresso che rispondono a diversi eventi, come, per esempio, cambiamenti di luce, temperatura e pressione barometrica. Per scrivere un programma per Arduino si scrive prima il programma su un pc e poi si carica il programma compilato (noto come *sketch*) dal pc alla memoria flash di Arduino tramite una connessione usb. La piattaforma Arduino standard non fornisce un sistema operativo, ma solamente un piccolo software noto come boot loader, che carica lo sketch in una regione specifica della memoria di Arduino (si veda la Figura 2.9). Una volta che lo sketch è stato caricato, inizia la sua esecuzione, restando in attesa degli eventi ai quali è programmato per rispondere. Per esempio, se il sensore di temperatura rileva il superamento di una certa soglia, lo sketch può chiedere ad Arduino di avviare il motore di un ventilatore. Arduino è considerato un sistema monoprogrammato, perché può essere presente in memoria soltanto uno sketch alla volta; se viene caricato un nuovo sketch, lo sketch esistente viene rimpiazzato. Inoltre, Arduino non fornisce alcuna interfaccia utente oltre ai sensori hardware d'ingresso.

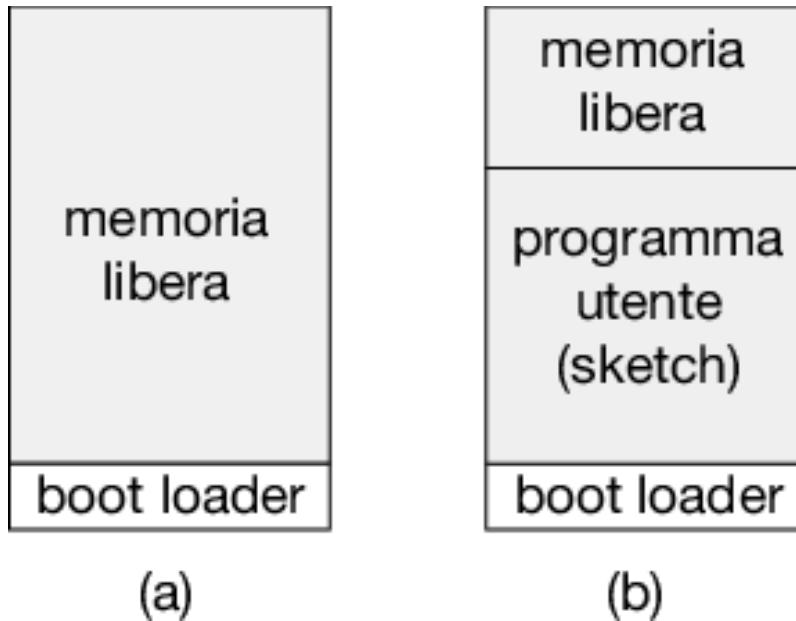


Figura 2.9 Esecuzione in Arduino. (a) All'avviamento del sistema. (b) Durante l'esecuzione di un programma.

Freebsd (derivato da unix Berkeley) è un esempio di sistema multitasking. Quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (*shell*) scelto dall'utente. Tuttavia, poiché il Freebsd è un sistema multitasking, l'interprete dei comandi può continuare l'esecuzione mentre si esegue un altro programma (Figura 2.10). Per avviare un nuovo processo, la shell (interprete dei comandi) esegue la chiamata di sistema `fork()`; si carica il programma selezionato in memoria tramite la chiamata di sistema `exec()` e infine si esegue il programma. A seconda di come il comando è stato impostato, la shell attende il termine del processo, oppure esegue il processo in *background*. In quest'ultimo caso la shell richiede immediatamente un altro comando. Se un processo è eseguito in *background*, non può ricevere dati direttamente dalla tastiera, giacché anche la shell sta usando tale risorsa. L'eventuale operazione di i/o è dunque eseguita tramite un file o tramite un'interfaccia gui. Nel frattempo l'utente è libero di richiedere alla shell l'esecuzione di altri, di controllare lo svolgimento del processo in esecuzione, di modificare la priorità di quel programma e così via. Completato il proprio compito, il processo esegue una chiamata di sistema `exit()` per terminare la propria esecuzione, riportando al processo chiamante un codice di stato 0 oppure un codice d'errore diverso da 0. Questo codice di stato (o d'errore) rimane disponibile per la shell o per altri programmi. I processi sono trattati nel Capitolo 3, dove si illustra un esempio di programma che utilizza le chiamate di sistema `fork()` ed `exec()`.

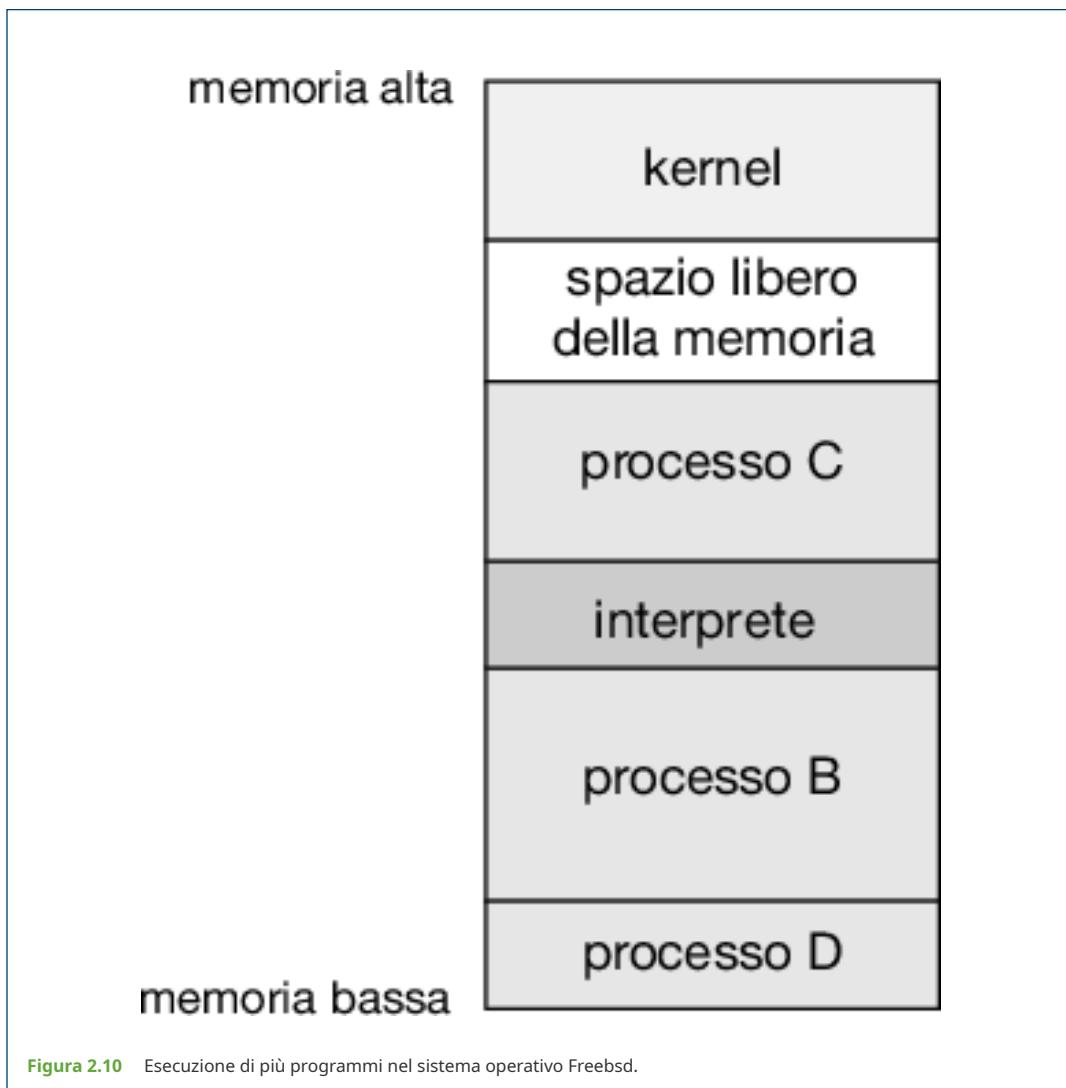


Figura 2.10 Esecuzione di più programmi nel sistema operativo Freebsd.

2.3.3.2 Gestione dei file

Il file system è esaminato più in profondità nei Capitoli 13, 14 e 15, tuttavia possiamo già identificare diverse chiamate di sistema riguardanti i file.

Innanzitutto è necessario poter creare (`create()`) e cancellare (`delete()`) i file. Ogni chiamata di sistema richiede il nome del file e probabilmente anche altri attributi. Una volta creato il file è necessario aprirlo (`open()`) e usarlo. Si può anche leggere (`read()`), scrivere (`write()`) o riposizionare (`reposition()`), per esempio riavvolgendo e saltando alla fine del file. Si deve infine poter chiudere (`close()`) un file per indicare che non è più in uso.

Queste stesse operazioni possono essere necessarie anche per le directory, nel caso in cui il file system sia strutturato in directory. È inoltre necessario poter determinare i valori degli attributi dei file o delle directory ed eventualmente modificarli. Tra gli attributi dei file figurano: nome, tipo, codici di protezione, informazioni di accounting, e così via. Per queste funzioni sono richieste almeno due chiamate di sistema, e precisamente `get_file_attributes()` e `set_file_attributes()`. Alcuni sistemi operativi forniscono molte più chiamate di sistema, per esempio per spostare (`move()`) e copiare (`copy()`) file. Altri forniscono api che eseguono operazioni di questo tipo tramite codice appropriato combinato a chiamate di sistema, e altri ancora mettono semplicemente a disposizione programmi di sistema che le eseguono. Se i programmi di sistema sono invocabili da altri programmi, ognuno di loro funge da api per altri programmi.

2.3.3.3 Gestione dei dispositivi

Per essere eseguito un processo necessita di parecchie risorse: spazio in memoria, spazio su disco, accesso a file, e così via. Se le risorse sono disponibili, si possono concedere e il controllo può ritornare al processo utente, altrimenti il processo deve attendere finché non siano disponibili risorse sufficienti.

Le diverse risorse controllate dal sistema operativo si possono concepire come dei dispositivi, alcuni dei quali sono in effetti dispositivi fisici (per esempio unità disco), mentre altre sono da considerarsi dispositivi astratti o virtuali (i file, per esempio). In presenza di utenti multipli, il sistema potrebbe richiedere una `request()` del dispositivo, al fine di assicurargne l'uso esclusivo. Dopo l'uso, ne avviene il rilascio (tramite `release()`). Si tratta di funzioni analoghe alle chiamate `open()` e `close()` per i file. Altri sistemi operativi permettono l'accesso incontrollato ai dispositivi, con il rischio che la contesa per l'accesso provochi lo stallo (Capitolo 8).

Una volta richiesto e assegnato il dispositivo, è possibile leggervi (`read()`), scrivervi (`write()`) ed eventualmente procedere a un riposizionamento (`reposition()`), esattamente come nei file; la somiglianza tra file e dispositivi di I/O è infatti tale che molti sistemi operativi, tra cui unix, li combinano in un'unica struttura file-dispositivi. In tal caso si usa lo stesso insieme di system call per file e dispositivi. A volte, i dispositivi di I/O sono identificati da nomi particolari, attributi speciali, o dal collocamento in certe directory.

L'interfaccia con l'utente può anche far apparire simili i file e i dispositivi, nonostante le chiamate di sistema sottostanti non lo siano: è un altro esempio di una delle scelte che il progettista deve intraprendere nella costruzione del sistema operativo e della relativa interfaccia utente.

2.3.3.4 Gestione delle informazioni

Molte chiamate di sistema hanno semplicemente lo scopo di trasferire le informazioni tra il programma utente e il sistema operativo. La maggior parte dei sistemi, per esempio, ha una chiamata di sistema per ottenere l'ora (`time()`) e la data attuali (`date()`). Altre chiamate di sistema possono restituire informazioni sul sistema, come il numero degli utenti collegati, il numero della versione del sistema operativo, la quantità di memoria disponibile o di spazio nei dischi, e così via.

Un altro insieme di chiamate di sistema è utile per il debugging di programmi. Molti sistemi operativi forniscono chiamate di sistema per ottenere un'immagine della memoria (effettuare il `dump()`). Questa funzionalità è utile per il debugging. Il programma `strace`, disponibile sui sistemi Linux, fornisce la sequenza delle chiamate di sistema eseguite. Anche i microprocessori offrono una modalità conosciuta come a singolo passo (*single step*) nella quale viene eseguita una *trap* dopo ogni istruzione. La *trap* viene solitamente catturata dal debugger.

Molti sistemi operativi possono effettuare un'analisi del tempo utilizzato da un programma e indicare la quantità di tempo in cui il programma rimane in esecuzione in una particolare locazione o in un insieme di locazioni. Un'analisi del tempo richiede un'utilità di tracciamento o delle interruzioni regolari da parte del timer. A ogni occorrenza di un'interruzione del timer il valore del contatore di programma viene memorizzato. Con una frequenza di interruzioni sufficientemente alta è possibile ottenere una statistica del tempo trascorso nelle varie parti di un programma.

Il sistema operativo contiene inoltre informazioni su tutti i propri processi; a queste informazioni si può accedere tramite alcune chiamate di sistema. In genere esistono anche chiamate di sistema per modificare le informazioni sui processi (`get_process_attributes()` e `set_process_attributes()`). Nel Paragrafo 3.1.3 si spiega quali sono tali informazioni.

2.3.3.5 Comunicazione

Esistono due modelli molto diffusi di comunicazione tra processi: il modello a scambio di messaggi e quello a memoria condivisa. Nel modello a scambio di messaggi i processi comunicanti si scambiano messaggi per il trasferimento delle informazioni sia direttamente sia indirettamente attraverso una casella di posta (*mailbox*) comune. Prima di effettuare una comunicazione occorre aprire un collegamento. Il nome dell'altro comunicante deve essere noto, sia che si tratti di un altro processo nello stesso calcolatore, sia di un processo in un altro calcolatore collegato attraverso una rete di comunicazione. Tutti i calcolatori di una rete hanno un nome di macchina (*host name*) con il quale sono individuati, e un identificativo di rete, per esempio un indirizzo ip. Analogamente, ogni processo ha un nome di processo, che si converte in un identificatore che il sistema operativo può impiegare per farvi riferimento. La conversione nell'identificatore si compie con le chiamate di sistema `get_hostid()` e `get_processid()`. Questi identificatori sono quindi passati alle chiamate di sistema d'uso generale `open()` e `close()` messe a disposizione dal file system oppure, a seconda del modello di comunicazione del sistema, alle specifiche chiamate di sistema `open_connection()` e `close_connection()`. Generalmente il processo ricevente deve acconsentire alla comunicazione con una chiamata di sistema `accept_connection()`. Nella maggior parte dei casi i processi che gestiscono la comunicazione sono demoni specifici, cioè programmi di sistema realizzati esplicitamente per questo scopo. Questi programmi eseguono una chiamata di sistema `wait_for_connection()` e sono chiamati in causa quando si stabilisce un collegamento. L'origine della comunicazione, nota come *client*, e il demone ricevente, noto come *server*, possono quindi scambiarsi i messaggi per mezzo delle chiamate di sistema `read_message()` e `write_message()`; la chiamata di sistema `close_connection()` pone fine alla comunicazione.

Nel modello a memoria condivisa, invece, i processi usano chiamate di sistema `shared_memory_create()` e `shared_memory_attach()` per creare e accedere alle aree di memoria possedute da altri processi. Occorre ricordare che, normalmente, il sistema operativo tenta di impedire a un processo l'accesso alla memoria di un altro processo. Il modello a memoria condivisa richiede che più processi concordino nel superare tale limite; a questo punto tali processi possono scambiarsi le informazioni leggendo e scrivendo i dati nelle aree di memoria condivise. Il formato e la posizione dei dati sono determinati esclusivamente da questi processi e non sono sotto il controllo del sistema operativo. I processi sono anche responsabili del rispetto della condizione di non scrivere contemporaneamente nella stessa posizione. Questi meccanismi sono discussi nel Capitolo 6. Nel Capitolo 4 si illustra anche una variante del modello di processo, detto *thread*, che prevede a priori la condivisione della memoria.

Entrambi i modelli sono assai comuni e in molti sistemi operativi sono presenti contemporaneamente. Lo scambio di messaggi è utile soprattutto quando è necessario trasferire una piccola quantità di dati poiché, in questo caso, non sussiste la necessità di evitare conflitti; è inoltre più facile da realizzare rispetto alla condivisione della memoria per la comunicazione tra calcolatori diversi. La condivisione della memoria permette la massima velocità e semplicità nelle comunicazioni, poiché queste ultime, se avvengono all'interno del calcolatore, si possono svolgere alla velocità della memoria. Vi sono, tuttavia, problemi per quel che riguarda la protezione e la sincronizzazione tra processi che condividono la memoria.

2.3.3.6 Protezione

La protezione fornisce un meccanismo per controllare l'accesso alle risorse di un calcolatore. Storicamente ci si preoccupava della protezione solo su calcolatori multiprogrammati e con numerosi utenti. Ora, con l'avvento delle reti e di Internet, tutti i calcolatori, dai server ai dispositivi mobili, devono tener conto della protezione.

Tra le chiamate di sistema che offrono meccanismi di protezione vi sono solitamente la `set_permission()` e la `get_permission()`, che permettono di modificare i permessi di accesso a risorse come file e dischi. Le chiamate di sistema `allow_user()` e `deny_user()` specificano se un particolare utente abbia il permesso di accesso a determinate risorse.

La protezione è trattata nel Capitolo 17. Il Capitolo 16 esamina la sicurezza in una prospettiva più ampia.

2.4 Servizi di sistema

Un'altra caratteristica importante di un sistema moderno è quella che riguarda l'insieme dei servizi di sistema. Facendo riferimento alla Figura 1.1, in cui s'illustra la gerarchia logica di un calcolatore, si può notare che il livello più basso è occupato dall'hardware. Seguono, nell'ordine, il sistema operativo, i programmi di sistema e i programmi applicativi. I servizi di sistema, detti anche utilità di sistema, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le chiamate di sistema, altri sono considerevolmente più complessi; in generale si possono classificare nelle seguenti categorie.

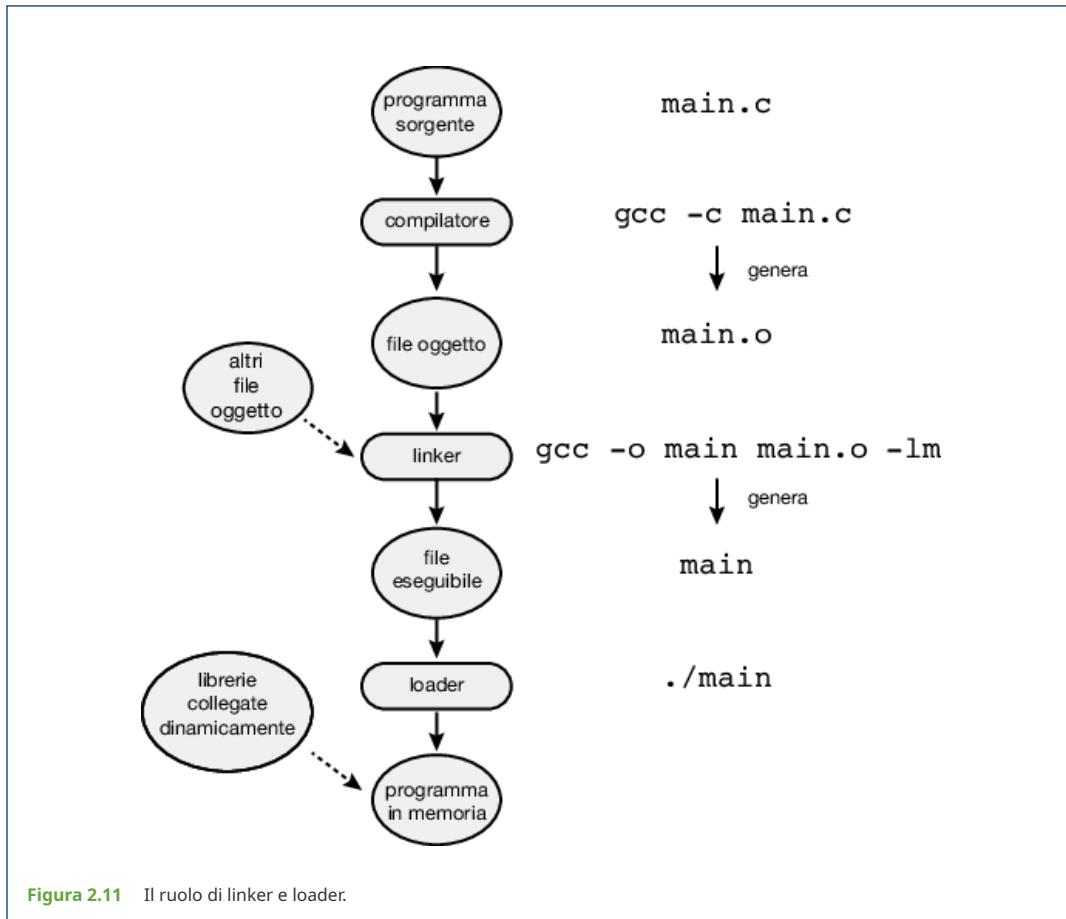
- Gestione dei file. Questi programmi creano, cancellano, copiano, rinominano, stampano, elencano e in genere compiono operazioni sui file e le directory.
- Informazioni di stato. Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti e simili informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni su terminale, o tramite altri dispositivi per l'uscita dei dati o, ancora, all'interno di una finestra della gui. Alcuni sistemi comprendono anche un registro (*registry*), al fine di archiviare e poter poi consultare informazioni sulla configurazione del sistema.
- Modifica dei file. Diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.
- Ambienti di supporto alla programmazione. Compilatori, assemblatori, debugger e interpreti dei comuni linguaggi di programmazione, come C, C++, Java e Python, sono spesso forniti insieme con il sistema operativo oppure disponibili per il download.
- Caricamento ed esecuzione dei programmi. Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*). Sono necessari anche i sistemi d'ausilio all'individuazione e correzione degli errori (*debugger*) per i linguaggi d'alto livello o per il linguaggio macchina.
- Comunicazioni. Questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi. Permettono agli utenti d'inviare messaggi agli schermi d'altri utenti, di consultare il Web, d'inviare messaggi di posta elettronica, di effettuare il login su calcolatori remoti, di trasferire file da un calcolatore a un altro.
- Servizi in background. Tutti i sistemi general-purpose hanno metodi per lanciare alcuni programmi di sistema al momento dell'avvio. Alcuni di questi processi terminano dopo aver completato i loro compiti, mentre altri restano in esecuzione fino a quando il sistema viene arrestato. I processi di sistema costantemente in esecuzione sono noti come servizi, sottosistemi oppure demoni. Un esempio è il demone di rete discusso nel Paragrafo 2.3.3.5. In tale esempio un sistema aveva bisogno di un servizio per rilevare le connessioni di rete e quindi assegnare le richieste ai processi corretti. Altri esempi includono le utilità di scheduling dei processi, che avviano i processi secondo una pianificazione specifica, i servizi di monitoraggio di errori di sistema e i server di stampa. Un sistema tipico ha decine di demoni. I sistemi operativi che eseguono attività importanti in contesto utente invece che in kernel possono utilizzare appositi demoni per eseguire queste attività.

Oltre ai programmi di sistema, con la maggior parte dei sistemi operativi sono forniti programmi che risolvono problemi comuni o che eseguono operazioni comuni. Questi programmi applicativi comprendono browser web, word processor, fogli di calcolo, sistemi di basi di dati, compilatori, programmi per analisi statistiche e visualizzazioni, oltre che videogiochi.

L'immagine che gli utenti si fanno di un sistema è influenzata principalmente dalle applicazioni e dai programmi di sistema, più che dalle chiamate di sistema. Quando un utente del sistema operativo macos usa la gui del sistema, si trova di fronte un insieme di finestre e il puntatore del mouse; quando, invece, usa la riga di comando, si trova di fronte a una shell in stile unix, magari in una delle finestre della gui. In entrambi i casi, l'insieme di chiamate di sistema sottostanti è lo stesso, ma l'aspetto e il modo d'operare del sistema sono ben diversi. Come esempio dell'ulteriore confusione che si può generare, si consideri un sistema in cui viene effettuato un dual boot da macos a Windows. In questo caso lo stesso utente sulla stessa macchina ha due differenti interfacce e due differenti insiemi di applicazioni che usano le stesse risorse fisiche. Con lo stesso hardware un utente può quindi utilizzare diverse interfacce, sequenzialmente o in modo concorrente.

2.5 Linker e loader

Generalmente un programma risiede su disco in forma di file binario eseguibile (per esempio, i file `a.out` o `prog.exe`). Per essere eseguito su una cpu, il programma deve essere caricato in memoria e inserito nel contesto di un processo. Descriviamo qui i passaggi di questa procedura, dalla compilazione di un programma al suo caricamento in memoria, dove diventa idoneo per l'esecuzione sul core di una cpu che sia disponibile. I passaggi sono evidenziati nella Figura 2.11.



I file sorgenti vengono compilati in file oggetto progettati per essere caricati in una qualsiasi posizione della memoria fisica, noti come file oggetto rilocabili. In seguito, il linker combina i file oggetto rilocabili in un singolo file binario eseguibile. Durante la fase di collegamento (linking) possono essere inclusi altri file oggetto o alcune librerie, come la libreria standard del linguaggio C o la libreria matematica standard (ciò avviene specificando il flag `-lm`).

Per caricare il file eseguibile in memoria, dove diventa idoneo per l'esecuzione sul core di una cpu, viene utilizzato un loader. Un'attività associata al collegamento e al caricamento (loading) è la rilocazione (relocation), che assegna gli indirizzi definitivi alle componenti del programma e riaggusta il codice e i dati nel programma secondo questi indirizzi in modo che, per esempio, il codice possa richiamare le funzioni della libreria e accedere alle sue variabili durante l'esecuzione. Nella Figura 2.11 possiamo osservare che per eseguire il loader è sufficiente scrivere il nome del file eseguibile sulla riga di comando. Quando si immette il nome di un programma sulla riga di comando dei sistemi unix (per esempio, digitando `./main`) la shell crea innanzitutto un nuovo processo per eseguire il programma, utilizzando la chiamata di sistema `fork()`, e richiama poi il loader con la chiamata di sistema `exec()`, passando alla `exec()` il nome del file eseguibile. Il loader carica quindi in memoria il programma specificato, utilizzando lo spazio d'indirizzamento del processo appena creato. Quando si utilizza un'interfaccia grafica, facendo doppio clic sull'icona associata al file eseguibile si richiama il loader, con un meccanismo simile.

Il processo descritto finora presuppone che tutte le librerie siano collegate nel file eseguibile e caricate in memoria. In realtà, la maggior parte dei sistemi consente a un programma di collegare dinamicamente le librerie quando viene caricato. Windows, per esempio, supporta librerie collegate dinamicamente (dll). Il vantaggio di questo approccio è che si evita di collegare e caricare librerie che potrebbero non essere utilizzate. In questo caso una libreria è collegata in modo condizionale e viene caricata in memoria solo se

è richiesta durante l'esecuzione del programma. Per esempio, nella Figura 2.11, la libreria matematica non è collegata al file eseguibile `main`; il linker inserisce piuttosto le informazioni di rilocazione che permettono alla libreria di essere collegata dinamicamente e caricata in memoria quando il programma è già caricato. Nel Capitolo 9 vedremo che più processi possono condividere librerie collegate dinamicamente, con un risparmio significativo nell'uso della memoria.

I file oggetto e i file eseguibili hanno in genere formati standard e includono il codice macchina compilato e una tabella dei simboli contenente metadati relativi a funzioni e variabili a cui si fa riferimento nel programma. Nei sistemi unix e Linux questo formato standard è noto come elf (Executable and Linkable Format). Vi sono formati elf distinti per file rilocabili ed eseguibili. Un'informazione presente nel file elf per i file eseguibili è il punto di ingresso (*entry point*) del programma, che contiene l'indirizzo della prima istruzione da eseguire. I sistemi Windows utilizzano il formato pe (Portable Executable), mentre macos usa il formato Mach-O.

2.6 Perché le applicazioni dipendono dal sistema operativo

Fondamentalmente le applicazioni compilate su un sistema operativo non sono eseguibili su altri sistemi operativi. Se lo fossero il mondo sarebbe un posto migliore e la nostra scelta di quale sistema operativo utilizzare dipenderebbe dalle utilità e dalle caratteristiche piuttosto che dalle applicazioni disponibili.

Basandoci sulla nostra analisi precedente, possiamo ora vedere una parte del problema: ogni sistema operativo fornisce un insieme univoco di chiamate di sistema. Le chiamate di sistema fanno parte dell'insieme di servizi forniti dai sistemi operativi alle applicazioni. Anche se le chiamate di sistema fossero in qualche modo uniformi, altri ostacoli renderebbero difficile l'esecuzione di programmi applicativi su diversi sistemi operativi. Se avete usato più di un sistema operativo potrebbe esservi capitato di usare la stessa applicazione su sistemi diversi. Com'è possibile?

Un'applicazione può essere resa disponibile per l'esecuzione su più sistemi operativi in tre modi.

1. L'applicazione può essere scritta in un linguaggio interpretato (come Python o Ruby) che ha un interprete disponibile per più sistemi operativi. L'interprete legge ogni riga del programma sorgente, esegue istruzioni equivalenti del set di istruzioni nativo e invoca le chiamate proprie del sistema operativo. Le prestazioni sono inferiori rispetto a quelle delle applicazioni native; inoltre, l'interprete fornisce solo un sottoinsieme delle funzionalità di ciascun sistema operativo, limitando potenzialmente l'insieme delle funzionalità delle applicazioni associate.
2. L'applicazione può essere scritta in un linguaggio che utilizza una macchina virtuale contenente l'applicazione in esecuzione. La macchina virtuale fa parte dell'rte completo del linguaggio. Un esempio di questo metodo è Java. Java ha un rte che include un loader, un verificatore di byte-code e altre componenti che permettono di caricare l'applicazione Java nella macchina virtuale Java. Questo rte è stato portato, o sviluppato ex-novo, su molti sistemi operativi, dai mainframe agli smartphone, e in teoria qualsiasi applicazione Java può essere eseguita all'interno di un rte, ovunque esso sia disponibile. Sistemi di questo tipo presentano svantaggi simili a quelli degli interpreti, esaminati precedentemente.
3. Lo sviluppatore di un'applicazione può utilizzare un linguaggio o un'api standard in cui il compilatore genera binari nel linguaggio specifico del sistema operativo e della macchina. L'applicazione deve essere portata su ciascun sistema operativo su cui verrà eseguita. Il porting può richiedere molto tempo e deve essere fatto per ogni nuova versione dell'applicazione, con successivi test e debug. L'esempio probabilmente più noto è dato dall'api posix e dal suo insieme di standard per il mantenimento della compatibilità del codice sorgente tra diverse varianti di sistemi operativi unix-like.

In teoria questi tre approcci sembrano fornire soluzioni semplici per lo sviluppo di applicazioni in grado di funzionare su diversi sistemi operativi. Tuttavia, la mancanza generale di mobilità delle applicazioni ha diverse cause, che tuttora rendono lo sviluppo di applicazioni multipiattaforma un compito impegnativo. A livello di applicazione, le librerie fornite nel sistema operativo contengono api per offrire funzionalità come interfacce gui e un'applicazione progettata per chiamare un set di api (per esempio, quelle rese disponibili da ios sull'iPhone della Apple) non funzionerà su un sistema operativo che non fornisce quelle api (come Android). Ai livelli più bassi del sistema si incontrano altri ostacoli, inclusi quelli di seguito descritti.

- Ogni sistema operativo ha un formato binario per le applicazioni che definisce il layout dell'intestazione, delle istruzioni e delle variabili. Questi elementi devono trovarsi in determinate posizioni all'interno di strutture specifiche nel file eseguibile, in modo che il sistema operativo possa aprire il file e caricare l'applicazione per una sua esecuzione corretta.
- Le cpu hanno set di istruzioni differenti e solo le applicazioni che utilizzano le istruzioni appropriate possono essere eseguite correttamente.
- I sistemi operativi forniscono chiamate di sistema che consentono alle applicazioni di richiedere vari servizi, come la creazione di file e l'apertura di connessioni di rete. Queste chiamate differiscono nei vari sistemi operativi sotto diversi aspetti, tra cui gli operandi utilizzati e il loro ordine, il modo in cui un'applicazione invoca le chiamate di sistema, la numerazione delle chiamate, il loro significato e i valori restituiti.

Alcuni approcci hanno aiutato a risolvere, anche se non completamente, queste differenze architettoniche. Per esempio, Linux e quasi tutti i sistemi unix hanno adottato il formato elf per i file binari eseguibili. Sebbene elf fornisca uno standard comune su sistemi Linux e unix, non è legato ad alcuna architettura specifica e non garantisce quindi che un file eseguibile possa essere eseguito su piattaforme hardware diverse.

Le api, come menzionato in precedenza, specificano determinate funzioni a livello di applicazione. A livello di architettura viene utilizzata un'abi (Application Binary Interface) per definire in che modo i diversi componenti di un codice binario possano interfacciarsi su un determinato sistema operativo e una determinata architettura. Un'abi specifica i dettagli di basso livello, tra cui la dimensione degli indirizzi, i metodi di passaggio dei parametri alle chiamate di sistema, l'organizzazione dello stack di runtime, il formato binario delle librerie di sistema e la dimensione dei tipi di dati, solo per citarne alcuni. In genere, viene specificata un'abi per ogni determinata architettura (per esempio, esiste un'abi per il processore armv8) e dunque un'abi è l'analogico a livello di architettura di un'api. Se un file binario eseguibile è stato compilato e collegato secondo una particolare abi, dovrebbe essere in grado di funzionare su sistemi diversi che supportino la stessa abi. Tuttavia, poiché una particolare abi è definita per un determinato sistema operativo in esecuzione su una determinata architettura, le abi possono fare ben poco per fornire compatibilità multipiattaforma.

In breve, tutte queste differenze significano che un'applicazione non riuscirà a funzionare a meno che un interprete, un rte o il file binario eseguibile siano scritti e compilati su un sistema operativo specifico e su un determinato tipo di cpu (per esempio Intel x86 o armv8). Provate a immaginare dunque la quantità di lavoro necessaria per fare in modo che un programma come il browser Firefox possa essere eseguito su Windows, macos, varie versioni Linux, ios e Android, a volte su differenti architetture hardware.

2.7 Progettazione e realizzazione di un sistema operativo

Nei seguenti paragrafi si trattano i problemi riguardanti la progettazione e la realizzazione di un sistema. Naturalmente non si dispone di soluzioni complete, ma vari approcci si sono dimostrati efficaci.

2.7.1 Scopi della progettazione

Il primo problema che s'incontra nella progettazione di un sistema riguarda la definizione degli obiettivi e delle specifiche del sistema stesso. Al più alto livello, la progettazione del sistema è influenzata in modo decisivo dalla scelta dell'architettura fisica e del tipo di sistema: desktop e laptop tradizionali, sistemi mobili, distribuiti o real-time.

D'altra parte, oltre questo livello di progettazione, i requisiti possono essere molto difficili da specificare, anche se, in generale, si possono distinguere in due gruppi fondamentali: *obiettivi degli utenti* e *obiettivi del sistema*.

Gli utenti desiderano che un sistema abbia alcune caratteristiche ovvie: deve essere utile, facile da imparare e usare, affidabile, sicuro e veloce; queste specifiche non sono particolarmente utili nella progettazione di un sistema, poiché non tutti concordano sui metodi da applicare per raggiungere questi scopi.

Requisiti analoghi sono richiesti da chi deve progettare, creare e operare con il sistema: il sistema operativo deve essere di facile progettazione, realizzazione e manutenzione; deve essere flessibile, affidabile, senza errori ed efficiente. Anche in questo caso si tratta di requisiti vaghi, interpretabili in vari modi.

Non esiste una soluzione unica al problema della definizione dei requisiti di un sistema operativo. L'ampia gamma di sistemi mostra che da requisiti diversi possono risultare le soluzioni più varie per ambienti diversi. Per esempio, i requisiti di Wind River VxWorks, un sistema operativo real-time per sistemi embedded, erano assai diversi da Windows Server, un sistema operativo multiaccesso di grandi dimensioni per applicazioni aziendali.

Specificare e progettare un sistema operativo richiede creatività. Per quanto nessun libro possa indicare precisamente come fare, sono stati sviluppati alcuni principi generali nel campo del software engineering che ora discuteremo.

2.7.2 Meccanismi e politiche

Un principio molto importante è quello che riguarda la distinzione tra meccanismi e criteri o politiche (*policy*). I meccanismi determinano *come* eseguire qualcosa; i criteri, invece, stabiliscono *che cosa* si debba fare. Il timer del sistema (Paragrafo 1.4.3), per esempio, è un meccanismo che assicura la protezione della cpu, ma la decisione riguardante la quantità di tempo da impostare nel timer per un utente specifico riguarda le politiche.

La distinzione tra meccanismi e politiche è molto importante ai fini della flessibilità. Le politiche sono soggette a cambiamenti di luogo o di tempo. Nei casi peggiori il cambiamento di una politica può richiedere il cambiamento del meccanismo sottostante. Sarebbe preferibile disporre di meccanismi generali: in questo caso un cambiamento di politica implicherebbe solo la ridefinizione di alcuni parametri del sistema. Per esempio, consideriamo un meccanismo che assegna priorità a certe categorie di programmi rispetto ad altre. Se tale meccanismo è debitamente separato dalla politica con cui si procede, esso è utilizzabile per far sì che programmi che impiegano intensamente operazioni di i/o abbiano priorità su quelli che impiegano intensamente la cpu, oppure viceversa.

I sistemi operativi basati su microkernel (Paragrafo 2.8.3) portano alle estreme conseguenze la separazione dei meccanismi dalle politiche, fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi completamente indipendenti dalle politiche, consentono l'aggiunta di meccanismi e criteri più complessi tramite moduli del kernel creati dagli utenti o anche tramite programmi utente. Si consideri invece Windows, un sistema operativo commerciale molto popolare e disponibile da oltre trent'anni. Microsoft ha codificato nel sistema sia i meccanismi sia le politiche in modo da forzare caratteristiche percepite dall'utente comuni su tutti i dispositivi che eseguono il sistema operativo Windows. Tutte le applicazioni hanno interfacce simili, poiché l'interfaccia stessa è integrata nel kernel e nelle librerie di sistema. Apple ha adottato una strategia simile con i suoi sistemi operativi macos e ios.

Possiamo fare un confronto analogo tra sistemi operativi commerciali e sistemi operativi open-source. Per esempio, possiamo confrontare Windows, trattato in precedenza, con Linux, un sistema operativo open-source utilizzato su una vasta gamma di dispositivi e disponibile da oltre 25 anni. Il kernel "standard" di Linux ha uno specifico algoritmo di schedulazione della cpu (trattato nel Paragrafo 5.7.1) che costituisce un meccanismo che supporta determinate politiche. Tuttavia, chiunque è libero di modificare o sostituire lo scheduler per supportare politiche diverse.

Le decisioni relative alle politiche sono importanti per tutti i problemi di assegnazione delle risorse. Invece, ogni volta che un problema riguarda il *come* piuttosto che il *che cosa* occorre definire un meccanismo.

2.7.3 Realizzazione

Una volta progettato, un sistema operativo va realizzato. Poiché i sistemi operativi sono collezioni di molti programmi scritti da molte persone in un lungo periodo di tempo, è difficile fare affermazioni generali su come essi vengano implementati.

I primi sistemi operativi erano scritti in linguaggio assembly. Oggi la maggior parte è scritta in linguaggi di alto livello come C o C++, con piccole porzioni di codice scritte in linguaggio assembly. Di fatto, viene spesso utilizzata una combinazione di diversi linguaggi di alto livello: i livelli più bassi del kernel potrebbero essere scritti in assembly e in C, le routine di livello superiore potrebbero essere scritte in C e C++ e le librerie di sistema in C++ o anche in linguaggi di più alto livello. Android fornisce un buon esempio: il suo kernel è scritto principalmente in C, con piccole porzioni in linguaggio assembly, la maggior parte delle librerie di

sistema è scritta in C o C++ e i suoi ambienti applicativi, che forniscono l'interfaccia al sistema per gli sviluppatori, sono scritti principalmente in Java. Tratteremo l'architettura di Android in modo più dettagliato nel Paragrafo 2.8.5.2.

I vantaggi derivanti dall'uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, per implementare un sistema operativo, sono gli stessi che si ottengono quando il linguaggio si usa per i programmi applicativi: il codice si scrive più rapidamente, è più compatto ed è più facile da capire e mettere a punto. Inoltre il perfezionamento delle tecniche di compilazione consente di migliorare il codice generato per l'intero sistema operativo con una semplice ricompilazione. Infine, un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un'altra architettura (*porting*). Ciò è particolarmente importante nei sistemi operativi destinati a funzionare su diverse piattaforme hardware, come piccoli dispositivi integrati, sistemi Intel x86 e chip arm montati su telefoni e tablet.

I soli eventuali svantaggi che possono presentarsi nella realizzazione di un sistema operativo in un linguaggio di alto livello sono una minore velocità d'esecuzione e una maggiore occupazione di spazio di memoria, una questione ormai superata nei sistemi moderni. D'altra parte, benché un programmatore esperto di un linguaggio assembly possa produrre piccole procedure di grande efficienza, per quel che riguarda i programmi molto estesi, un moderno compilatore può eseguire complesse analisi e applicare raffinate ottimizzazioni che producono un codice eccellente. Le moderne cpu sono organizzate con vari blocchi d'elaborazione concatenati (*pipelining*) e parecchie unità funzionali, le cui complesse interdipendenze sfuggono facilmente alla limitata capacità della mente umana.

Come in altri sistemi, i miglioramenti principali nelle prestazioni dei sistemi operativi sono dovuti più a strutture dati e algoritmi migliori che a un'ottima codifica in linguaggio assembly. Inoltre, sebbene i sistemi operativi siano molto grandi, solo una piccola parte del codice assume un'importanza critica per le prestazioni: il gestore degli interrupt, il gestore dell'i/o, il gestore della memoria e lo scheduler della cpu sono probabilmente le procedure più critiche. Una volta scritto il sistema e verificato il suo corretto funzionamento, è possibile identificare i colli di bottiglia e riprogettare quelle parti, in modo da ottenere una maggior efficienza.

2.8 Struttura del sistema operativo

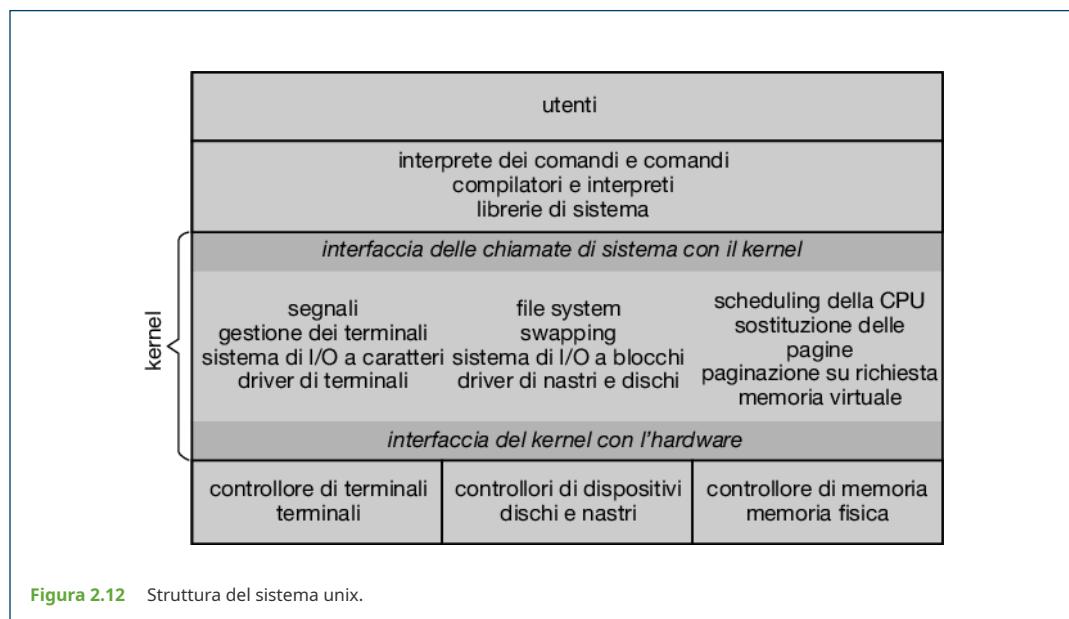
Affinché possa funzionare correttamente ed essere facilmente modificato, un sistema vasto e complesso come un sistema operativo moderno va progettato con estrema attenzione. Anziché progettare un sistema come un unico blocco, un orientamento diffuso è di suddividerlo in piccoli componenti, detti *moduli*; ciascun modulo deve costituire una porzione ben definita del sistema, con interfacce e funzioni definite con precisione. È possibile utilizzare un approccio simile quando si strutturano i programmi: invece di inserire tutto il codice nella funzione `main()`, si suddivide la logica del programma in un certo numero di funzioni, articolando in maniera chiara i parametri e i valori restituiti e invocando poi queste funzioni dal `main()`.

Abbiamo trattato brevemente i componenti comuni dei sistemi operativi nel Capitolo 1. In questo paragrafo vedremo come questi componenti siano interconnessi e combinati in un kernel.

2.8.1 Struttura monolitica

La struttura più semplice per l'organizzazione di un sistema operativo è l'assenza di struttura: tutte le funzionalità del kernel vengono inserite in un singolo file binario statico che viene eseguito in un unico spazio d'indirizzamento. Questo approccio, noto come struttura monolitica, è una tecnica comune di progettazione dei sistemi operativi.

Un esempio di questa strutturazione limitata è il sistema operativo unix originale, che consiste di due parti separate: il kernel e i programmi di sistema. Il kernel è ulteriormente suddiviso in una serie di interfacce e driver dei dispositivi, aggiunti e ampliati nel corso dell'evoluzione di unix. Possiamo vedere unix come un sistema parzialmente stratificato, come mostrato nella Figura 2.12. Sotto l'interfaccia alle chiamate di sistema e sopra l'hardware si trova il kernel, che fornisce il file system, lo scheduling della cpu, la gestione della memoria e altre funzionalità del sistema operativo attraverso le chiamate di sistema: si tratta di un'enorme quantità di funzionalità diverse combinate in un unico spazio d'indirizzamento.



Il sistema operativo Linux è basato su unix ed è strutturato in modo simile, come mostrato nella Figura 2.13. Le applicazioni utilizzano in genere la libreria standard del linguaggio C, `glibc`, quando comunicano con il kernel mediante l'interfaccia alle chiamate di sistema. Il kernel Linux è monolitico, in quanto viene eseguito interamente in modalità kernel in un unico spazio d'indirizzamento ma, come vedremo nel Paragrafo 2.8.4, è dotato di una struttura modulare che consente di modificare il kernel durante l'esecuzione.

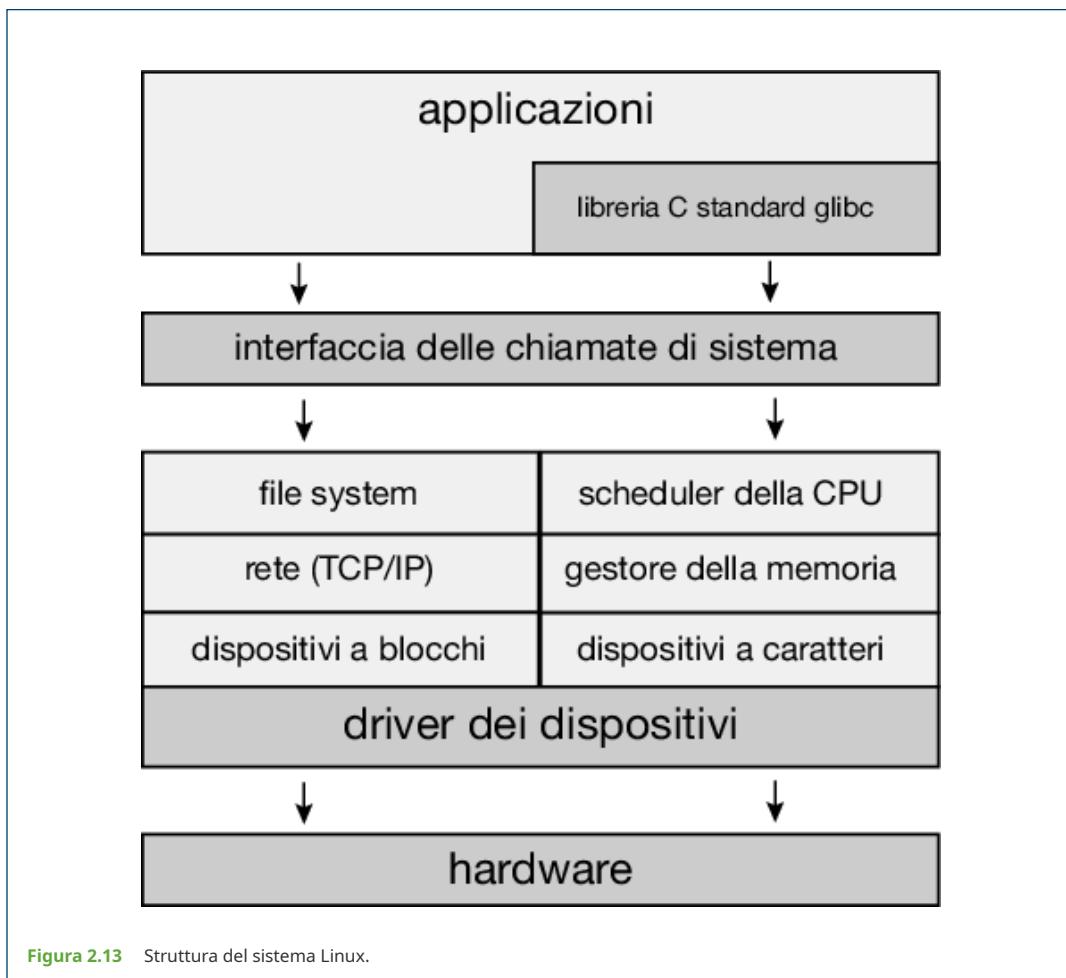


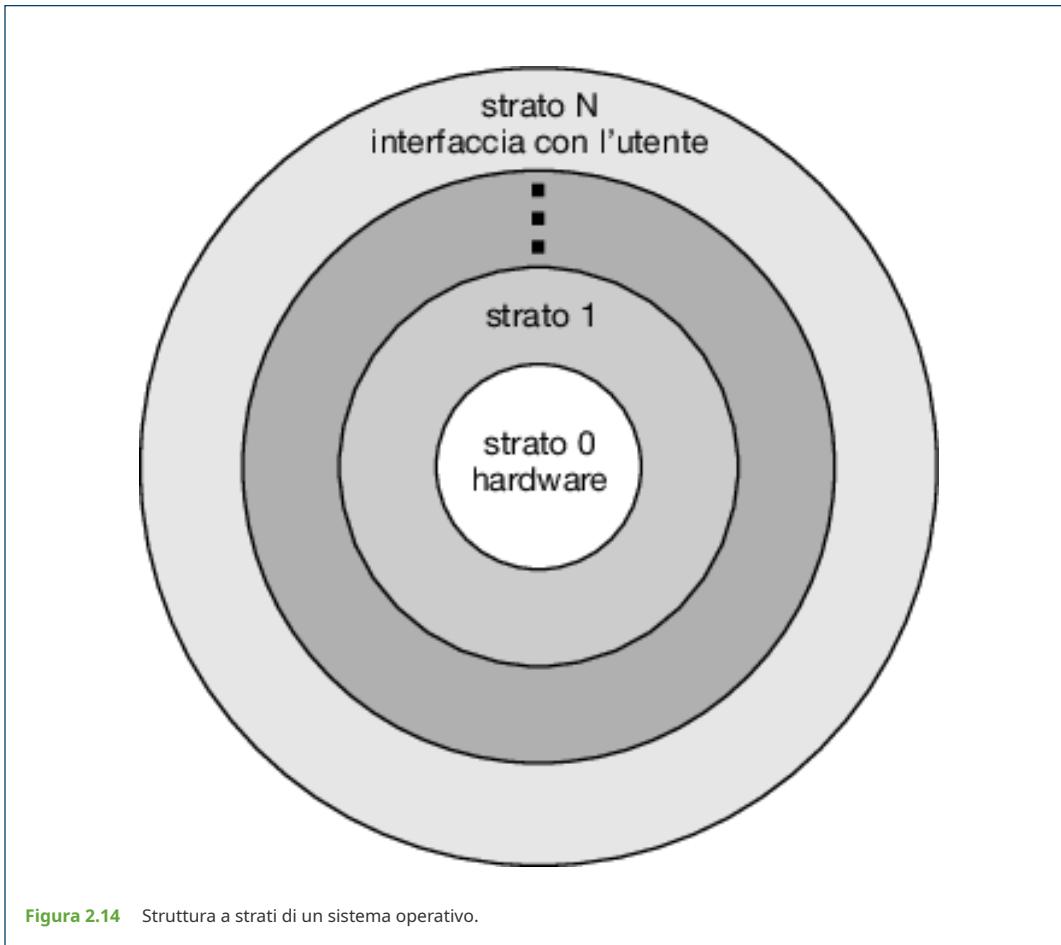
Figura 2.13 Struttura del sistema Linux.

Nonostante l'apparente semplicità dei kernel monolitici, essi sono difficili da implementare ed estendere. I kernel monolitici hanno però un netto vantaggio in termini di prestazioni: l'interfaccia alle chiamate di sistema presenta un overhead molto ridotto e la comunicazione all'interno del kernel è veloce. Pertanto, nonostante gli svantaggi dei kernel monolitici, la loro velocità e la loro efficienza giustificano la presenza di elementi di una struttura di questo tipo nei sistemi operativi unix, Linux e Windows.

2.8.2 Approccio stratificato

Un sistema monolitico viene anche chiamato sistema strettamente accoppiato (*tightly coupled*), perché le modifiche a una parte del sistema possono avere effetti di ampia portata su altre parti. In alternativa è possibile progettare un sistema debolmente accoppiato (*loosely coupled*), suddiviso in componenti separati più piccoli con funzionalità specifiche e limitate. Tutti questi componenti costituiscono nel loro insieme il kernel. Il vantaggio di questo approccio modulare è che i cambiamenti in un componente influiscono solo su quel componente e su nessun altro, consentendo a chi implementa il sistema una maggiore libertà nella creazione e modifica dei meccanismi interni.

Vi sono molti modi per rendere modulare un sistema operativo. Uno di essi è l'approccio stratificato, secondo il quale il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N). Si veda la Figura 2.14.



Lo strato di un sistema operativo è la realizzazione di un oggetto astratto, che incapsula i dati e le operazioni che trattano tali dati. Un tipico strato di sistema operativo (chiamiamolo M) è composto da strutture dati e da un insieme di routine richiamabili dagli strati di livello più alto. Lo strato M , a sua volta, è in grado di invocare operazioni degli strati di livello inferiore.

Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e di debugging. Gli strati sono composti in modo che ciascuno usi solo funzioni (operazioni) e servizi appartenenti a strati di livello inferiore. Questo approccio semplifica il debugging e la verifica del sistema. Il primo strato si può mettere a punto senza intaccare il resto del sistema, poiché per realizzare le proprie funzioni usa, per definizione, solo lo strato hardware, che si presuppone sia corretto. Passando al secondo strato si presume, dopo la messa a punto, la correttezza del primo. Il procedimento si ripete per ogni strato. Se si riscontra un errore, questo deve trovarsi in quello strato, poiché gli strati inferiori sono già stati corretti; quindi la suddivisione in strati semplifica la progettazione e la realizzazione di un sistema.

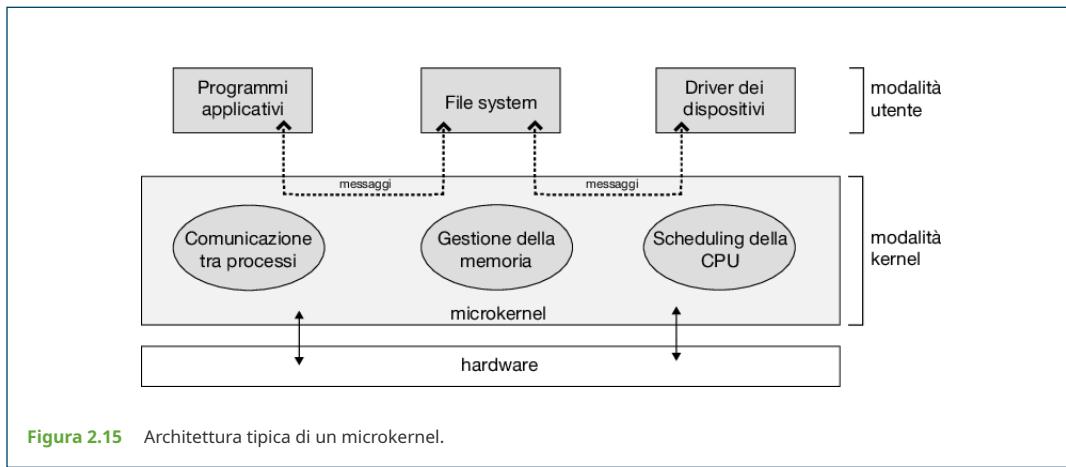
Ogni strato si realizza impiegando unicamente le operazioni messe a disposizione dagli strati inferiori, considerando soltanto le azioni che compiono, senza entrare nel merito di come queste sono realizzate. Di conseguenza ogni strato nasconde a quelli superiori l'esistenza di determinate strutture dati, operazioni e hardware.

I sistemi stratificati sono stati utilizzati con successo nelle reti di computer (come tcp/ip) e in applicazioni web. Tuttavia, relativamente pochi sistemi operativi utilizzano un approccio a strati puro. Una causa di questo scarso utilizzo risiede nella difficoltà di definire in modo appropriato le funzionalità di ogni strato. Inoltre, le prestazioni complessive di questi sistemi risultano scarse a causa dell'overhead dovuto al fatto che la richiesta di un servizio del sistema operativo da parte di un programma utente deve attraversare più strati. Tuttavia, una parziale stratificazione della struttura è comune nei sistemi operativi contemporanei. Generalmente questi sistemi hanno un numero inferiore di strati, ognuno con più funzioni, e offrono la maggior parte dei vantaggi del codice modulare evitando i difficili problemi connessi alla definizione e all'interazione degli strati.

2.8.3 Microkernel

Abbiamo già visto che il sistema unix originale aveva una struttura monolitica. A mano a mano che il sistema operativo unix è stato esteso, il kernel è cresciuto notevolmente, diventando sempre più difficile da gestire. Verso la metà degli anni '80 un gruppo di ricercatori della Carnegie Mellon University progettò e realizzò un sistema operativo, Mach, con il kernel strutturato in moduli secondo il cosiddetto orientamento a microkernel. Seguendo questo orientamento si progetta il sistema operativo rimuovendo dal

kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema. Ne risulta un kernel di dimensioni assai inferiori. Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione. La Figura 2.15 mostra l'architettura tipica di un microkernel.



Lo scopo principale del microkernel è fornire funzioni di comunicazione tra i programmi client e i vari servizi, anch'essi in esecuzione nello spazio utente. La comunicazione viene realizzata mediante scambio di messaggi, come descritto nel Paragrafo 2.3.3.5. Per accedere a un file, per esempio, un programma client deve interagire con il file server; ciò non avviene mai in modo diretto, ma tramite uno scambio di messaggi con il microkernel.

Uno dei vantaggi del microkernel è la facilità di estensione del sistema operativo: i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel. Poiché è ridotto all'essenziale, se il kernel deve essere modificato, i cambiamenti da fare sono ridotti e il sistema operativo risultante è più semplice da portare su diverse architetture. Inoltre offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto.

L'esempio più noto di un sistema operativo microkernel è probabilmente Darwin, il componente kernel dei sistemi operativi macos e ios. Darwin, in effetti, consiste di due kernel, uno dei quali è il Mach microkernel. Tratteremo più dettagliatamente i sistemi macos e ios nel Paragrafo 2.8.5.1.

Un altro esempio è costituito da qnx, un sistema operativo real-time per sistemi embedded basato sul microkernel qnx Neutrino che fornisce i servizi di scheduling e di consegna dei messaggi, oltre a gestire le interruzioni e la comunicazione di rete a basso livello. Tutti gli altri servizi necessari sono forniti da processi ordinari eseguiti al di fuori del kernel in modalità utente.

Purtroppo i microkernel possono incorrere in cali di prestazioni dovuti all'aumento dell'overhead. Quando due servizi di livello utente devono comunicare, i messaggi devono essere copiati tra i servizi, che risiedono in spazi d'indirizzamento separati. Inoltre, il sistema operativo potrebbe dover commutare da un processo all'altro per poter effettuare lo scambio di messaggi. L'overhead dovuto alla copia dei messaggi e alla commutazione tra i processi è stato il principale ostacolo alla crescita dei sistemi operativi basati su microkernel. Consideriamo la storia di Windows nt: la prima versione era basata su un microkernel stratificato e le sue prestazioni erano inferiori a quelle di Windows 95. Windows nt 4.0 risolse parzialmente il problema delle prestazioni spostando strati dallo spazio utente allo spazio kernel e integrandoli più strettamente. Al tempo della progettazione di Windows xp, l'architettura di Windows era ormai più monolitica che microkernel. Il Paragrafo 2.8.5.1 illustrerà come macos abbia risolto i problemi di prestazioni del microkernel Mach.

2.8.4 Moduli

Forse il miglior approccio attualmente disponibile per la progettazione dei sistemi operativi si basa sull'utilizzo di moduli del kernel caricabili dinamicamente. In questo contesto, il kernel è costituito da un insieme di componenti di base, integrati poi da funzionalità aggiunte dinamicamente durante l'avvio o l'esecuzione per mezzo di moduli. Questa strategia è comune nelle implementazioni moderne di unix, come Linux, macos e Solaris, come anche in Windows.

L'idea di base è che il kernel debba fornire direttamente i servizi principali, mentre gli altri servizi sono implementati in modo dinamico, quando il kernel è in esecuzione. Il collegamento dinamico dei servizi è preferibile all'aggiunta di nuove funzionalità direttamente nel kernel, che richiederebbe di ricompilare il kernel ogni volta che si effettua un cambiamento. Così, per esempio, si potrebbero includere lo scheduling della cpu e gli algoritmi di gestione della memoria direttamente nel kernel e aggiungere il supporto per diversi file system attraverso moduli caricabili.

Il risultato complessivo ricorda un sistema stratificato, in quanto ogni sezione del kernel ha interfacce protette ben definite, ma è più flessibile di un sistema a strati, perché ogni modulo può chiamare qualsiasi altro modulo. L'approccio è anche simile a quello basato su microkernel per il fatto che il modulo principale ha solo le funzioni essenziali e la conoscenza di come caricare altri moduli e

comunicare con essi, ma è più efficiente, poiché i moduli non devono invocare la funzionalità di trasmissione di messaggi per comunicare.

Linux utilizza moduli del kernel caricabili (lkm, *loadable kernel module*), principalmente per supportare i driver dei dispositivi e i file system. I lkm possono essere "inseriti" nel kernel quando il sistema viene avviato o durante la sua esecuzione, per esempio quando un dispositivo usb viene collegato a una macchina accesa. Se il kernel di Linux non ha il driver necessario, questo può essere caricato dinamicamente. I lkm possono anche essere rimossi dal kernel durante l'esecuzione. Nel caso di Linux, i lkm consentono di avere un kernel dinamico e modulare, pur mantenendo i vantaggi prestazionali di un sistema monolitico.

2.8.5 Sistemi ibridi

Nella pratica pochi sistemi operativi adottano un'unica struttura ben definita. I sistemi operativi, piuttosto, combinano strutture diverse, che portano a sistemi ibridi indirizzati alle prestazioni, alla sicurezza e all'usabilità. Per esempio, sia Linux sia Solaris sono monolitici, perché avere il sistema operativo in un unico spazio di indirizzamento garantisce prestazioni migliori, ma sono anche modulari, quindi una nuova funzionalità può essere aggiunta dinamicamente al kernel. Windows è in gran parte monolitico (ancora una volta principalmente per questioni di prestazioni), ma conserva alcuni comportamenti tipici dei sistemi microkernel, tra cui il supporto per sottosistemi separati (conosciuti come personalità del sistema operativo) che vengono eseguiti come processi in modalità utente. I sistemi Windows forniscono anche il supporto per i moduli del kernel caricabili dinamicamente. Forniamo casi di studio di Linux e di Windows 10 nei Capitoli 20 e 21, rispettivamente. Nel resto di questo paragrafo esploriamo la struttura di tre sistemi ibridi: Apple macos e i due principali sistemi operativi per dispositivi mobili, ios e Android.

2.8.5.1 macOS e iOS

Il sistema operativo macos di Apple è progettato per funzionare principalmente su computer desktop e laptop, mentre ios è un sistema operativo mobile progettato per iPhone e iPad. A livello di architettura macos e ios hanno molto in comune e quindi li presentiamo insieme, evidenziando ciò che condividono e gli aspetti in cui differiscono l'uno dall'altro. L'architettura generale di questi due sistemi è mostrata nella Figura 2.16. Tra i vari strati, evidenziamo i seguenti.

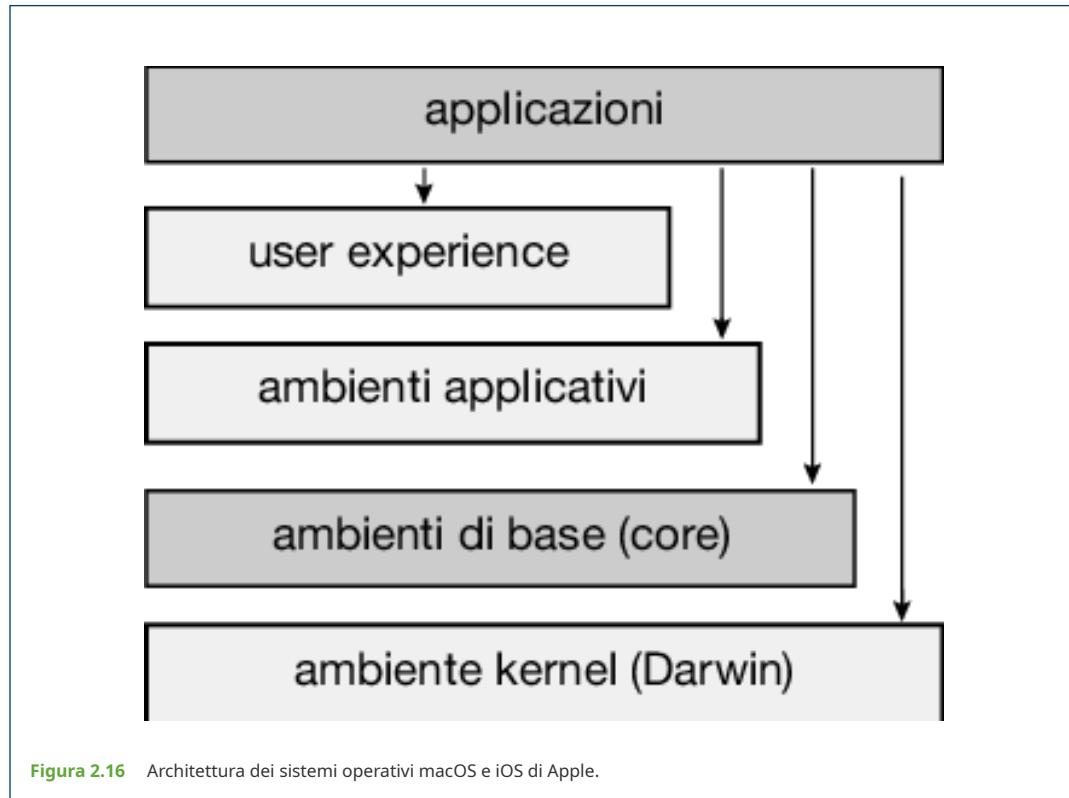


Figura 2.16 Architettura dei sistemi operativi macOS e iOS di Apple.

- Strato dell'interfaccia utente (user experience). Questo strato definisce l'interfaccia software che consente agli utenti di interagire con i dispositivi informatici. Il sistema macos usa l'interfaccia utente Aqua, progettata per un uso con mouse o trackpad, mentre ios usa l'interfaccia utente Springboard, progettata per dispositivi touch.
- Strato degli ambienti applicativi. Questo strato include gli ambienti Cocoa e Cocoa Touch, che forniscono un'api per i linguaggi di programmazione Objective-C e Swift. La differenza principale tra Cocoa e Cocoa Touch è che il primo viene utilizzato per

lo sviluppo di applicazioni macos, mentre il secondo viene utilizzato da ios per fornire supporto a funzionalità hardware esclusive per dispositivi mobili, come i touch-screen.

- Ambienti di base (core). Questo strato definisce gli ambienti che supportano la grafica e i contenuti multimediali, inclusi Quicktime e OpenGL.
- Ambiente kernel. Questo ambiente, noto anche come Darwin, include il microkernel Mach e il kernel bsd unix. Parleremo di Darwin a breve.

Come mostrato nella Figura 2.16, le applicazioni possono essere progettate per sfruttare le funzionalità di *user experience* o per aggirarle e interagire direttamente con l'ambiente applicativo o l'ambiente di base. Inoltre, un'applicazione può rinunciare completamente a questi ambienti e comunicare direttamente con il kernel. (Un esempio di quest'ultima situazione è un programma C senza interfaccia utente che effettua chiamate di sistema posix).

Elenchiamo di seguito alcune differenze significative tra macos e ios.

- Poiché macos è destinato ai sistemi desktop e laptop, è compilato per funzionare su architetture Intel. ios è invece progettato per dispositivi mobili ed è quindi compilato per architetture basate su arm. Analogamente, il kernel ios è stato leggermente modificato per rispondere a caratteristiche e necessità specifiche dei sistemi mobili, come la gestione energetica e una gestione molto ottimizzata della memoria. Inoltre, ios ha impostazioni di sicurezza più severe rispetto a macos.
- Il sistema operativo ios è generalmente molto più restrittivo rispetto a macos nei confronti degli sviluppatori, fino a essere chiuso agli sviluppatori in alcuni casi. Per esempio, ios limita l'accesso alle api posix e bsd, mentre le stesse sono liberamente disponibili per gli sviluppatori su macos.

Ci concentriamo ora su Darwin, che utilizza una struttura ibrida. Darwin è un sistema a strati costituito principalmente dal microkernel Mach e dal kernel bsd unix. La struttura di Darwin è mostrata nella Figura 2.17.

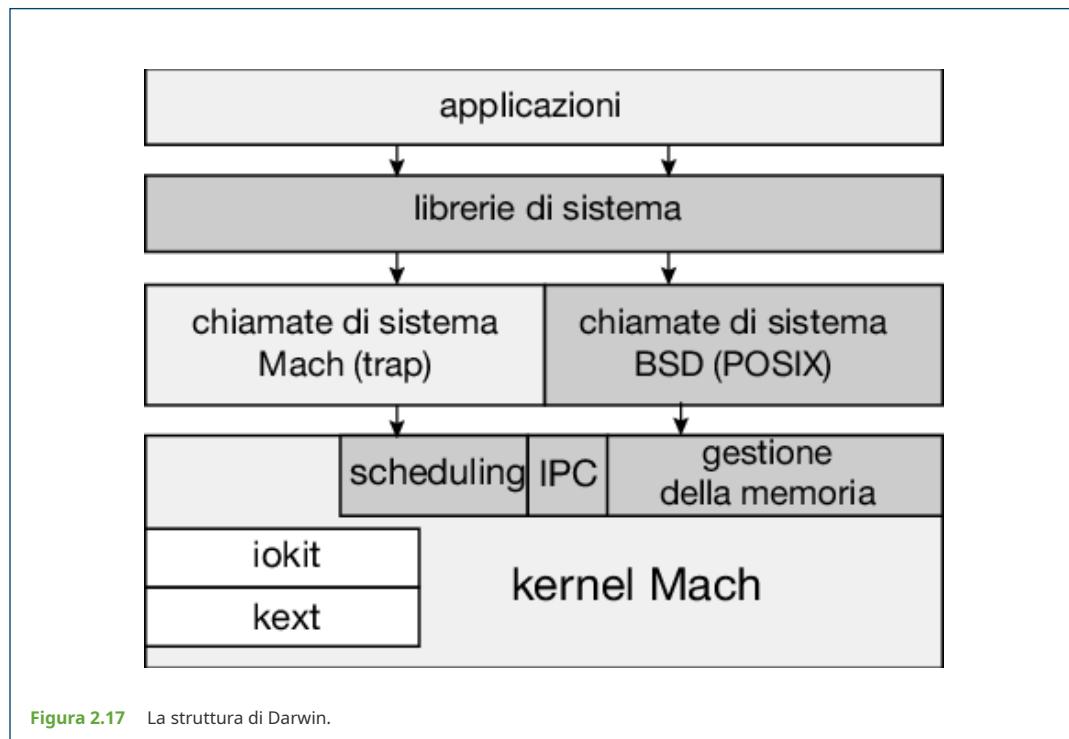


Figura 2.17 La struttura di Darwin.

Mentre la maggior parte dei sistemi operativi fornisce un'unica interfaccia alle chiamate di sistema, per esempio attraverso la libreria standard del linguaggio C su sistemi unix e Linux, Darwin fornisce due interfacce alle chiamate di sistema: chiamate di sistema Mach (dette *trap*) e chiamate di sistema bsd (che forniscono funzionalità posix). L'interfaccia a queste chiamate è un ricco insieme di librerie che include non solo la libreria standard del C, ma anche librerie per il supporto delle reti, della sicurezza e dei linguaggi di programmazione (solo per citarne alcune).

Oltre all'interfaccia alle chiamate di sistema, Mach fornisce servizi di sistema fondamentali, tra cui la gestione della memoria, lo scheduling della cpu e servizi di comunicazione fra processi (ipc) quali il trasferimento di messaggi e le chiamate di procedure remote (rpc). Gran parte delle funzionalità fornite da Mach è resa disponibile attraverso le astrazioni del kernel, che includono i task (i processi Mach), i thread, gli oggetti di memoria e le porte (utilizzate per ipc). Per esempio, un'applicazione può creare un nuovo processo utilizzando la chiamata di sistema bsd `fork()`. Mach, a sua volta, utilizzerà l'astrazione task del kernel per rappresentare il processo nel kernel.

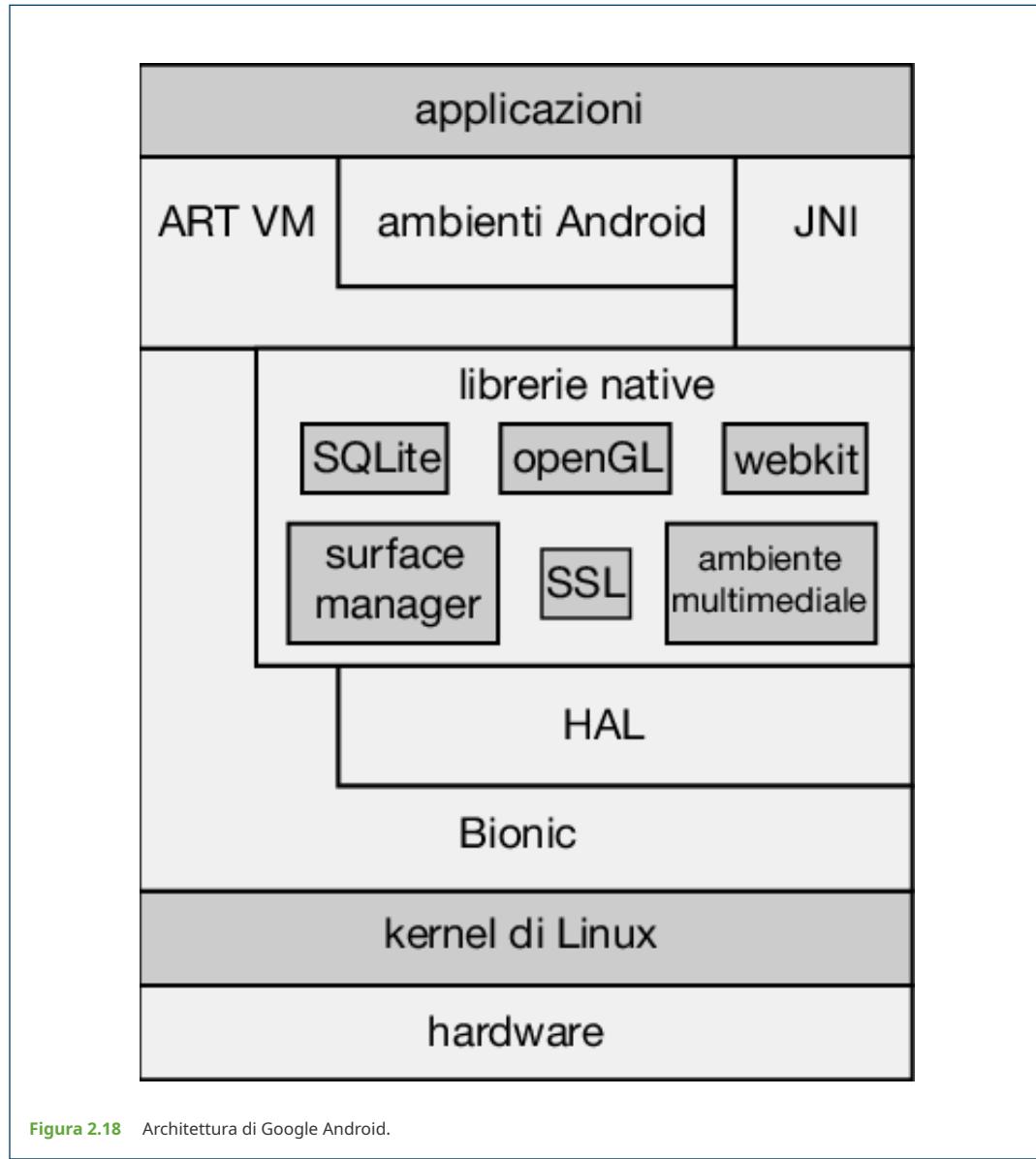
Oltre a Mach e bsd, l'ambiente kernel fornisce un kit i/o per lo sviluppo di driver dei dispositivi e moduli caricabili dinamicamente (chiamati in macos estensioni del kernel o kext).

Nel Paragrafo 2.8.3 abbiamo visto come l'overhead dello scambio di messaggi tra diversi servizi in esecuzione nello spazio utente comprometta le prestazioni dei microkernel. Per risolvere questi problemi di prestazioni, Darwin combina Mach, bsd, il kit i/o e qualsiasi estensione del kernel in un unico spazio d'indirizzamento. Mach non è quindi un microkernel puro, nel senso che diversi sottosistemi sono in esecuzione nello stesso spazio utente. All'interno di Mach si può ancora avere uno scambio di messaggi, ma non è necessaria alcuna copia, perché i servizi hanno accesso allo stesso spazio d'indirizzamento.

Apple ha rilasciato il sistema operativo Darwin come open-source. Di conseguenza, vari progetti hanno aggiunto ulteriori funzionalità a Darwin, come il sistema a finestre X-11 e il supporto per file system diversi. A differenza di Darwin, tuttavia, l'interfaccia Cocoa e altri ambienti Apple proprietari disponibili per lo sviluppo di applicazioni macos sono chiusi.

2.8.5.2 Android

Il sistema operativo Android è stato progettato dalla Open Handset Alliance (guidata principalmente da Google) e sviluppato per gli smartphone e i tablet Android. Mentre ios è progettato per funzionare su dispositivi mobili di Apple ed è un software proprietario, Android gira su una varietà di piattaforme mobili ed è open-source; questo spiega in parte la sua rapida ascesa in popolarità. La struttura di Android è illustrata nella Figura 2.18.



Android è simile a ios, in quanto è una pila di strati software che fornisce un ricco insieme di ambienti che supportano la grafica, l'audio e le funzionalità hardware. Queste funzionalità, a loro volta, forniscono una piattaforma per lo sviluppo di applicazioni mobili che funzionano su una moltitudine di dispositivi Android.

I progettisti di software per dispositivi Android sviluppano applicazioni in linguaggio Java, ma generalmente al posto dell'api Java standard utilizzano un'api Android progettata da Google per lo sviluppo Java. Le applicazioni Java sono compilate in modo da poter essere eseguite su Android RunTime art, una macchina virtuale progettata per Android e ottimizzata per dispositivi mobili con memoria limitata e ridotta capacità di elaborazione della cpu. I programmi Java sono prima compilati in un file .class, contenente bytecode Java, e quindi tradotti in un file eseguibile .dex. Mentre molte macchine virtuali Java eseguono la compilazione just-in-time (jit) per migliorare l'efficienza delle applicazioni, art esegue la compilazione anticipata (aot, ahead-of-time); i file .dex vengono compilati nel codice macchina nativo quando sono installati su un dispositivo, dove possono essere eseguiti su art. La compilazione aot consente un'esecuzione più efficiente delle applicazioni e un consumo energetico ridotto, caratteristiche fondamentali per i sistemi mobili.

Gli sviluppatori Android possono anche scrivere programmi Java che utilizzano l'interfaccia nativa Java, jni, che consente agli sviluppatori di bypassare la macchina virtuale e scrivere programmi Java che possono accedere a specifiche funzionalità hardware. I programmi scritti utilizzando jni generalmente non sono portabili da un hardware a un altro.

Il set di librerie native disponibili per le applicazioni Android include framework per lo sviluppo di browser web (*webkit*), per il supporto di database (sqlite) e per il supporto di rete, come nel caso di secure socket (ssl).

Poiché Android può essere eseguito su un numero quasi illimitato di dispositivi, Google ha scelto di astrarre l'hardware attraverso uno strato di astrazione hardware detto hal (*hardware abstraction layer*). Astraendo tutto l'hardware, tra cui la fotocamera, il chip gps e altri sensori, hal fornisce alle applicazioni una visione coerente e indipendente dallo specifico hardware. Questo strumento, naturalmente, consente agli sviluppatori di scrivere programmi portatili su piattaforme hardware diverse.

Mentre la libreria standard del linguaggio C utilizzata dai sistemi Linux è gnu C (*glibc*), Google ha sviluppato per Android la libreria C standard Bionic, che offre un'occupazione di memoria inferiore ed è progettata per lavorare su cpu più lente, come quelle presenti sui dispositivi mobili (inoltre, Bionic consente a Google di aggirare le licenze gpl di *glibc*).

In fondo alla pila di strati software di Android c'è il kernel di Linux, modificato da Google in diverse sue parti per supportare le esigenze speciali dei sistemi mobili, come il risparmio energetico. Google ha inoltre apportato modifiche alla gestione e all'allocazione della memoria e ha aggiunto una nuova forma di ipc nota come Binder (che tratteremo nel Paragrafo 3.8.2.1).

2.9 Generare e avviare un sistema operativo

Un sistema operativo si può progettare, codificare e realizzare specificamente per una singola macchina; tuttavia, è più diffusa la pratica di progettare sistemi operativi da impiegare in macchine di una stessa classe con configurazioni diverse.

2.9.1 Generazione di sistemi operativi

Nella maggior parte dei casi, quando si acquista un computer si trova il sistema operativo già installato. Per esempio, è possibile acquistare un computer portatile nuovo con Windows o macos preinstallati. Supponiamo ora che l'utente desideri sostituire il sistema operativo preinstallato oppure installare altri sistemi operativi in aggiunta (o ancora, supponiamo di acquistare un computer privo di sistema operativo). In tutte queste situazioni, si hanno diverse possibilità per installare il sistema operativo appropriato sul computer e configuralo per l'uso.

In caso si stia generando (o costruendo) un sistema operativo da zero, occorre seguire questi passaggi.

1. Scrivere il codice sorgente del sistema operativo (o ottenere il codice sorgente già scritto).
2. Configurare il sistema operativo per il sistema su cui verrà eseguito.
3. Compilare il sistema operativo.
4. Installare il sistema operativo.
5. Avviare il computer e il nuovo sistema operativo.

La configurazione del sistema richiede di specificare le funzionalità da includere e varia a seconda del sistema operativo. In genere, i parametri che descrivono la configurazione del sistema sono memorizzati in qualche tipo di file di configurazione; una volta creato, questo file può essere utilizzato in diversi modi.

Un amministratore di sistema può spingersi fino a usare il file di configurazione per modificare una copia del codice sorgente del sistema operativo. Il sistema viene quindi completamente ricompilato (system build) producendo in uscita una versione del sistema operativo, che comprende dichiarazioni e inizializzazioni dei dati e costanti, conforme al sistema descritto nel file di configurazione.

A un livello di personalizzazione un po' più basso, la descrizione del sistema può portare alla selezione di moduli oggetto precompilati da una libreria esistente; questi moduli sono collegati tra loro per formare il sistema operativo generato. Questo processo permette di avere una libreria contenente i driver di tutti i dispositivi i/o supportati, perché solo quelli necessari vengono selezionati e collegati al sistema operativo. Poiché il sistema non viene ricompilato la generazione del sistema è più veloce, ma il sistema risultante potrebbe essere eccessivamente generico e potrebbe non supportare diverse configurazioni hardware.

All'altro estremo è possibile costruire un sistema completamente modulare. In questo caso la selezione avviene al momento dell'esecuzione piuttosto che al momento della compilazione o del collegamento. La generazione del sistema implica semplicemente l'impostazione dei parametri che descrivono la configurazione del sistema.

Le principali differenze tra questi approcci sono la dimensione e la generalità del sistema generato e la facilità di modificarlo al variare della configurazione hardware. Per i sistemi embedded non è insolito adottare il primo approccio e creare un sistema operativo per una configurazione hardware statica specifica. Tuttavia, la maggior parte dei moderni sistemi operativi che supportano computer desktop e portatili e dispositivi mobili ha adottato il secondo approccio: il sistema operativo è ancora generato per una configurazione hardware specifica, ma l'uso di tecniche come i moduli del kernel caricabili permette modifiche dinamiche al sistema.

Illustriamo ora come costruire un sistema Linux da zero. In genere è necessario eseguire le seguenti operazioni.

1. Scaricare il codice sorgente di Linux da <http://www.kernel.org>.
2. Configurare il kernel usando il comando `"make menuconfig"`. Questo passaggio genera il file di configurazione `.config`.
3. Compilare il kernel principale usando il comando `"make"`, che compila il kernel in base ai parametri di configurazione presenti nel file `.config` producendo il file `vmlinuz`, che è l'immagine del kernel.
4. Compilare i moduli del kernel usando il comando `"make modules"`. Proprio come con la compilazione del kernel, la compilazione dei moduli dipende dai parametri di configurazione specificati nel file `.config`.
5. Utilizzare il comando `"make modules install"` per installare i moduli del kernel in `vmlinuz`.
6. Installare il nuovo kernel sul sistema con il comando `"make install"`.

Al riavvio del sistema, il nuovo sistema operativo inizierà a funzionare.

In alternativa è possibile modificare un sistema esistente installando una macchina virtuale Linux. Ciò consentirà al sistema operativo host (per esempio, Windows o macos) di eseguire Linux (abbiamo introdotto la virtualizzazione nel Paragrafo 1.7 e approfondiremo l'argomento nel Capitolo 18).

Ci sono diverse possibilità per l'installazione di Linux come macchina virtuale. Un'alternativa è creare una macchina virtuale da zero. Questa opzione è simile alla creazione di un sistema Linux da zero, anche se non è necessario compilare il sistema operativo. Un altro approccio consiste nell'utilizzare l'immagine di un sistema operativo Linux già creato e configurato. Questa opzione richiede semplicemente il download dell'immagine e la sua installazione tramite un software di virtualizzazione come VirtualBox o vmware. Per esempio, per creare il sistema operativo utilizzato nella macchina virtuale fornita con questo testo, gli autori hanno eseguito le seguenti operazioni:

1. hanno scaricato l'immagine iso di Ubuntu da <https://www.ubuntu.com/>
2. hanno fornito istruzioni al software di virtualizzazione VirtualBox in modo da utilizzare l'iso per il boot e hanno avviato la macchina virtuale
3. hanno risposto alle domande di installazione e hanno poi installato e avviato il sistema operativo come una macchina virtuale.

2.9.2 Avvio del sistema

Una volta che un sistema operativo è stato generato deve essere reso disponibile per l'uso da parte dell'hardware. Come può l'hardware sapere dove si trova il kernel o come caricarlo? Il processo di avvio di un computer, caricando il kernel del sistema operativo, è noto come *boot* e sulla maggior parte dei sistemi procede come segue.

1. Una piccola porzione di codice noto come programma di bootstrap o boot loader individua il kernel.
2. Il kernel viene caricato in memoria e avviato.
3. Il kernel inizializza l'hardware.
4. Il file system principale (*root file system*) viene montato.

Descriviamo ora brevemente il processo di avvio, in modo più dettagliato.

Alcuni sistemi informatici utilizzano il processo di avvio a più fasi (*multistage*): quando il computer viene acceso per la prima volta viene eseguito un piccolo boot loader situato nel firmware non volatile, noto come bios. Questo programma iniziale di solito non fa altro che caricare un secondo boot loader, che si trova in un punto fisso del disco chiamato blocco di avvio (*boot block*). Il programma memorizzato nel blocco di avvio potrebbe essere abbastanza sofisticato da caricare l'intero sistema operativo in memoria e iniziare l'esecuzione, ma più spesso si tratta di un codice semplice (dato che deve essere contenuto in un singolo blocco del disco) che conosce solamente l'indirizzo su disco e la lunghezza del resto del programma di bootstrap.

Molti sistemi informatici recenti hanno sostituito il processo di avvio basato sul bios con uefi (*Unified Extensible Firmware Interface*). uefi offre diversi vantaggi rispetto al bios, tra cui un migliore supporto per i sistemi a 64 bit e per dischi più grandi, ma il maggiore vantaggio è probabilmente che uefi è un gestore di avvio unificato e completo e quindi più veloce rispetto al processo di avvio a più stadi del bios.

Sia utilizzando il bios che utilizzando l'uefi il programma di bootstrap può eseguire una serie di attività tra cui, oltre a caricare il file contenente il kernel in memoria, eseguire la diagnostica per determinare lo stato della macchina, per esempio ispezionando la memoria e la cpu e rilevando la presenza di dispositivi. Se la diagnostica ha successo, il programma può continuare con i passaggi di avvio. Il bootstrap può anche inizializzare tutti gli aspetti del sistema, dai registri della cpu ai controllori dei dispositivi, al contenuto della memoria principale. Poi, presto o tardi, avvia il sistema operativo e monta il file system principale: solo a questo punto possiamo dire che il sistema è in esecuzione.

grub è un programma di bootstrap open-source per sistemi Linux e unix. I parametri di avvio per il sistema sono impostati in un file di configurazione di grub che viene caricato all'avvio. grub è flessibile e consente di apportare modifiche in fase di avvio, inclusa la modifica dei parametri del kernel e persino la selezione tra diversi kernel che possono essere avviati. Come esempio possiamo vedere i seguenti parametri del kernel presenti nel file speciale `/proc/cmdline` che viene utilizzato all'avvio:

```
BOOT_IMAGE=/boot/vmlinuz-4.4.0-59-generic
root=UUID=5f2e2232-4e47-4fe8-ae94-45ea749a5c92
```

`BOOT_IMAGE` è il nome dell'immagine del kernel da caricare in memoria e `root` specifica un identificativo univoco del file system principale.

Per risparmiare spazio e ridurre il tempo di avvio, l'immagine del kernel Linux è un file compresso che viene estratto solo dopo essere stato caricato in memoria. Durante il processo di avvio, il boot loader crea in genere un file system temporaneo in ram, noto come `initramfs`. Questo file system contiene driver e moduli del kernel necessari che devono essere installati per supportare il vero file system (che non è in memoria centrale). Una volta avviato il kernel e installati i driver necessari, cambia il file system principale dalla posizione temporanea in ram al percorso appropriato. Infine, Linux crea il processo `systemd`, il processo iniziale di sistema, quindi avvia altri servizi (per esempio, un server web e/o un database). Come ultimo passo il sistema presenta all'utente un prompt di login. Nel Paragrafo 11.5.2 descriveremo il processo di avvio per Windows.

Vale la pena notare che il meccanismo di avvio non è indipendente dal boot loader. Pertanto esistono versioni specifiche di grub per bios e per uefi e il firmware deve conoscere quale specifico boot loader deve essere usato.

Il processo di avvio per i sistemi mobili è leggermente diverso da quello dei pc tradizionali. Per esempio, sebbene il suo kernel sia basato su Linux, Android non usa grub, ma lascia ai distributori il compito di fornire un boot loader. Il boot loader Android più comune è lk ("little kernel"). Come i sistemi Linux, anche i sistemi Android utilizzano l'immagine del kernel compressa e un file system in ram iniziale; tuttavia, mentre Linux elimina `initramfs` una volta caricati tutti i driver necessari, Android lo mantiene come file system principale del dispositivo. Una volta caricato il kernel e installato il file system, Android avvia il processo di inizializzazione (`init`) e crea diversi servizi prima di visualizzare la schermata principale.

I boot loader per la maggior parte dei sistemi operativi, inclusi Windows, Linux, macos, ios e Android, consentono l'avvio in modalità di ripristino o in modalità utente singolo per diagnosticare problemi hardware, correggere file system corrotti e anche reinstallare il sistema operativo. Oltre ai guasti hardware, i sistemi di elaborazione possono soffrire di errori software e di prestazioni inadeguate del sistema operativo. Analizzeremo queste problematiche nel prossimo paragrafo.

2.10 Debugging dei sistemi operativi

In questo capitolo abbiamo menzionato spesso il debugging. Ne diamo ora una descrizione più dettagliata. Il debugging può essere genericamente definito come l'attività di individuare e risolvere errori hardware e software nel sistema, i cosiddetti bachi (*bug*). I problemi che condizionano le prestazioni sono considerati bachi, quindi il debugging può comprendere anche una regolazione delle prestazioni (*performance tuning*), che ha lo scopo di migliorare le prestazioni eliminando i colli di bottiglia (*bottleneck*) del sistema. In questo paragrafo tratteremo del debugging dei processi e del kernel e dei problemi di prestazioni; il tema del debugging dell'hardware esula invece dagli scopi di questo testo.

2.10.1 Analisi dei malfunzionamenti

Se un processo fallisce, la maggior parte dei sistemi operativi scrive le informazioni relative all'errore avvenuto in un file di log (*log file*), in modo da aggiornare operatori o utenti del sistema di ciò che è avvenuto. Il sistema operativo può anche acquisire e memorizzare in un file un'immagine del contenuto della memoria utilizzata dal processo, chiamata core dump (la memoria ai primordi dell'era informatica era chiamata *core*). Il debugger, uno strumento che permette al programmatore di esplorare il codice e la memoria di un processo, può esaminare i programmi in esecuzione e i core dump.

Se il debugging di processi a livello utente è una sfida, a livello del kernel del sistema operativo esso è un'attività ancora più difficile a causa della dimensione e della complessità del kernel, del suo controllo dell'hardware e della mancanza di strumenti per eseguire il debugging a livello utente. Un guasto nel kernel viene chiamato crash. Quando si verifica un crash le informazioni riguardanti l'errore vengono salvate in un file di log, mentre lo stato della memoria viene salvato in un'immagine del contenuto della memoria al momento del crash (*crash dump*).

Il debugging del sistema operativo e quello dei processi usano spesso strumenti e tecniche differenti, perché la natura delle due attività è molto diversa. Teniamo in considerazione il fatto che un malfunzionamento del kernel nel codice relativo al file system renderebbe rischioso per il kernel provare a salvare il suo stato in un file prima del riavvio. Una tecnica comune consiste nel salvare lo stato di memoria del kernel in una sezione del disco adibita esclusivamente a questo scopo, al di fuori del file system. Quando il kernel rileva un errore irrecuperabile scrive l'intero contenuto della memoria, o per lo meno delle parti della memoria di sistema possedute dal kernel, nell'area di disco a ciò destinata. Nel momento in cui il kernel si riavvia, viene eseguito un processo che raccoglie i dati da quest'area e li scrive in un apposito file all'interno del file system per un'analisi. Ovviamente queste tecniche sarebbero inutili nel debugging di normali processi di livello utente.

2.10.2 Monitoraggio e regolazione delle prestazioni

Abbiamo detto in precedenza che la regolazione delle prestazioni ha lo scopo di migliorare le prestazioni eliminando i colli di bottiglia. Per identificare eventuali colli di bottiglia dobbiamo essere in grado di monitorare le prestazioni del sistema. A tale scopo il sistema operativo deve comprendere strumenti per effettuare misurazioni sul comportamento del sistema e mostrare i risultati.

Questi strumenti possono effettuare osservazioni a livello di processo o a livello di sistema, utilizzando contatori (*counter*) oppure tracciamento (*tracing*). Esploriamo tutte queste tipologie nei seguenti paragrafi.

2.10.2.1 Contatori

I sistemi operativi tengono traccia dell'attività del sistema attraverso una serie di contatori, che contano per esempio il numero di chiamate di sistema effettuate o il numero di operazioni eseguite su un dispositivo o su un disco di rete. Di seguito sono riportati alcuni esempi di strumenti Linux che utilizzano contatori.

A livello di processo

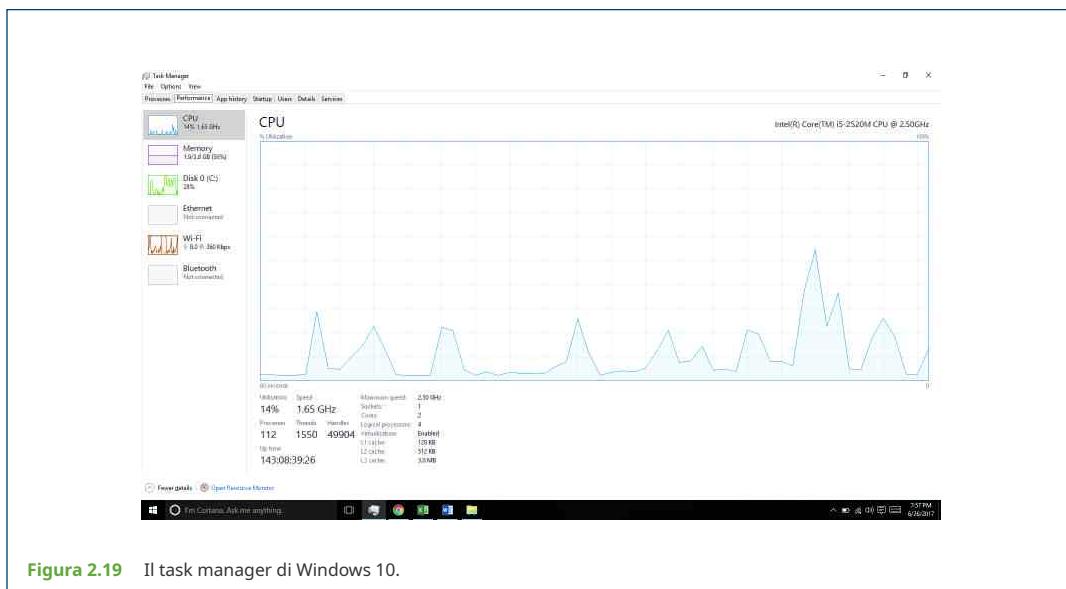
- `ps`: fornisce informazioni su un singolo processo o su una selezione di processi
- `top`: riporta le statistiche in tempo reale per i processi correnti

A livello di sistema

- `vmstat`: riporta le statistiche sull'utilizzo della memoria
- `netstat`: riporta le statistiche per le interfacce di rete
- `iostat`: riporta le statistiche sull'utilizzo dell'i/o per i dischi

La maggior parte degli strumenti basati su contatori nei sistemi Linux legge le statistiche dal file system `/proc`, uno "pseudo" file system esistente solo nella memoria del kernel e usato principalmente per richiedere diverse statistiche relative ai processi e al kernel. Il file system `/proc` è organizzato come una gerarchia di directory, dove un processo (o meglio il valore intero univoco, detto id, assegnato al processo) appare come una sottodirectory di `/proc`. Per esempio, la directory `/proc/2155` conterrà statistiche riguardanti il processo con id 2155. Il file system `/proc` contiene anche varie statistiche del kernel.

Nei sistemi Windows è presente il Task Manager di Windows (Gestione Attività), uno strumento che fornisce informazioni sulle applicazioni correnti, sui processi, e sull'utilizzo della cpu, della memoria e della rete. Una schermata del Task Manager di Windows 10 è mostrata nella Figura 2.19.



2.10.3 Tracing

Mentre gli strumenti basati su contatori si limitano a fornire il valore corrente di alcune statistiche mantenute dal kernel, gli strumenti di tracing raccolgono i dati relativi a uno specifico evento, come i passaggi coinvolti nell'invocazione di una chiamata di sistema.

I seguenti sono esempi di strumenti Linux per il tracciamento di eventi.

A livello di processo

- **strace**: traccia le chiamate di sistema invocate da un processo
- **gdb**: un debugger a livello di codice sorgente

A livello di sistema

- **perf**: una raccolta di strumenti per analizzare le prestazioni di Linux
- **tcpdump**: intercetta i pacchetti di rete

Gli studi per rendere i sistemi operativi più facili da comprendere, per renderne più semplice il debugging e per ottimizzarne le prestazioni mentre sono in esecuzione costituiscono un'area attiva nella ricerca e nella pratica operativa. Una nuova generazione di strumenti di analisi delle prestazioni abilitati dal kernel ha apportato miglioramenti significativi nel modo in cui questo obiettivo può essere raggiunto. Di seguito analizzeremo bcc, un kit di strumenti per il tracing dinamico del kernel in Linux.

LA LEGGE DI KERNIGHAN

“Il debugging è due volte più complesso rispetto alla stesura del codice. Di conseguenza, chi scrive il codice nella maniera più intelligente possibile non è, per definizione, abbastanza intelligente per eseguirne il debugging.”

2.10.4 BCC

Il debugging delle interazioni tra il livello utente e il codice del kernel è quasi impossibile senza un set di strumenti in grado di comprendere entrambi i codici e analizzare le loro interazioni. Affinché questo set di strumenti sia veramente utile, deve essere in grado di eseguire il debug di qualsiasi area del sistema, comprese aree che non sono state pensate per il debugging, e farlo senza influenzare l'affidabilità del sistema. Questo set di strumenti deve anche avere un impatto minimo sulle prestazioni; idealmente non dovrebbe influire quando non viene utilizzato e dovrebbe avere un impatto proporzionale all'uso. Il kit bcc soddisfa questi requisiti e fornisce un ambiente di debug dinamico, sicuro e a basso impatto.

bcc (bpf Compiler Collection) è un ricco toolkit che fornisce funzionalità di tracing per sistemi Linux costituito da un'interfaccia front-end per lo strumento ebpf (Extended Berkeley Packet Filter). La tecnologia bpf è stata sviluppata all'inizio degli anni '90 per

filtrare il traffico in una rete di computer. Il bpf “esteso” (ebpf) ha aggiunto varie funzionalità a bpf. I programmi ebpf sono scritti in un sottoinsieme del linguaggio C e sono compilati in istruzioni ebpf che possono essere inserite dinamicamente in un sistema Linux in esecuzione. Le istruzioni ebpf possono essere utilizzate per acquisire eventi specifici (come una certa chiamata di sistema che viene invocata) o per monitorare le prestazioni del sistema (per esempio il tempo necessario per eseguire i/o su disco). Per garantire un corretto comportamento delle istruzioni ebpf, queste vengono verificate prima di essere inserite nel kernel Linux in esecuzione. Il programma verificatore controlla che le istruzioni non influiscano negativamente sulle prestazioni o sulla sicurezza del sistema.

Nonostante ebpf offrisse un ricco insieme di funzionalità per il tracing all'interno del kernel di Linux, si è sempre incontrata una notevole difficoltà nell'utilizzo della sua interfaccia C per lo sviluppo di programmi. bcc è stato sviluppato per semplificare la scrittura di strumenti che sfruttano ebpf e fornisce un'interfaccia front-end in Python. Uno strumento bcc è scritto in Python e incorpora il codice C che si interfaccia con la ebpf, che a sua volta si interfaccia con il kernel. Inoltre, bcc compila il programma C in istruzioni ebpf e lo inserisce nel kernel utilizzando le *probe* o i *tracepoint*, due tecniche che consentono di tracciare gli eventi nel kernel di Linux.

Gli aspetti specifici della scrittura di strumenti bcc personalizzati vanno oltre lo scopo di questo testo, ma il pacchetto bcc (che è installato sulla macchina virtuale Linux che forniamo) fornisce numerosi strumenti in grado di monitorare diverse aree di attività in un kernel Linux in esecuzione. Per esempio, lo strumento `disksnoop` di bcc traccia l'attività di i/o del disco. Digitando il comando

```
./disksnoop.py
```

si può ottenere, per esempio, questo output:

TIME (s)	T	BYTES	LAT (ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

La tabella visualizzata indica il momento (visualizzando il timestamp) in cui si è verificata l'operazione di i/o, se si tratta di un'operazione di lettura (R) o scrittura (W) e quanti byte sono coinvolti nell'i/o. L'ultima colonna mostra la durata dell'i/o (espressa

come latenza) in millisecondi.

Molti degli strumenti forniti da bcc possono essere utilizzati per applicazioni specifiche, come database MySql e programmi Java e Python. È anche possibile collocare delle probe per monitorare l'attività di un processo specifico, come nel seguente esempio.

```
./opensnoop -p 1225
```

tracerà esclusivamente le chiamate di sistema `open()` eseguite dal processo con id 1225.

Ciò che rende bcc particolarmente potente è che i suoi strumenti possono essere utilizzati su sistemi attivi in produzione che eseguono applicazioni critiche senza causare danni al sistema. Ciò è particolarmente utile per gli amministratori di sistema che devono monitorare le prestazioni per identificare possibili colli di bottiglia o exploit di sicurezza. La Figura 2.20 illustra l'ampia gamma di strumenti attualmente forniti da bcc ed ebpf e la loro capacità di tracciare essenzialmente qualsiasi area del sistema operativo Linux. bcc è una tecnologia in rapida evoluzione e nuove funzionalità vengono costantemente aggiunte.

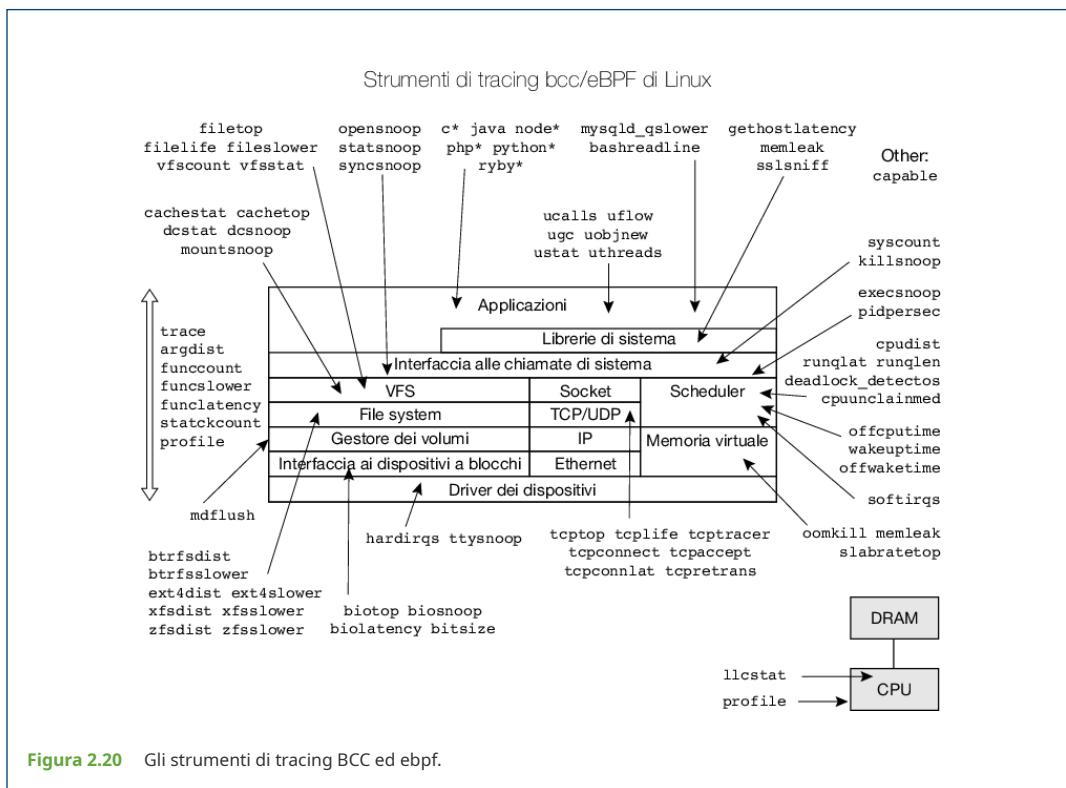


Figura 2.20 Gli strumenti di tracing BCC ed ebpf.

2.11 Sommario

- Un sistema operativo offre un ambiente per l'esecuzione dei programmi fornendo servizi a utenti e applicazioni.
- I tre approcci principali per l'interazione con un sistema operativo sono (1) gli interpreti dei comandi, (2) le interfacce grafiche e (3) le interfacce touch-screen.
- Le chiamate di sistema forniscono un'interfaccia per i servizi resi disponibili da un sistema operativo. I programmatore utilizzano un'api (Application Programming Interface) come interfaccia per l'accesso ai servizi forniti dalle chiamate di sistema.
- Le chiamate di sistema possono essere suddivise in sei categorie principali: (1) controllo dei processi, (2) gestione dei file, (3) gestione dei dispositivi, (4) manutenzione delle informazioni, (5) comunicazioni e (6) protezione.
- La libreria standard del linguaggio C fornisce l'interfaccia alle chiamate di sistema su sistemi unix e Linux.
- I sistemi operativi includono anche una raccolta di programmi di sistema che offrono varie funzionalità agli utenti.
- Un linker combina diversi moduli oggetto rilocabili in un singolo file binario eseguibile. Un loader carica il file eseguibile in memoria, dove diventa idoneo per essere eseguito su una cpu disponibile.
- Vi sono diversi motivi per cui le applicazioni sono specifiche del sistema operativo, tra cui i differenti formati binari per i file eseguibili di un programma, i diversi set di istruzioni per cpu distinte e le chiamate di sistema che variano da un sistema operativo all'altro.
- Un sistema operativo è progettato pensando a obiettivi specifici che determinano, in ultima analisi, le politiche del sistema operativo. Un sistema operativo implementa queste politiche attraverso meccanismi specifici.
- Un sistema operativo monolitico non ha struttura; tutte le funzionalità sono contenute in un singolo file binario statico che viene eseguito in un unico spazio d'indirizzamento. Sebbene tali sistemi siano difficili da modificare, il loro principale vantaggio è l'efficienza.
- Un sistema operativo stratificato è suddiviso in un certo numero di strati, in cui lo strato inferiore è l'interfaccia hardware e quello superiore è l'interfaccia utente. Sebbene i sistemi stratificati abbiano avuto un certo successo, questo approccio non è generalmente ideale per la progettazione di sistemi operativi a causa di problemi di prestazioni.
- L'approccio microkernel per la progettazione di sistemi operativi utilizza un kernel minimale; la maggior parte dei servizi è eseguita come applicazione a livello utente. La comunicazione avviene tramite lo scambio di messaggi.
- Un approccio modulare per la progettazione di sistemi operativi fornisce i servizi del sistema operativo attraverso moduli che possono essere caricati e rimossi durante l'esecuzione. Molti sistemi operativi contemporanei sono progettati come sistemi ibridi e combinano un kernel monolitico con l'utilizzo di moduli.
- Un boot loader carica un sistema operativo in memoria, esegue l'inizializzazione e avvia l'esecuzione del sistema.
- Le prestazioni di un sistema operativo possono essere monitorate utilizzando contatori o tracing. I contatori sono una raccolta di statistiche a livello di sistema o di processo, mentre il tracing segue l'esecuzione di un programma attraverso il sistema operativo.

Esercizi di ripasso

2.1 Qual è lo scopo delle chiamate di sistema?

2.2 Qual è lo scopo dell'interprete dei comandi? Perché è solitamente separato dal kernel?

2.3 Quali chiamate di sistema devono essere eseguite dall'interprete dei comandi, o shell, per avviare un nuovo processo?

2.4 Qual è lo scopo dei programmi di sistema?

2.5 Qual è il vantaggio principale dell'approccio a strati (layer) all'architettura di sistema? Quali sono invece i suoi svantaggi?

2.6 Elencate cinque servizi forniti da un sistema operativo e spiegate la convenienza per l'utente di ciascuno. In quali casi sarebbe impossibile per i programmi a livello utente offrire questi servizi? Argomentate la vostra risposta.

2.7 Perché alcuni sistemi memorizzano il sistema operativo nel firmware mentre altri lo memorizzano su disco?

2.8 Come potrebbe essere progettato un sistema perché offra la possibilità di scegliere quale sistema operativo avviare? Che cosa dovrebbe fare in questo caso il bootstrap?

Esercizi

2.9 I servizi e le funzioni offerti da un sistema operativo possono essere divisi in due categorie. Procedete a una loro breve descrizione, analizzandone le differenze.

2.10 Descrivete tre metodi generali per passare parametri al sistema operativo.

2.11 Descrivete come si possa ottenere un profilo statistico del tempo consumato da un programma per eseguire le differenti parti del proprio codice. Argomentate l'importanza di simili profili statistici.

2.12 Quali sono i vantaggi e gli svantaggi di usare la medesima interfaccia alle chiamate di sistema sia per i file sia per i dispositivi?

2.13 Sarebbe possibile per l'utente sviluppare un nuovo interprete dei comandi utilizzando le chiamate di sistema offerte dal sistema operativo?

2.14 Descrivete perché Android esegue la compilazione ahead-of-time (aot) invece della compilazione just-in-time (jit).

2.15 Quali sono i due modelli della comunicazione tra processi? Quali i loro punti di forza e di debolezza?

2.16 Evidenziate le differenze tra un'interfaccia per la programmazione di applicazioni (api) con una abi (*application binary interface*).

2.17 Perché è auspicabile separare i meccanismi dalle politiche?

2.18 Talvolta è difficile realizzare un'architettura a strati se due componenti del sistema operativo dipendono l'uno dall'altro. Identificate una situazione in cui non risulti immediatamente evidente come stratificare due componenti del sistema che hanno funzionalità strettamente connesse.

2.19 Quale vantaggio si riscontra nell'architettura orientata al microkernel? In che modo interagiscono i programmi utenti e i servizi del sistema in tale architettura? Quali sono gli svantaggi?

2.20 Quali sono i vantaggi dell'utilizzo di moduli del kernel caricabili dinamicamente?

2.21 In che cosa sono simili ios e Android? In che cosa differiscono?

2.22 Spiegate perché i programmi Java in esecuzione su sistemi Android non usano le api e la macchina virtuale standard di Java.

2.23 Il sistema operativo sperimentale Synthesis ha un assemblatore incorporato nel kernel. Per ottimizzare le prestazioni delle chiamate di sistema, il kernel assembla le procedure nello spazio del kernel, al fine di ridurre al minimo il percorso che le chiamate di sistema devono seguire attraverso il kernel. Tale metodo è in antitesi al metodo stratificato che, per rendere più semplice la costruzione di un sistema operativo, determina un prolungamento del percorso delle chiamate di sistema attraverso il kernel. Valutate i pro e i contro di tale metodo nella progettazione di un kernel e nell'ottimizzazione delle prestazioni di un sistema.

2.24 Nel Paragrafo 2.3 si è descritto un programma che copia i contenuti di un file di origine in un file di destinazione. Questo programma comincia con la richiesta, indirizzata all'utente, dei nomi dei file di origine e di destinazione. Si scriva un tale programma usando la api Windows o posix. Prestate particolare attenzione alla gestione degli errori, assicurandosi ad esempio che il file di origine esista. Dopo averlo sviluppato e testato, eseguite il programma insieme a un'applicazione per la tracciatura delle chiamate di sistema, se si dispone di un sistema operativo con tale funzionalità. I sistemi Linux forniscono l'utilità `strace`, e i sistemi macos hanno il comando `dtruss`. (Il comando `dtruss`, che in effetti è un front-end di `dtrace`, richiede privilegi `admin`, quindi deve essere lanciato utilizzando `sudo`). Questi strumenti possono essere utilizzati come segue (supponiamo che il nome del file eseguibile sia `FileCopy`):

Linux:

```
strace ./FileCopy
```

macos:

```
sudo dtruss ./FileCopy.
```

Dato che i sistemi Windows non offrono queste caratteristiche, è necessario eseguire il tracciamento della versione Windows del programma utilizzando un debugger.

In questo progetto imparerete come creare un modulo del kernel e come caricarlo nel kernel di Linux. In seguito modificherete il modulo in modo che crei un elemento nel file system `/proc`. Il progetto è realizzabile utilizzando la macchina virtuale Linux disponibile con questo testo. Anche se è possibile usare un editor per scrivere questi programmi in C, dovrete utilizzare l'applicazione terminal per compilare i programmi e inserire i comandi nella riga di comando per gestire i moduli del kernel.

Come scoprirete, il vantaggio di sviluppare moduli del kernel è che si tratta di un metodo relativamente semplice di interagire con il kernel, che vi permette di scrivere programmi che ne richiamano direttamente le funzioni. È importante tenere a mente che si sta effettivamente scrivendo codice del kernel e che questo interagisce direttamente con il kernel. Ciò di norma significa che eventuali errori nel codice potrebbero mandare in crash il sistema! Tuttavia, visto che userete una macchina virtuale, eventuali errori richiederanno nella peggiore delle ipotesi soltanto il riavvio del sistema.

Parte I – Creazione di moduli del kernel

La prima parte di questo progetto richiede di eseguire una sequenza di passaggi per creare un modulo e inserirlo nel kernel di Linux. È possibile elencare tutti i moduli del kernel che sono attualmente caricati con il comando

```
lsmod
```

Questo comando consente di visualizzare i moduli attualmente caricati nel kernel in tre colonne: una colonna rappresenta il nome, un'altra la dimensione, e l'ultima indica dove il modulo viene utilizzato.

Il programma della Figura 2.21 (denominato `simple.c` e disponibile con il codice sorgente per questo testo) mostra un semplice modulo del kernel che consente di stampare opportuni messaggi quando il modulo del kernel viene caricato e scaricato.

```
#include <linux/init.h> #include <linux/kernel.h> #include <linux/module.h>

/* Questa funzione viene chiamata quando viene caricato il modulo. */

int simple_init (void) { printk(KERN_INFO "Loading Module\n"); return 0; }

/* Questa funzione viene chiamata quando il modulo viene rimosso. */

void simple_exit(void) { printk(KERN_INFO "Removing Module\n"); }

/* Macro per la registrazione di ingresso e di uscita del modulo. */

module_init(simple_init); module_exit(simple_exit); MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple
Module"); MODULE_AUTHOR("SGG");
```

Figura 2.21 Il programma `simple.c`.

La funzione `simple_init()` è il punto di ingresso del modulo (*module entry point*), ovvero la funzione che viene richiamata quando il modulo viene caricato nel kernel. Analogamente, la funzione `simple_exit()` è il punto di uscita del modulo (*module exit point*) ed è invocata quando il modulo viene rimosso dal kernel.

La funzione di ingresso del modulo deve restituire un valore intero, dove 0 rappresenta il successo e ogni altro valore rappresenta il fallimento. La funzione di uscita del modulo restituisce `void`. Né il punto di ingresso, né il punto di uscita del modulo ricevono parametri. Le due seguenti macro sono utilizzate per registrare i punti di ingresso e di uscita nel kernel:

```
module_init(simple_init)
```

```
module_exit(simple_exit)
```

Notate che entrambe le funzioni `simple_init()` e `simple_exit()` effettuano chiamate alla funzione `printk()`. La funzione `printk()` è l'equivalente nel kernel della funzione `printf()`; il suo output viene inviato a un buffer di log del kernel il cui contenuto può essere letto dal comando `dmesg`. Una differenza tra `printf()` e `printk()` è che quest'ultima ci permette di specificare un flag di priorità i cui valori sono riportati nel file di include `<linux/printk.h>`. Nel nostro caso, la priorità è `KERN_INFO` ed è definita come un messaggio informativo.

Le ultime linee, `MODULE_LICENSE()`, `MODULE_DESCRIPTION()` e `MODULE_AUTHOR()`, forniscono dettagli riguardanti la licenza software, la descrizione del modulo, e l'autore. Queste informazioni non sono importanti per i nostri scopi, ma le includiamo perché è una pratica comune nello sviluppo di moduli del kernel.

Il modulo `simple.c` viene compilato con il `Makefile` che accompagna il codice sorgente di questo progetto. Per compilare il modulo, digitate nella riga di comando:

```
make
```

La compilazione produce diversi file. Il file `simple.ko` rappresenta il modulo del kernel compilato. Il passaggio successivo illustra l'inserimento di questo modulo nel kernel di Linux.

Caricamento e rimozione dei moduli del kernel

I moduli del kernel vengono caricati con il comando `insmod`, come segue:

```
sudo insmod simple.ko
```

Per verificare se il modulo è stato caricato immettete il comando `lsmod` e cercate il modulo `simple`. Ricordiamo che il punto di ingresso del modulo viene invocato quando il modulo è inserito nel kernel. Per controllare il contenuto di questo messaggio nel buffer di log del kernel, digitate il comando

```
dmesg
```

Si dovrebbe vedere il messaggio "Loading Module."

Per rimuovere il modulo dal kernel si invoca il comando `rmmmod` (si noti che il suffisso `.ko` non è necessario):

```
sudo rmmmod simple
```

Assicuratevi di controllare con il comando `dmesg` che il modulo sia stato effettivamente rimosso.

Poiché il buffer di log del kernel può rapidamente riempirsi, in molti casi vale la pena di svuotarlo periodicamente, con l'utilizzo del comando:

```
sudo dmesg -c
```

Parte I – Esercizio

Eseguite i passi sopra descritti per creare il modulo del kernel e per caricare e scaricare il modulo. Assicuratevi di controllare il contenuto del buffer di log del kernel usando `dmesg` per verificare di aver correttamente seguito la procedura.

Parte II – Strutture dati del kernel

Dato che i moduli del kernel vengono eseguiti nell’ambito del kernel è possibile leggere valori e chiamare funzioni che sono disponibili solo nel kernel e non per le normali applicazioni utente. Per esempio, il file di inclusione `<linux/hash.h>` definisce varie funzioni hash che possono essere utilizzate nel kernel. Questo file contiene anche la costante `GOLDEN_RATIO_PRIME` (definita come `unsigned long`). Questo valore può essere stampato nel modo seguente:

```
printf(KERN_INFO "%lu/n", GOLDEN_RATIO_PRIME);
```

Come altro esempio, il file `<linux/gcd.h>` definisce la seguente funzione:

```
unsigned long gcd(unsigned long a, unsigned b);
```

che restituisce il massimo divisore comune dei parametri `a` e `b`.

Dopo che avete caricato e scaricato correttamente il vostro modulo, effettuate questi ulteriori passi:

1. stampate il valore di `GOLDEN_RATIO_PRIME` nella funzione `simple_init()`;
2. stampate il valore del massimo divisore comune di 3.300 e 24 nella funzione `simple_exit()`.

Dato che gli errori di compilazione non sono particolarmente utili quando si effettuano sviluppi a livello kernel, è importante compilare spesso il vostro programma effettuando regolarmente dei make. Curatevi di caricare e rimuovere il modulo del kernel e verificare il log buffer tramite `dmesg` per assicurarvi che le modifiche a `simple.c` funzionino correttamente.

Nel Paragrafo 1.4.3 è stato descritto il ruolo del timer e del relativo interrupt handler. In Linux, la frequenza del timer (*tick rate*) è il valore di `Hz` definito in `<asm/param.h>`. Il valore di `Hz` determina la frequenza di interruzioni del timer, e il suo valore dipende da tipo di macchina e architettura. Per esempio, se il valore di `Hz` è 100, si hanno 100 interruzioni del timer ogni secondo, ossia una ogni 10 millisecondi. Inoltre, il sistema mantiene la variabile globale `jiffies`, che contiene il numero di timer interrupt che si sono verificati dal boot del sistema. La variabile `jiffies` è dichiarata nel file `<linux/jiffies.h>`.

1. Stampate il valore di `jiffies` e `Hz` nella funzione `simple_init()`.
2. Stampate il valore di `jiffies` nella funzione `simple_exit()`.

Prima di procedere con il resto dell’esercitazione, pensate a come potete usare i differenti valori di `jiffies` in `simple_init()` e `simple_exit()` per determinare il numero di secondi trascorsi fra il caricamento e la rimozione del modulo del kernel.

Parte III – Il File System /proc

Il file system `/proc` è uno “pseudo” file system che esiste solo nella memoria del kernel e viene utilizzato principalmente per raccogliere varie statistiche relative al kernel e ai singoli processi. Questo esercizio richiede di progettare moduli del kernel che creano ulteriori elementi nel file system `/proc` che riportano sia statistiche del kernel sia informazioni relative a processi specifici. Il programma è rappresentato nelle Figure 2.22 e 2.23.

Incominciamo descrivendo come si crea un nuovo elemento nel file system `/proc`. Il programma seguente (chiamato `hello.c` e disponibile con il codice sorgente per questo libro) crea una voce di `/proc` chiamata `/proc/hello`. Se un utente esegue il comando:

```
cat /proc/hello
```

viene restituito il famoso messaggio `Hello World`.

Creiamo una nuova voce `/proc/hello` nel punto di ingresso del modulo `proc_init()` utilizzando la funzione `proc_create()`. A questa funzione viene passato `proc_ops` che contiene un riferimento a una `struct file_operations`. Questa `struct` inizializza i membri `.owner` e `.read`. Il valore di `.read` è il nome della funzione `proc_read()` che deve essere chiamata ogni volta che viene letto `/proc/hello`.

Esaminando questa funzione `proc_read()`, vediamo che la stringa `“Hello World\n”` viene scritta nella variabile `buffer`, che esiste nella memoria del kernel. Dato che si può accedere a `/proc/hello` dallo spazio utente, è necessario copiare il contenuto di `buffer` nello spazio utente utilizzando la funzione del kernel `copy_to_user()`. Questa funzione copia il contenuto del buffer della memoria kernel nella variabile `usr_buf`, che si trova nello spazio utente.

Ogni volta che si legge il file `/proc/hello`, viene chiamata ripetutamente la funzione `proc_read()`, fino a che non restituisce 0, quindi vi deve essere una logica che assicura che la funzione restituisca 0 dopo che ha raccolto i dati (in questo caso la stringa `“Hello World\n”`) che devono andare nel corrispondente file `/proc/hello`.

Infine, notate che il file `/proc/hello` viene eliminato nel punto di uscita del modulo `proc_exit()`, utilizzando la funzione `remove_proc_entry()`.

Parte IV – Esercizio

Questo esercizio richiede di progettare due moduli del kernel.

Progettate un modulo del kernel che crei un file di `/proc` chiamato `/proc/jiffies` che riporta il valore di `jiffies` al momento in cui viene letto il file `/proc/jiffies`, per esempio mediante il comando

```
cat /proc/jiffies
```

Assicuratevi di eliminare `/proc/jiffies` quando viene rimosso il modulo.

Progettate un modulo del kernel che crei un file di `/proc` chiamato `/proc/seconds` che riporta il numero di secondi trascorsi dal momento in cui il modulo è stato caricato. Ciò richiede di usare il valore di `jiffies` e la frequenza `hz`. Quando un utente esegue il comando:

```
cat /proc/seconds
```

il vostro modulo del kernel riporterà il numero di secondi trascorsi da quando il modulo è stato inizialmente caricato. Assicuratevi di eliminare `/proc/seconds` quando viene rimosso il modulo.

CAPITOLO 3

Processi

I primi sistemi elaborativi consentivano l'esecuzione di un solo programma alla volta, che aveva il completo controllo del sistema e accesso a tutte le sue risorse. Gli attuali sistemi elaborativi consentono, invece, che più programmi siano caricati in memoria ed eseguiti in modo concorrente. Tale evoluzione richiede un più severo controllo e una maggiore compartimentazione dei vari programmi. Da tali necessità deriva la nozione di processo d'elaborazione – o, più brevemente, processo – che in prima istanza si può definire come un programma in esecuzione, e costituisce l'unità di lavoro dei moderni sistemi time-sharing.

Maggiore è la complessità di un sistema operativo, maggiori sono i servizi che si suppone esso fornisca ai propri utenti. Benché il suo compito principale sia l'esecuzione dei programmi utenti, deve anche occuparsi dei vari compiti di sistema che è più conveniente lasciare fuori dal kernel. Un sistema è quindi costituito da un insieme di processi: quelli del sistema operativo eseguono il codice di sistema, i processi utente il codice utente. Tutti questi processi si possono eseguire potenzialmente in modo concorrente e l'uso della cpu (o di più cpu) è commutato tra i vari processi. Il sistema operativo può rendere il calcolatore più produttivo avvicendando i diversi processi nell'uso della cpu. In questo capitolo tratteremo dei processi e del loro funzionamento.

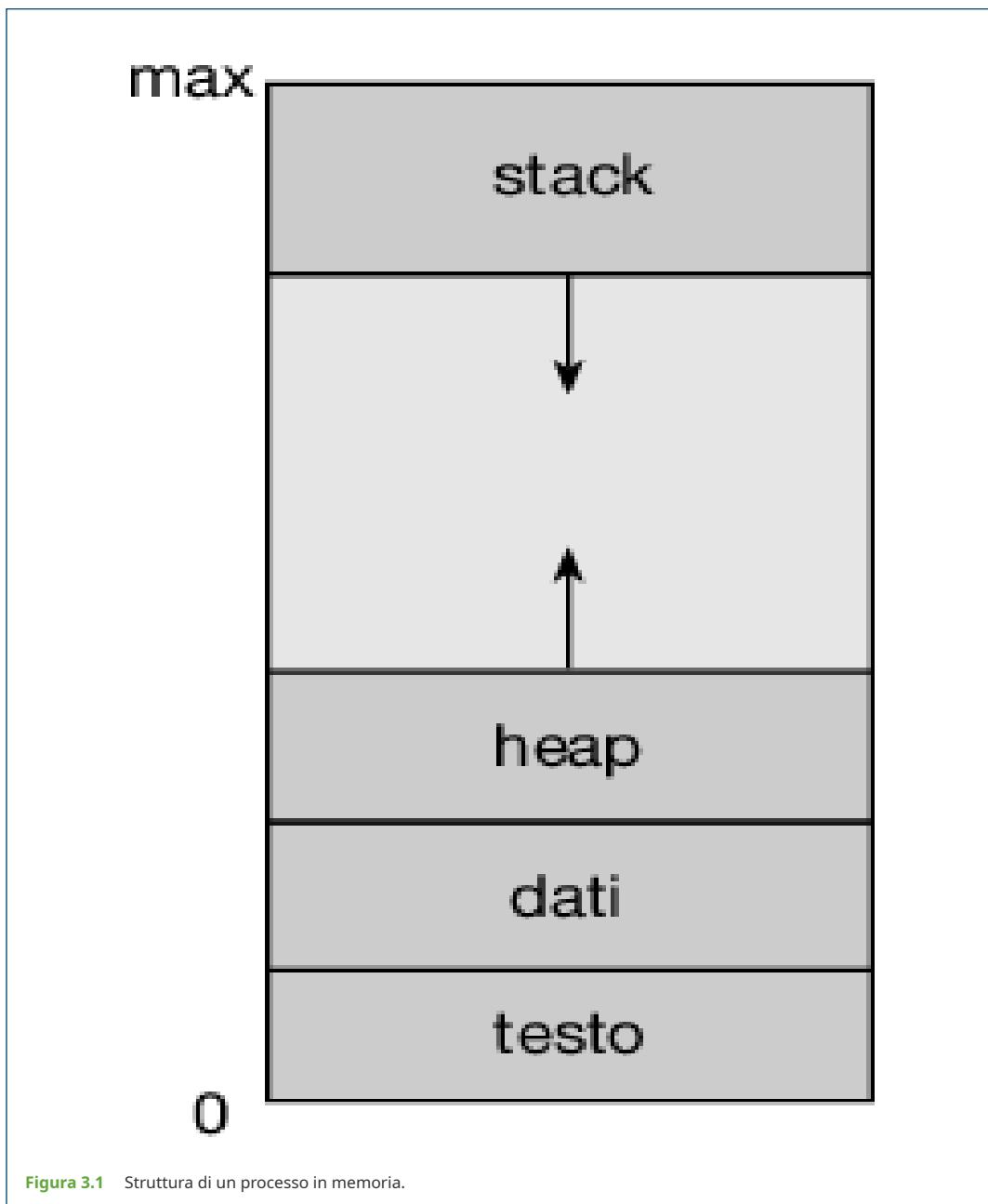
3.1 Concetto di processo

Una questione che sorge dall'analisi dei sistemi operativi è quella di dare un nome alle attività della cpu. Un sistema batch (*lotti*) esegue job (*lavori*), mentre un sistema time-sharing esegue programmi utenti o task. Persino in un sistema monoutente un utente può far eseguire diversi programmi contemporaneamente: un word processor, un browser, un programma di posta elettronica. Anche se l'utente esegue un solo programma alla volta, come avviene su dispositivi embedded che non supportano il multitasking, il sistema operativo deve svolgere le proprie attività interne, per esempio la gestione della memoria. Queste attività sono simili per molti aspetti, perciò sono denominate processi.

Sebbene noi preferiamo il termine più moderno *processo*, occorre ricordare che la maggior parte della terminologia e della teoria dei sistemi operativi si è sviluppata in un periodo in cui l'attività principale dei sistemi operativi riguardava la gestione dei *job* in sistemi batch. Quando lo riterremo appropriato useremo quindi le parole *lavoro* o *job* nella descrizione del ruolo dei sistemi operativi. Per esempio sarebbe fuorviante evitare di usare termini comunemente accettati che contengono la parola *job*, per esempio *job scheduling*, solo perché il termine *processo* ha ormai soppiantato il termine *job*.

3.1.1 Il processo

Informalmente, come affermato precedentemente, un processo è un programma in esecuzione. Lo stato dell'attività corrente di un processo è rappresentato dal valore del contatore di programma e dal contenuto dei registri del processore. La struttura di un processo in memoria è generalmente suddivisa in più sezioni, come mostrato nella Figura 3.1. Tali sezioni sono le seguenti.



- Sezione di testo: contenente il codice eseguibile.
- Sezione dati: contenente le variabili globali.
- Heap: memoria allocata dinamicamente durante l'esecuzione del programma.
- Stack: memoria temporaneamente utilizzata durante le chiamate di funzioni (per esempio, per i parametri della funzione, gli indirizzi di ritorno e le variabili locali).

Si noti che le dimensioni delle sezioni di testo e dati sono fisse, ovvero non cambiano durante l'esecuzione del programma, mentre le sezioni stack e heap possono ridursi e crescere dinamicamente durante l'esecuzione. Ogni volta che si chiama una funzione, un record di attivazione (*activation record*) contenente i suoi parametri, le variabili locali e l'indirizzo di ritorno viene inserito nello stack; quando la funzione restituisce il controllo al chiamante, il record di attivazione viene rimosso dallo stack. Allo stesso modo, l'heap crescerà quando viene allocata memoria dinamicamente e si ridurrà quando la memoria viene restituita al sistema. Visto che le sezioni stack e heap crescono l'una verso l'altra, tocca al sistema operativo garantire che *non si sovrappongano*.

Sottolineiamo che un programma di per sé non è un processo; un programma è un'entità *passiva*, come un file su disco contenente una lista di istruzioni (normalmente chiamato file eseguibile), mentre un processo è un'entità *attiva*, con un contatore di programma che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo

allorquando il file eseguibile è caricato in memoria. Due tecniche comuni per ottenere questo effetto sono il doppio clic sull''icona del file eseguibile e la digitazione del nome del file eseguibile nella riga di comando (scrivendo per esempio: `prog.exe` oppure `a.out`).

Sebbene due processi siano associabili allo stesso programma, sono tuttavia da considerare due sequenze d'esecuzione distinte. Alcuni utenti possono, per esempio, far eseguire diverse istanze dello stesso programma di posta elettronica, così come un utente può invocare più istanze dello stesso browser. Ciascuna di queste è un diverso processo e, benché le sezioni di testo siano equivalenti, quelle dei dati, dello heap e dello stack sono diverse. È inoltre usuale che durante la propria esecuzione un processo generi altri processi. Questo argomento è trattato nel Paragrafo 3.4.

Si noti che un processo può essere un ambiente di esecuzione per altro codice. L'ambiente di programmazione Java fornisce un buon esempio. Nella maggior parte dei casi, un programma Java viene eseguito all'interno della macchina virtuale Java (jvm). La jvm è in esecuzione come un processo che interpreta il codice Java caricato e intraprende azioni (tramite istruzioni macchina native) per conto di quel codice. Per esempio, per eseguire il programma Java compilato `Program.class`, dobbiamo scrivere

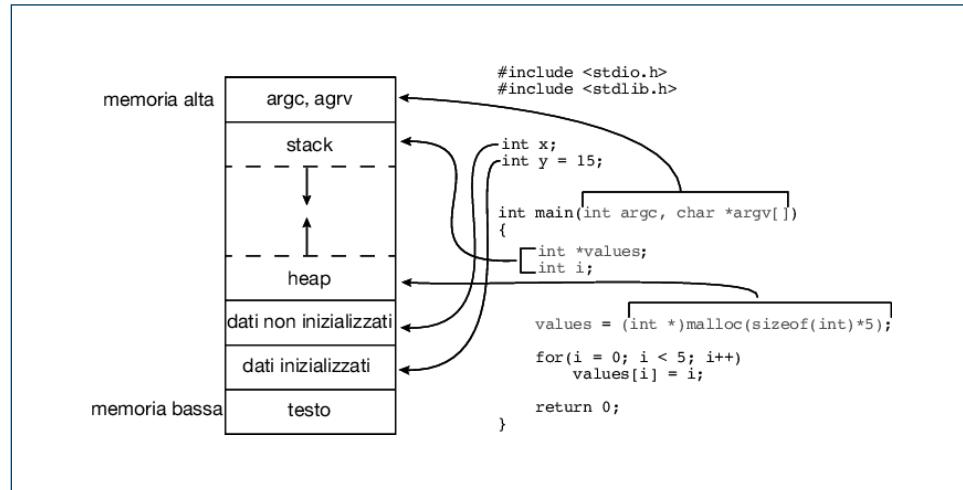
```
java Program
```

Il comando `java` manda in esecuzione la jvm come un processo ordinario che a sua volta esegue il programma `Java Program` nella macchina virtuale. Il concetto è analogo a quello di simulazione, fatta eccezione per il fatto che il codice anziché essere scritto per un insieme di istruzioni differente è scritto in linguaggio Java.

STRUTTURA IN MEMORIA DI UN PROGRAMMA C

La figura seguente mostra la struttura di un programma C in memoria, evidenziando la relazione tra le diverse sezioni di un processo e un programma C reale. Questa figura è simile alla rappresentazione generale di un processo in memoria, mostrata nella Figura 3.1, con alcune differenze.

- La sezione dei dati globali è suddivisa in sezioni distinte per (a) dati inizializzati e (b) dati non inizializzati.
- È presente una sezione separata per i parametri `argc` e `argv` passati alla funzione `main()`.



Il comando gnu `size` può essere usato per determinare la dimensione (in byte) di alcune di queste sezioni. Supponendo che il nome del file eseguibile del programma C sopra riportato sia `memory`, eseguendo il comando `size memory` si ottiene il seguente output:

```

text data bss dec hex filename
1158 284 8 1450 5aa memory

```

Il campo `data` si riferisce ai dati non inizializzati, mentre `bss` fa riferimento ai dati inizializzati (`bss` è un termine storico che sta per *block started by symbol*). I valori `dec` e `hex` sono la somma delle tre sezioni rappresentate rispettivamente in decimale ed esadecimale.

3.1.2 Stato del processo

Un processo durante l'esecuzione è soggetto a cambiamenti del suo stato, definito in parte dall'attività corrente del processo stesso. Un processo può trovarsi in uno tra i seguenti stati.

- Nuovo. Si crea il processo.
- Esecuzione (running). Le sue istruzioni vengono eseguite.
- Attesa (waiting). Il processo attende che si verifichi qualche evento (come il completamento di un'operazione di i/o o la ricezione di un segnale).
- Pronto (ready). Il processo attende di essere assegnato a un'unità d'elaborazione.
- Terminato. Il processo ha terminato l'esecuzione.

Questi termini sono piuttosto arbitrari e variano secondo il sistema operativo. Gli stati che rappresentano sono in ogni modo presenti in tutti i sistemi, anche se alcuni sistemi operativi introducono ulteriori distinzioni tra gli stati dei processi. È importante capire che in ciascuna unità d'elaborazione può essere in *esecuzione* solo un processo per volta, sebbene molti processi possano essere *pronti* o nello stato di *attesa*. Il diagramma di transizione fra questi stati è riportato nella Figura 3.2.

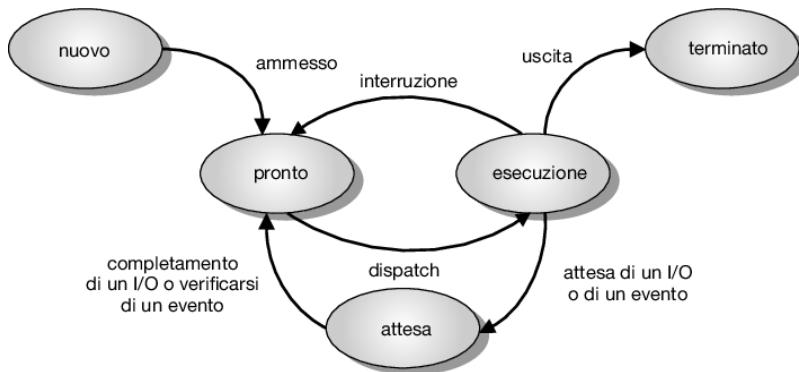


Figura 3.2 Diagramma di transizione degli stati di un processo.

3.1.3 Blocco di controllo del processo

Ogni processo è rappresentato nel sistema operativo da un blocco di controllo (*process control block*, pcb, o *task control block*, tcb). Un pcb (Figura 3.3) contiene molte informazioni connesse a un processo specifico, tra cui le seguenti.

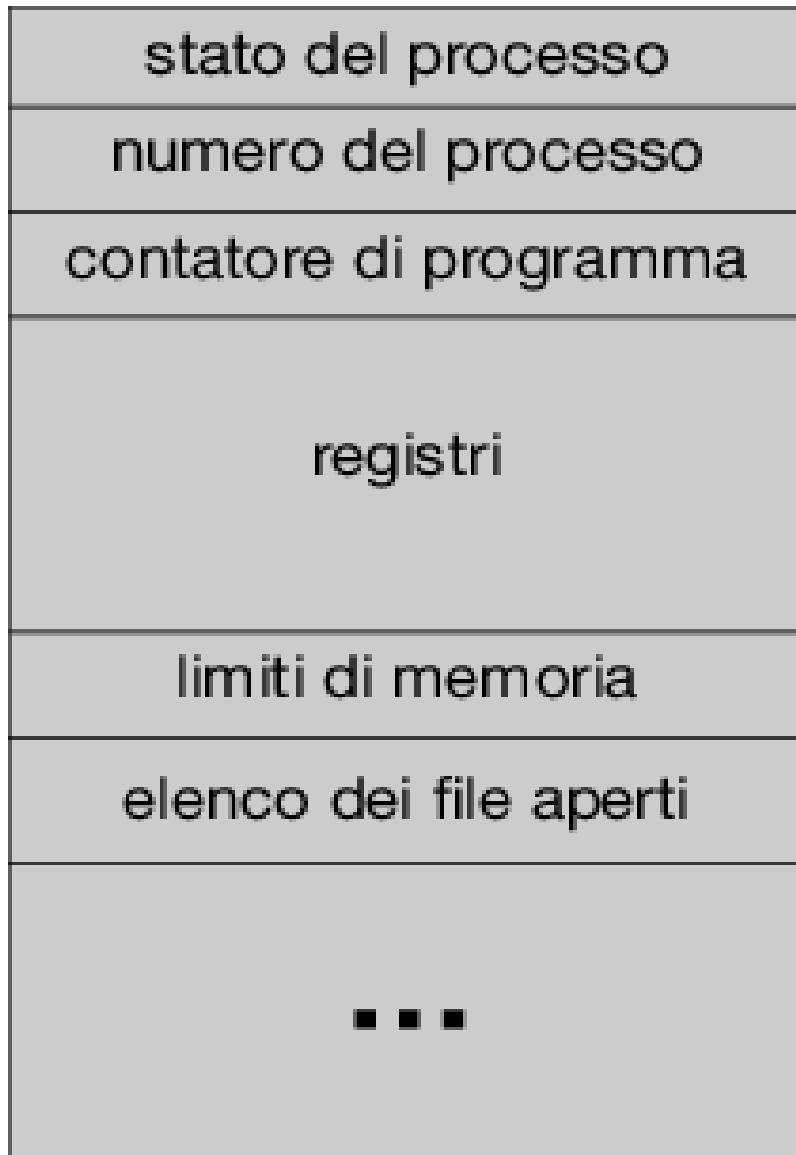


Figura 3.3 Blocco di controllo di un processo (pcb).

- Stato del processo. Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.
- Contatore di programma. Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- Registri della cpu. I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri d'uso generale e registri contenenti i codici di condizione (*condition codes*). Quando si verifica un'interruzione della cpu si devono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo, quando viene rischedulato.
- Informazioni sullo scheduling di cpu. Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling (nel Capitolo 5 si descrive lo scheduling dei processi).
- Informazioni sulla gestione della memoria. Queste informazioni possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo (Capitolo 9).
- Informazioni di accounting. Queste informazioni comprendono la quota di uso della cpu e il tempo d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.

- Informazioni sullo stato dell'i/o. Queste informazioni comprendono la lista dei dispositivi di i/o assegnati a un determinato processo, l'elenco dei file aperti, e così via.

In sintesi, il pcb si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

3.1.4 Thread

Il modello dei processi illustrato fin qui sottintende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione, detto thread. Se un processo sta, per esempio, eseguendo un browser, l'esecuzione avviene seguendo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta. Tramite lo stesso processo, per esempio, un utente non può contemporaneamente inserire caratteri e verificare la correttezza ortografica di quel che sta scrivendo. Nella maggior parte dei sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più percorsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta. Questa funzione è particolarmente utile sui sistemi multicore, in cui più thread possono essere eseguiti in parallelo. In un sistema che supporta i thread, il pcb viene esteso per includere informazioni su ogni thread. Per supportare i thread sono necessari anche altri cambiamenti nel sistema. Il Capitolo 4 è dedicato ai thread.

RAPPRESENTAZIONE DEI PROCESSI IN LINUX

Il blocco di controllo dei processi nel sistema operativo Linux è rappresentato dalla struttura C `task_struct`, che si trova nel file `<linux/sched.h>`, nella directory del codice sorgente del kernel. Questa struttura contiene una descrizione completa del processo, compreso il suo stato, informazioni sullo scheduling e sulla gestione della memoria, la lista dei file aperti, puntatori al processo padre e un elenco dei suoi figli e fratelli. (Il *padre* o *genitore* di un processo è il processo che lo ha creato; i *figli* sono i processi generati; i suoi *fratelli* sono processi con lo stesso padre). Alcuni dei campi sono i seguenti:

```
long state; /* stato del processo */

struct sched_entity se; /* informazioni per lo scheduling */

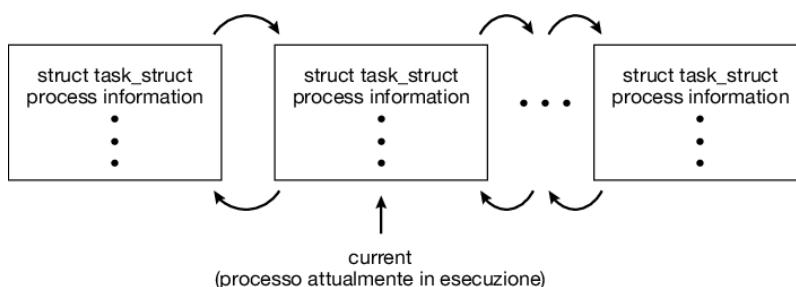
struct task_struct *parent; /* processo padre */

struct list_head children; /* processi figlio*/

struct files_struct *files; /* lista dei file aperti */

struct mm_struct *mm; /* spazio degli indirizzi del processo */
```

Per esempio, lo stato del processo è rappresentato dal campo `long state` in questa struttura. Nel kernel di Linux l'insieme dei processi attivi è rappresentato da una lista doppiamente concatenata di `task_struct`, e il kernel mantiene un puntatore di nome `current` al processo attualmente in esecuzione, come illustrato di seguito.



Per illustrare le manipolazioni eseguite dal kernel sui campi della struttura `task_struct` di uno specifico processo, supponiamo che il sistema debba cambiare lo stato del processo in esecuzione al valore `nuovo_stato`; se `current` punta, come detto poc'anzi, al processo attualmente in esecuzione, l'istruzione da eseguire è:

```
current->state = nuovo_stato;
```

3.2 Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nell'avere sempre un processo in esecuzione in modo da massimizzare l'utilizzo della cpu. L'obiettivo del time-sharing è di commutare l'uso della cpu tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo scheduler dei processi seleziona un processo da eseguire dall'insieme di quelli disponibili. Ogni core della cpu può eseguire un processo alla volta. In un sistema con un singolo core non verrà quindi mai eseguito più di un processo alla volta, mentre un sistema multicore può eseguire più processi contemporaneamente. Se vi sono più processi che core, i processi in eccesso dovranno attendere fino a quando un core diventa libero e può essere riassegnato. Il numero di processi in memoria in un dato istante è noto come il grado di multiprogrammazione.

Il bilanciamento tra gli obiettivi della multiprogrammazione e la condivisione del tempo richiede di tener conto anche del comportamento complessivo di un processo. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di i/o (*i/o bound*), o come avente una prevalenza d'elaborazione (*cpu bound*). Un processo *i/o bound* impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di i/o. Un processo *cpu bound*, viceversa, richiede operazioni di i/o con poca frequenza e impiega la maggior parte del proprio tempo nelle elaborazioni.

3.2.1 Code di scheduling

Entrando nel sistema, ogni processo è inserito in una coda di processi pronti e in attesa d'essere eseguiti, detta coda dei processi pronti (*ready queue*). Questa coda generalmente viene memorizzata come una lista concatenata: un'intestazione della coda dei processi pronti contiene i puntatori al primo pcb della lista, e ciascun pcb comprende un campo puntatore che indica il successivo processo contenuto nella coda.

Il sistema operativo ha anche altre code. Quando si assegna un core della cpu a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta di i/o. Supponiamo che il processo effettui una richiesta di i/o su un disco. Poiché i dispositivi periferici sono molto più lenti dei processori, il processo dovrà attendere che l' i/o diventi disponibile. I processi in attesa di un determinato evento, per esempio il completamento dell' i/o, vengono collocati in una coda di attesa, o *wait queue* (Figura 3.4).

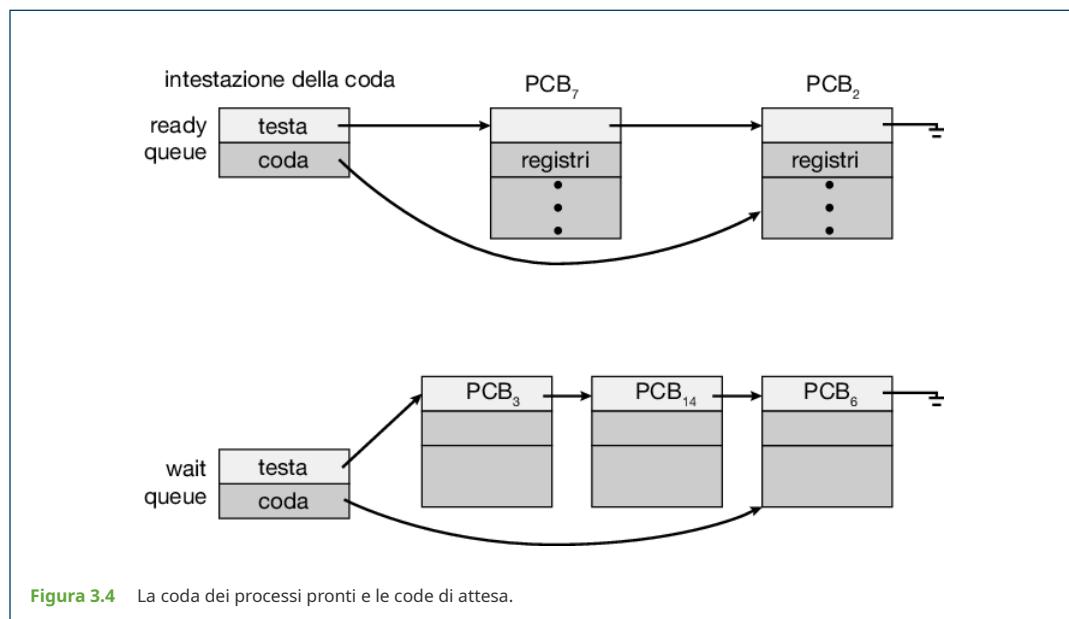
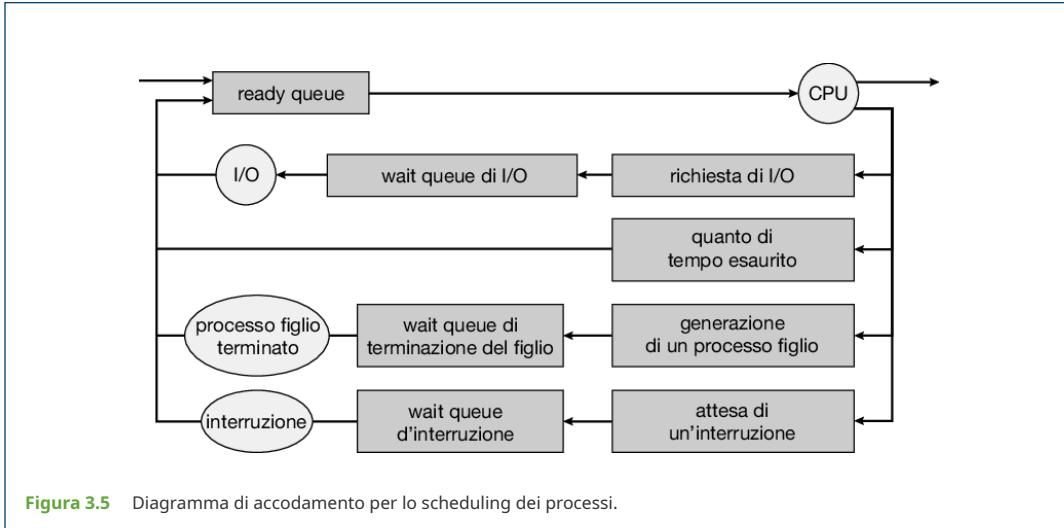


Figura 3.4 La coda dei processi pronti e le code di attesa.

Una comune rappresentazione dello scheduling dei processi è data da un diagramma di accodamento come quello illustrato nella Figura 3.5. Sono presenti due tipi di coda: la ready queue e un insieme di wait queue. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.



Un nuovo processo si colloca inizialmente nella ready queue, dove attende finché non è selezionato per essere eseguito (*dispatched*). Una volta che il processo è assegnato alla cpu ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

- il processo può emettere una richiesta di i/o e quindi essere inserito in una coda di i/o;
- il processo può creare un nuovo processo figlio e attenderne la terminazione;
- il processo può essere rimosso forzatamente dalla cpu a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di i/o o al termine del processo figlio, il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella ready queue. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene rimosso da tutte le code, e vengono deallocati il suo pcb e le varie risorse.

3.2.2 Scheduling della CPU

Nel corso della sua esistenza, un processo si sposta ripetutamente tra la coda dei processi pronti e diverse code di attesa. Il ruolo dello scheduler della cpu è selezionare un processo nella coda dei processi pronti e allocarlo a un core della cpu. Questa selezione deve essere effettuata frequentemente. Un processo i/o bound può restare in esecuzione solo per pochi millisecondi prima di attendere una richiesta di i/o. Un processo cpu bound, d'altro canto, richiede un core della cpu per tempi più lunghi, ma è improbabile che lo scheduler della cpu garantisca il core a un processo per un periodo prolungato. È invece probabile che lo scheduler sia progettato per togliere forzatamente la cpu a un processo in esecuzione e schedularne un altro processo da eseguire. Lo scheduling della cpu viene dunque eseguito almeno una volta ogni 100 millisecondi, anche se generalmente ciò avviene molto più frequentemente.

Alcuni sistemi operativi hanno una forma intermedia di scheduling, nota come swapping, la cui idea chiave è che a volte possa essere vantaggioso eliminare processi dalla memoria (e dalla contesa per la cpu), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta.

Questo schema si chiama avvicendamento dei processi in memoria (*swapping*), perché un processo può essere rimosso dalla memoria e collocato su disco (*swapped out*), dove viene salvato il suo stato corrente, e successivamente ricaricato in memoria da disco (*swapped in*), con il ripristino del suo stato. L'avvicendamento dei processi in memoria è in genere necessario solo quando la memoria è sovrautilizzata e deve essere liberata. Lo swapping è illustrato nel Capitolo 9.

3.2.3 Cambio di contesto

Come spiegato nel Paragrafo 1.2.1 le interruzioni forzano il sistema a sospendere il lavoro attuale della cpu per eseguire routine del kernel. Le interruzioni sono eventi comuni nei sistemi general-purpose. In presenza di una interruzione, il sistema deve salvare il contesto del processo corrente, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione. Il contesto è rappresentato all'interno del pcb del processo, e comprende i valori dei registri della cpu, lo stato del processo (si veda la Figura 3.2) e informazioni relative alla gestione della memoria. In termini generali, si esegue un salvataggio dello stato corrente della cpu, sia che essa esegua in modalità utente o in modalità di sistema; in seguito, si attiverà un corrispondente ripristino dello stato per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

Il passaggio della cpu a un nuovo processo implica il salvataggio dello stato del processo attuale e il ripristino dello stato del nuovo processo. Questa procedura è nota col nome di cambio di contesto (*context switch*) ed è mostrata nella Figura 3.6. Nell'evenienza di un cambio di contesto, il sistema salva il contesto del processo uscente nel suo pcb e carica il contesto del processo subentrante, salvato in precedenza. Il cambio di contesto è puro overhead, perché il sistema esegue solo operazioni volte alla gestione dei processi, e non alla computazione. Il tempo necessario varia da sistema a sistema, dipendendo dalla velocità della memoria, dal numero di registri da copiare e dall'esistenza di istruzioni macchina appropriate (per esempio, una singola istruzione per caricare o trasferire in memoria tutti i registri). In genere si tratta di qualche millisecondo.

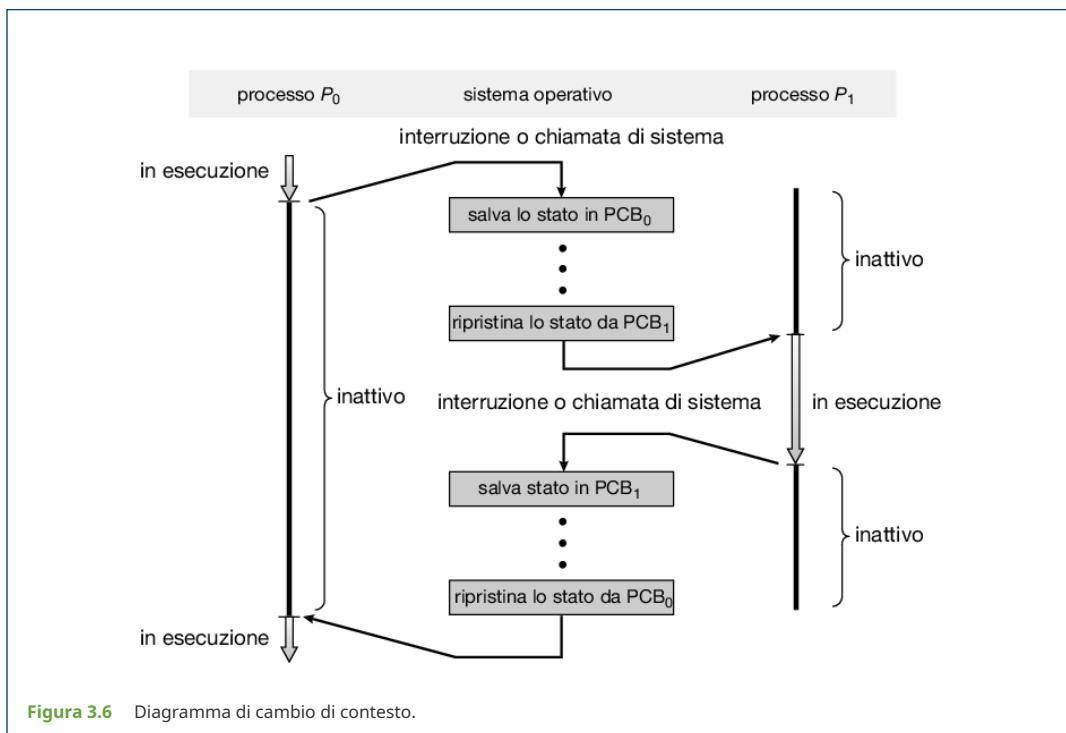


Figura 3.6 Diagramma di cambio di contesto.

La durata del cambio di contesto dipende molto dall'architettura; per esempio, alcune cpu sono dotate di più set di registri, quindi il cambio di contesto prevede la semplice modifica di un puntatore al gruppo di registri corrente. Naturalmente, se il numero dei processi attivi è maggiore di quello dei set di registri disponibili, il sistema rimedia copiando i dati dei registri nella e dalla memoria, come prima. Inoltre, più complesso è il sistema operativo, più lavoro si deve svolgere durante un cambio di contesto. Come vedremo nel Capitolo 9, l'uso di tecniche avanzate di gestione della memoria può richiedere lo spostamento di ulteriori dati a ogni cambio di contesto. Per esempio, si deve preservare lo spazio d'indirizzi del processo corrente mentre si prepara lo spazio per il processo successivo. Il modo in cui si preserva tale spazio e la relativa quantità di lavoro dipendono dal metodo di gestione della memoria del sistema operativo.

MULTITASKING NEI SISTEMI PER DISPOSITIVI MOBILI

A causa dei vincoli imposti sui dispositivi mobili, le prime versioni di iOS non fornivano il multitasking per le applicazioni utente; una sola applicazione veniva eseguita in foreground, mentre tutte le altre erano sospese. I processi del sistema operativo erano invece gestiti in multitasking, perché essendo scritti da Apple si comportavano in maniera corretta. A partire da iOS 4 Apple offre una forma limitata di multitasking per le applicazioni utente, consentendo così all'applicazione in foreground di funzionare contemporaneamente a più applicazioni in background. Su un dispositivo mobile, l'**applicazione in foreground** è l'applicazione aperta al momento e che appare sul display. L'**applicazione in background** rimane in memoria, ma non occupa lo schermo del display. L'API iOS 4 fornisce il supporto per il multitasking, permettendo così a un processo di rimanere in esecuzione in background senza essere sospeso. Tuttavia, questa possibilità è limitata e disponibile soltanto per un numero ridotto di tipologie di applicazioni. Quando l'hardware dei dispositivi mobili ha iniziato a offrire memorie più grandi, diversi core di elaborazione e una maggiore durata della batteria, le versioni di iOS hanno iniziato a supportare funzionalità più complete e meno restrittive per il multitasking. Per esempio, lo schermo più grande sui tablet iPad consente di eseguire due app in primo piano contemporaneamente, una tecnica nota come **split-screen**.

Fin dalle sue origini, Android ha supportato il multitasking e non ha posto vincoli sulle tipologie di applicazioni che possono essere eseguite in background. Un'applicazione che voglia richiedere l'elaborazione mentre è in background deve utilizzare un servizio, un componente applicativo separato che viene eseguito per conto del processo in background. Si consideri un'applicazione di streaming audio: se l'applicazione si sposta in background, il servizio continua a inviare i file audio al driver di periferica audio per suo conto. Il servizio continuerà a funzionare anche se l'applicazione in background viene sospesa. I servizi non hanno un'interfaccia utente e hanno un ingombro di memoria modesto, fornendo così una tecnica efficace per il multitasking in un ambiente mobile.

3.3 Operazioni sui processi

Nella maggior parte dei sistemi i processi si possono eseguire in modo concorrente, e si devono creare e cancellare dinamicamente; a tal fine il sistema operativo deve offrire un meccanismo che permetta di creare e terminare un processo. Nel presente paragrafo si esplorano quei meccanismi coinvolti nella creazione dei processi, in particolare per i sistemi unix e Windows.

3.3.1 Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Come menzionato in precedenza, il processo creante si chiama processo genitore (o padre), mentre il nuovo processo si chiama processo figlio. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un albero di processi.

La maggior parte dei sistemi operativi (compresi unix, Linux e Windows) identifica un processo per mezzo di un numero univoco, solitamente un intero, detto identificatore del processo o pid (*process identifier*). Il pid fornisce un valore univoco per ogni processo del sistema e può essere usato come indice per accedere a vari attributi di un processo all'interno del kernel.

La Figura 3.7 mostra un tipico albero dei processi del sistema operativo Linux, con il nome e il pid di ogni processo. (Qui usiamo genericamente il termine *processo* anche se in Linux si preferisce il termine *task*). Il processo `systemd` (che ha sempre pid uguale a 1) svolge il ruolo di processo padre di tutti i processi utente. Una volta che il sistema si è avviato, il processo `systemd` può creare vari processi utente, per esempio un server web o per la stampa, un server `ssh`, e processi simili. Nella Figura 3.7 vediamo due figli di `systemd` – `logind` e `sshd`. Il processo `logind` è responsabile della gestione dei client che si collegano direttamente al sistema. In questo esempio, un client ha effettuato l'accesso e sta utilizzando la shell bash, a cui è stato assegnato pid 8416. Utilizzando l'interfaccia a riga di comando `bash`, questo utente ha creato il processo `ps` e l'editor `vim`. Il processo `sshd` è responsabile della gestione dei client che si connettono al sistema tramite `ssh` (abbreviazione di *secure shell*).

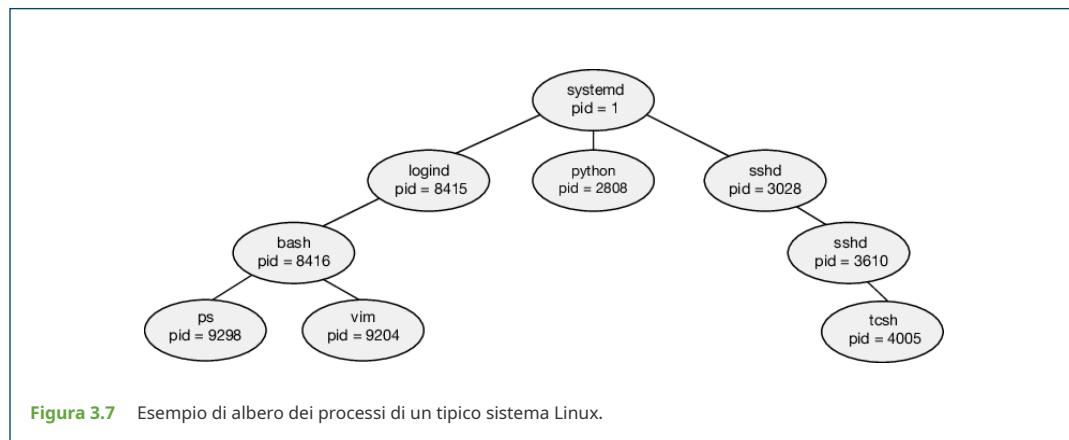


Figura 3.7 Esempio di albero dei processi di un tipico sistema Linux.

Nei sistemi unix e Linux si può ottenere l'elenco dei processi tramite il comando `ps`. Per esempio, digitando

```
ps -el
```

si otterranno informazioni complete su tutti i processi attualmente attivi nel sistema; si può costruire un albero come quello nella Figura 3.7 identificando ricorsivamente i processi genitore fino a giungere a `systemd`. I sistemi Linux forniscono anche il comando `pstree`, che mostra la struttura ad albero di tutti i processi nel sistema.

In generale, quando un processo crea un processo figlio, quest'ultimo avrà bisogno di determinate risorse (tempo d'elaborazione, memoria, file, dispositivi di I/O) per eseguire il proprio compito. Un processo figlio può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo genitore. Il processo genitore può avere la necessità di spartire le proprie risorse tra i suoi processi figli, oppure può essere in grado di condividerne alcune, come la memoria o i file, tra più processi figli. Limitando le risorse di un processo figlio a un sottoinsieme di risorse del processo genitore si può evitare che un processo sovraccarichi il sistema creando troppi processi figlio.

I sistemi UNIX tradizionali identificano il processo init come il padre di tutti i processi figli. Il processo init (noto anche come **System V** init) è il primo processo creato all'avvio del sistema e gli viene assegnato un pid pari a 1. Su un albero dei processi simile a quello mostrato nella Figura 3.7, init si trova alla radice.

I sistemi Linux adottarono inizialmente l'approccio di System V init, ma le distribuzioni recenti lo hanno sostituito con systemd. Come descritto nel Paragrafo 3.3.1, systemd è il processo iniziale del sistema, come System V init, ma è molto più flessibile e può fornire più servizi di init.

Oltre alle varie risorse fisiche e logiche, il processo genitore può passare al processo figlio i dati di inizializzazione. Per esempio, si consideri un processo che serva a mostrare i contenuti di un file – diciamo, il file `hw1.c` – sullo schermo di un terminale. Esso riceverà dal genitore, al momento della sua creazione, il nome del file `hw1.c` in ingresso, che userà per aprire e leggere il contenuto del file; potrà anche ricevere il nome del dispositivo al quale inviare i dati in uscita. In alternativa, alcuni sistemi operativi passano risorse ai processi figli, nel qual caso il nostro processo potrebbe ottenere come risorse due file aperti, cioè `hw1.c` e il dispositivo terminale, potendo così semplicemente trasferire il dato fra i due.

Quando un processo ne crea uno nuovo, per quel che riguarda l'esecuzione ci sono due possibilità:

1. il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli;
2. il processo genitore attende che alcuni o tutti i suoi processi figli terminino.

Ci sono due possibilità anche per quel che riguarda lo spazio d'indirizzi del nuovo processo:

1. il processo figlio è un duplicato del processo genitore (ha gli stessi programma e dati del genitore);
2. nel processo figlio si carica un nuovo programma.

Per illustrare queste differenze, consideriamo dapprima il sistema operativo unix. In unix, come abbiamo visto, ogni processo è identificato dal proprio identificatore di processo, un intero univoco. Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare senza difficoltà con il proprio processo figlio. Entrambi i processi (genitore e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema `fork()`, con una differenza: la chiamata di sistema `fork()` riporta il valore zero nel nuovo processo (il figlio), ma riporta l'identificatore del processo figlio (il pid diverso da zero) nel processo genitore.

Generalmente, dopo una chiamata di sistema `fork()`, uno dei due processi impiega una chiamata di sistema `exec()` per sostituire lo spazio di memoria del processo con un nuovo programma. La chiamata di sistema `exec()` carica in memoria un file binario, cancellando l'immagine di memoria del programma contenente la stessa chiamata di sistema `exec()`, quindi avvia la sua esecuzione. In questo modo i due processi possono comunicare e poi procedere in modo diverso. Il processo genitore può anche generare più processi figli, oppure, se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la chiamata di sistema `wait()` per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio. Poiché la chiamata `exec()` sovrappone lo spazio di indirizzi del processo con un nuovo programma, essa non restituisce il controllo a meno che non si verifichi un errore.

Il programma C della Figura 3.8 illustra le chiamate di sistema unix descritte precedentemente. Abbiamo due processi distinti ciascuno dei quali esegue una copia dello stesso programma. Il valore assegnato alla variabile `pid` è zero nel processo figlio, e un numero intero maggiore di zero (il pid del processo figlio) nel processo genitore. Il processo figlio eredita privilegi, attributi di scheduling e alcune risorse, come i file aperti, dal processo genitore. Usando la chiamata di sistema `execvp()` – una versione della chiamata di sistema `exec()` – il processo figlio sovrappone il proprio spazio d'indirizzi con il comando `/bin/ls` di unix (che si usa per ottenere l'elenco del contenuto di una directory). Eseguendo la chiamata di sistema `wait()`, il processo genitore attende che il processo figlio termini. Quando ciò accade (implicitamente o esplicitamente mediante l'invocazione di `exit()`), il processo genitore chiude la propria fase d'attesa dovuta alla chiamata di sistema `wait()` e termina usando la chiamata di sistema `exit()`. Questo schema è illustrato nella Figura 3.9.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid = fork();
    if (pid == 0)
        execvp("/bin/ls", NULL);
    else
        wait();
}
```

```

pid_t pid;

/* genera un nuovo processo */

pid = fork();

if (pid < 0) { /* errore */

fprintf(stderr, "generazione del nuovo processo fallita");

return 1;

}

else if (pid == 0) { /* processo figlio */

execvp("/bin/ls", "ls", NULL);

}

else /* processo genitore */

/* il genitore attende il completamento del figlio */

wait(NULL);

printf("il processo figlio ha terminato");

}

return 0;
}

```

Figura 3.8 Creazione di un processo separato utilizzando la chiamata di sistema fork() di unix.

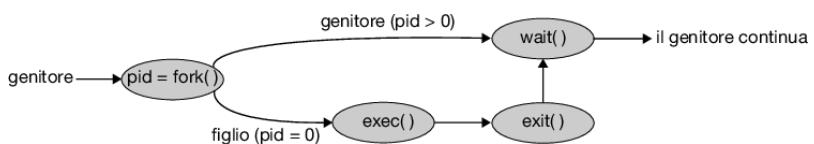


Figura 3.9 Creazione di un processo utilizzando la chiamata di sistema fork().

Naturalmente, non c'è modo di impedire al figlio di non invocare `exec()` e restare in esecuzione come una copia del processo genitore. In questo scenario, il genitore e il figlio sono processi concorrenti che eseguono lo stesso codice. Poiché il figlio è una copia del genitore, ogni processo ha la propria copia dei dati.

Come altro esempio consideriamo la creazione dei processi in Windows. Nella api di Windows la funzione `CreateProcess()` è simile alla `fork()` di unix, poiché serve a generare un figlio da un processo genitore. Mentre però `fork()` passa in eredità al figlio lo spazio degli indirizzi del genitore, `CreateProcess()` richiede il caricamento di un programma specificato nello spazio degli indirizzi del processo figlio al momento della sua creazione. Inoltre, mentre `fork()` non richiede parametri, `CreateProcess()` se ne aspetta non meno di dieci.

Il programma nella Figura 3.10, scritto in C, illustra la funzione `CreateProcess()`; essa genera un processo figlio che carica l'applicazione `mspaint.exe`. In questo esempio, i dieci parametri passati a `CreateProcess()` hanno in molti casi il loro valore di default.

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    /* alloca la memoria */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* genera processo figlio */
    if (!CreateProcess(NULL, /* usa riga di comando */

        "C:\\WINDOWS\\system32\\mspaint.exe", /* riga di comando */

        NULL, /* non eredita l'handle del processo */

        NULL, /* non eredita l'handle del thread */

        FALSE, /* disattiva l'ereditarieta' degli handle */

        0, /* nessun flag di creazione */

        NULL, /* usa il blocco ambiente del genitore */

        NULL, /* usa la directory esistente del genitore */

        &si,
        &pi))
    {
        fprintf(stderr, "generazione del nuovo processo fallita");

        return -1
    }

    /* il genitore attende il completamento del figlio */
    WaitForSingleObject(pi.hProcess, INFINITE);

    printf("il processo figlio ha terminato");

    /* rilascia gli handle */
}
```

```

        CloseHandle(pi.hProcess);

        CloseHandle(pi.hThread);

    }

}

```

Figura 3.10 Creazione di un nuovo processo con la api Windows.

I lettori interessati alla creazione e gestione dei processi nella api di Windows possono consultare i riferimenti bibliografici alla fine del capitolo.

I due parametri passati a `CreateProcess()` sono istanze delle strutture `STARTUPINFO` e `PROCESS_INFORMATION`. La prima specifica molte proprietà del nuovo processo, come la dimensione della finestra e il suo aspetto, e i riferimenti – detti anche handle – ai file di input e output standard. La seconda contiene un handle e gli identificatori per il nuovo processo e il suo thread. La funzione `ZeroMemory()` è invocata per allocare memoria per le due strutture prima di passare a `CreateProcess()`.

I primi due parametri passati a `CreateProcess()` sono il nome dell'applicazione e i parametri della riga di comando. Se il nome dell'applicazione è `NULL` (come nell'esempio della figura), è il parametro della riga di comando a specificare quale applicazione caricare. Nell'esempio, si carica l'applicazione `mspaint.exe` di Microsoft Windows. Al di là dei primi due parametri, l'esempio usa i parametri di default per far ereditare gli handle del processo e del thread, e per specificare l'assenza di flag di creazione. Il figlio adotta inoltre il blocco ambiente del genitore e la sua directory iniziale. Infine, si passano i due puntatori alle strutture `STARTUPINFO` e `PROCESS_INFORMATION` create all'inizio. Nella precedente Figura 3.9 il processo genitore attende la terminazione del figlio invocando la chiamata di sistema `wait()`. L'equivalente Windows è `WaitForSingleObject()`, a cui si passa l'handle del processo figlio – `pi.hProcess` – del cui completamento si è in attesa. A terminazione del figlio avvenuta, il controllo ritorna al processo genitore, al punto immediatamente successivo alla chiamata `WaitForSingleObject()`.

3.3.2 Terminazione di un processo

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema `exit()`; a questo punto, il processo figlio può riportare un'informazione di stato al processo genitore, che la riceve attraverso la chiamata di sistema `wait()`. Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'i/o, sono liberate dal sistema operativo.

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un'opportuna chiamata di sistema (per esempio `TerminateProcess()` in Windows). Generalmente solo il genitore del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un genitore deve conoscere le identità dei propri figli per terminarli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi, tra i quali i seguenti.

- Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate. Ciò richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli.
- Il compito assegnato al processo figlio non è più richiesto.
- Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza.

In alcuni sistemi, se un processo termina si devono terminare anche i suoi figli, indipendentemente dal fatto che la terminazione del genitore sia stata normale o anormale. Si parla di terminazione a cascata, una procedura avviata di solito dal sistema operativo.

Per illustrare un esempio di esecuzione e terminazione di un processo si consideri che nel sistema operativo Linux come in unix un processo può terminare per mezzo della chiamata di sistema `exit()`, fornendo uno stato di uscita (`exit status`) come parametro:

```

/* uscita con stato 1 */

exit(1);

```

Di fatto, in caso di terminazione normale, `exit()` può essere chiamato direttamente (come mostrato sopra) o indirettamente (tramite un'istruzione `return` nel `main()`).

Un processo genitore può attendere la terminazione di un processo figlio utilizzando la chiamata di sistema `wait()`, a cui viene passato un parametro che permette al genitore di ottenere lo stato di uscita del figlio. Questa chiamata di sistema restituisce anche l'id di processo del figlio terminato in modo che il genitore possa sapere di quale dei suoi figli si tratta:

```
pid_t pid;  
  
int status;  
  
pid = wait(&status);
```

Quando un processo termina, le sue risorse vengono deallocate dal sistema operativo. Tuttavia, la sua voce nella tabella dei processi deve rimanere fino a quando il padre chiama `wait()`, perché la tabella dei processi contiene lo stato di uscita del processo.

Un processo che è terminato, ma il cui genitore non ha ancora chiamato la `wait()`, è detto processo zombie. Tutti i processi passano in questo stato quando terminano, ma in genere rimangono zombie solo per breve tempo. Una volta che il genitore chiama la `wait()` il pid del processo zombie e la sua voce nella tabella dei processi vengono rilasciati.

Consideriamo ora che cosa accadrebbe se un genitore terminasse senza invocare la `wait()`, lasciando così orfani i suoi figli. Linux e unix affrontano questa situazione assegnando al processo `systemd` il ruolo di nuovo genitore dei processi orfani. (Ricordiamo dal Paragrafo 3.3.1 che il processo `init` è la radice della gerarchia dei processi nei sistemi unix). Il processo `init` invoca periodicamente `wait()`, consentendo in tal modo di raccogliere lo stato di uscita di qualsiasi processo orfano e rilasciando il suo pid e la relativa voce nella tabella dei processi.

Nonostante la maggior parte dei sistemi Linux abbia sostituito `init` con `systemd`, quest'ultimo può assumere lo stesso ruolo, con la differenza che Linux permette anche ad altri processi di ereditare i processi orfani e gestirne la terminazione.

3.3.2.1 Gerarchia dei processi Android

A causa dei vincoli imposti dalle risorse, come la memoria limitata, i sistemi operativi mobili possono aver bisogno di terminare processi esistenti per recuperare risorse di sistema. Anziché terminare un processo arbitrario, Android ha definito una gerarchia di importanza dei processi. Quando il sistema deve terminare un processo per rendere disponibili le risorse per un processo nuovo o più importante, esso effettua la scelta di quale processo terminare in base all'ordine crescente di importanza. Dal più al meno importante, la classificazione gerarchica dei processi è la seguente.

- Processo in primo piano: il processo visibile sullo schermo, che rappresenta l'applicazione con cui l'utente sta attualmente interagendo.
- Processo visibile: un processo che non è direttamente visibile in primo piano, ma che sta eseguendo un'attività a cui il processo in primo piano fa riferimento (vale a dire, un processo che esegue un'attività il cui stato è visualizzato dal processo in primo piano).
- Processo di servizio: un processo simile a un processo in background, ma che sta eseguendo un'attività evidente all'utente (come lo streaming di musica).
- Processo in background: un processo che può svolgere un'attività non evidente all'utente.
- Processo vuoto: un processo che non contiene componenti attive associate a un'applicazione.

Se occorre recuperare risorse di sistema, Android interrompe prima i processi vuoti, poi i processi in background, e così via. Ai processi viene assegnata una valutazione di importanza e Android assegna a un processo la posizione più alta possibile. Per esempio, se un processo fornisce un servizio ed è anche visibile, gli verrà assegnata la classificazione di processo visibile più importante.

Inoltre, le prassi di sviluppo Android suggeriscono di seguire le linee guida del ciclo di vita dei processi, che prevedono che lo stato di un processo sia salvato prima della sua terminazione e ripristinato quando l'utente ritorna all'applicazione.

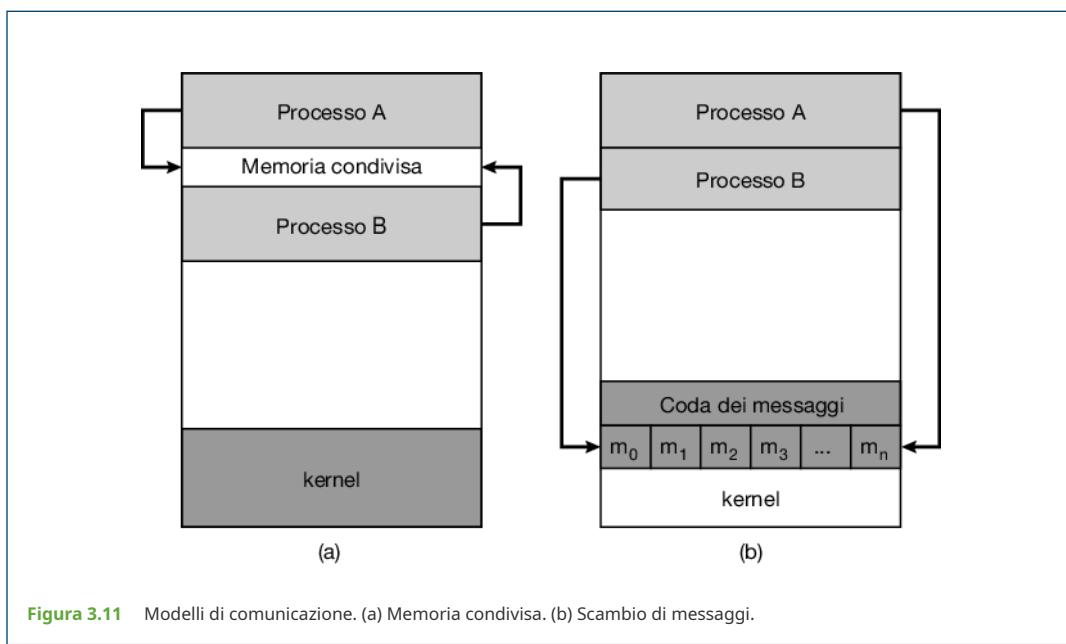
3.4 Comunicazione tra processi

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è *indipendente* se non può influire su altri processi del sistema o subirne l'influsso. Chiaramente, un processo che non condivide dati (temporanei o permanenti) con altri processi è indipendente. D'altra parte un processo è *cooperante* se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivide dati con altri processi è un processo cooperante.

Un ambiente che consente la cooperazione tra processi può essere utile per diverse ragioni.

- Condivisione d'informazioni. Poiché più utenti possono essere interessati alle stesse informazioni (per esempio un file condiviso) è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse.
- Velocizzazione del calcolo. Alcune attività d'elaborazione sono realizzabili più rapidamente se si suddividono in sottoattività eseguibili in parallelo. Un'accelerazione di questo tipo è ottenibile solo se il calcolatore dispone di più core di elaborazione.
- Modularità. Può essere utile la costruzione di un sistema modulare, che suddivide le funzioni di sistema in processi o thread distinti (si veda il Capitolo 2).

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di comunicazione tra processi (ipc, *interprocess communication*). I modelli fondamentali della comunicazione tra processi sono due: a memoria condivisa e a scambio di messaggi. Nel modello a memoria condivisa si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona. Nel secondo modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti. I due modelli sono messi a confronto nella Figura 3.11.



Nei sistemi operativi sono diffusi entrambi i modelli; spesso coesistono in un unico sistema. Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti. Lo scambio di messaggi è anche più facile da implementare, rispetto alla memoria condivisa, in un sistema distribuito.

(Anche se ci sono sistemi che forniscono una memoria condivisa distribuita, non verranno considerati in questo testo). La memoria condivisa può essere più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono gestiti alla stregua di ordinari accessi in memoria che non richiedono l'assistenza del kernel.

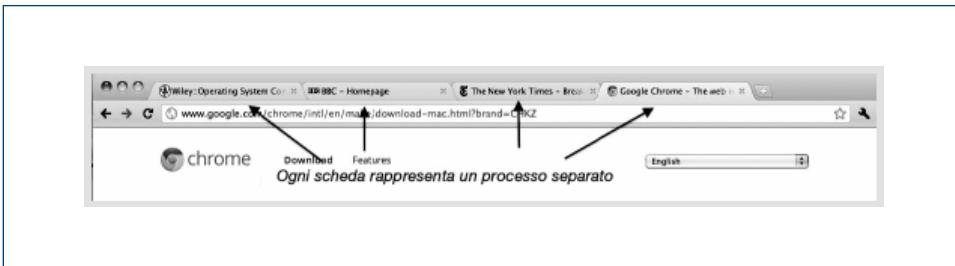
Nei Paragrafi 3.5 e 3.6 esploriamo i sistemi a memoria condivisa e a scambio di messaggi in modo più dettagliato.

3.5 IPC in sistemi a memoria condivisa

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca; gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi. Si ricordi che, normalmente, il sistema operativo tenta di impedire un processo l'accesso alla memoria di altri processi. La condivisione della memoria richiede che due o più processi raggiungano un accordo per superare questo limite, in modo da poter comunicare tramite scritture e letture dell'area condivisa. Il tipo e la collocazione dei dati sono determinati dai processi, e rimangono al di fuori del controllo del sistema operativo. I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente.

ARCHITETTURA MULTIPROCESSO – IL BROWSER CHROME

Per offrire un'esperienza di navigazione ricca e dinamica, in molti siti web sono presenti contenuti attivi come JavaScript, Flash e html5. Purtroppo queste applicazioni web possono anche contenere alcuni bug software che talvolta rallentano i tempi di risposta o provocano il crash del browser web. Non si trattrebbe di un grave problema in un browser che visualizza il contenuto di un solo sito web, ma i browser più moderni supportano la navigazione a schede, che permette di aprire in una singola istanza del browser diversi siti web allo stesso tempo, ogni sito in una scheda separata. Per passare da un sito all'altro, l'utente deve solo fare clic sulla scheda appropriata. Questa disposizione è illustrata di seguito:



Un problema di questo approccio è che se si blocca un'applicazione web contenuta in una scheda anche l'intero processo, e quindi tutte le schede che visualizzano altri siti web, si blocca.

Il browser Chrome di Google è stato progettato per affrontare tale problema con l'utilizzo di un'architettura multiprocess. Chrome identifica tre diversi tipi di processi: browser, renderer e plug-in.

- Il processo del **browser** è responsabile della gestione dell'interfaccia utente e dell'i/o da disco e da rete. All'avvio di Chrome viene creato un nuovo processo browser; verrà creato soltanto un processo browser per ogni istanza di Chrome.
- I processi **renderer** contengono la logica per il rendering di pagine web, e dunque la logica per la gestione di html, Javascript, immagini e così via. Come regola generale, viene creato un nuovo processo di rendering per ciascun sito web aperto in una nuova scheda. Diversi processi renderer possono quindi essere attivi contemporaneamente.
- Viene creato un processo **plug-in** per ogni tipo di plug-in (come Flash o QuickTime) in uso. I processi plug-in contengono il codice per il plug-in e del codice aggiuntivo che permette al plug-in di comunicare con i processi renderer associati e con il processo del browser.

Il vantaggio di questo approccio multiprocesso è che i siti web vengono eseguiti in modo che ognuno sia isolato dall'altro. Se un sito web genera un crash viene influenzato solo il suo processo di rendering, mentre tutti gli altri processi restano intatti. Inoltre, i processi renderer vengono eseguiti in una **sandbox**, il che significa che l'accesso al disco e alla rete sono limitati, in modo da minimizzare gli effetti negativi di eventuali exploit di sicurezza.

Per illustrare il concetto di cooperazione tra processi si consideri il problema del produttore/consumatore; tale problema è un comune paradigma per processi cooperanti. Un processo produttore produce informazioni che sono consumate da un processo consumatore. Un compilatore, per esempio, può produrre del codice assembly consumato da un assemblatore; quest'ultimo, a sua volta, può produrre moduli oggetto consumati dal loader. Il problema del produttore/consumatore è anche un'utile metafora del paradigma client-server. Si pensa in genere al server come al produttore e al client come al consumatore. Per esempio, un server web produce (ossia, fornisce) pagine html e immagini, consumate (ossia, lette) dal client, cioè il browser web che le richiede.

Una possibile soluzione del problema del produttore/consumatore si basa sulla memoria condivisa. L'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi. Il produttore potrà allora produrre un'unità mentre il consumatore ne consuma un'altra. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità non ancora prodotta.

Si possono utilizzare due tipi di buffer. Quello illimitato non pone limiti pratici alla dimensione del buffer. Il consumatore può dover attendere nuovi oggetti, ma il produttore può sempre produrne. Il problema del produttore e del consumatore con buffer limitato presuppone una dimensione fissa del buffer in questione. In questo caso, il consumatore deve attendere se il buffer è vuoto; viceversa, il produttore deve attendere se il buffer è pieno.

Consideriamo più attentamente in che modo il buffer limitato illustra la comunicazione tra processi con memoria condivisa. Le variabili seguenti risiedono in una zona di memoria condivisa sia dal produttore sia dal consumatore.

```
#define BUFFER_SIZE 10

typedef struct {

    ...

}elemento;

elemento buffer[BUFFER_SIZE];

int in = 0;

int out = 0;
```

Il buffer condiviso è realizzato come un array circolare con due puntatori logici: `in` e `out`. La variabile `in` indica la successiva posizione libera nel buffer; `out` indica la prima posizione piena nel buffer. Il buffer è vuoto se `in == out`; è pieno se `((in + 1) % BUFFER_SIZE) == out`.

Il codice per il processo produttore è illustrato nella Figura 3.12, quello per il processo consumatore nella Figura 3.13. Il processo produttore ha una variabile locale `next_produced` contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile locale `next_consumed` in cui si memorizza l'elemento da consumare.

```
item next_produced;

while (true) {

    /* produce un elemento in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out); /* non fa niente */

    buffer[in] = next_produced;
```

```
    in = (in + 1) % BUFFER_SIZE;  
}
```

Figura 3.12 Processo produttore con l'utilizzo della memoria condivisa.

```
item next_consumed;  
  
while (true) {  
  
    while (in == out)  
  
        ; /* non fa niente */  
  
    next_consumed = buffer[out];  
  
    out = (out + 1) % BUFFER_SIZE;  
  
    /* consuma l'elemento in next_consumed */  
  
}
```

Figura 3.13 Processo consumatore con l'utilizzo della memoria condivisa.

Questo metodo ammette un massimo di `BUFFER_SIZE-1` elementi contemporaneamente presenti nel buffer. Proponiamo come esercizio per il lettore la stesura di un algoritmo che permetta la presenza contemporanea di `BUFFER_SIZE` oggetti. Nel Paragrafo 3.7.1 si illustrerà la api posix per la memoria condivisa.

Una questione ignorata dalla precedente analisi è il caso in cui sia il produttore sia il consumatore tentino di accedere al buffer concorrentemente. Nei Capitoli 6 e 7 si vedrà come sia possibile implementare efficacemente la sincronizzazione tra processi cooperanti nel modello a memoria condivisa.

3.6 IPC in sistemi a scambio di messaggi

Nel Paragrafo 3.5 si è parlato di come possa avvenire la comunicazione tra processi cooperanti in un ambiente a memoria condivisa. Per essere applicato, lo schema proposto richiede che tali processi condividano l'accesso a una zona di memoria e che il codice per la realizzazione e la gestione della memoria condivisa sia scritto esplicitamente dal programmatore che crea l'applicazione. Un altro modo in cui il sistema operativo può ottenere i medesimi risultati consiste nel fornire ai processi appositi strumenti per lo scambio di messaggi.

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi. È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete. Per esempio, una chat sul Web potrebbe essere implementata tramite scambio di messaggi fra i vari partecipanti.

Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni: `send(message)`, cioè “invia messaggio”, e `receive(message)`, cioè “ricevi messaggio”. I messaggi possono avere lunghezza fissa o variabile. Nel primo caso, l'implementazione a livello del sistema è elementare, ma programmare applicazioni diviene più complicato. Nel secondo caso, l'implementazione del meccanismo a livello di sistema è più complessa, mentre la programmazione utente risulta semplificata. Compromessi di questo tipo si riscontrano spesso nella progettazione dei sistemi operativi.

Se i processi *P* e *Q* vogliono comunicare devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un canale di comunicazione (*communication link*), realizzabile in molti modi. In questo paragrafo non si tratta della realizzazione fisica del canale (come la memoria condivisa, i bus o le reti, illustrati nel Capitolo 19) ma della sua realizzazione logica. Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni `send()` e `receive()`:

- comunicazione diretta o indiretta;
- comunicazione sincrona o asincrona;
- gestione automatica o esplicita del buffer.

Le questioni legate a ciascuna di tali caratteristiche vengono illustrate in seguito.

3.6.1 Naming

Per comunicare, i processi devono disporre della possibilità di far riferimento ad altri processi; a tale scopo è possibile servirsi di una comunicazione diretta oppure indiretta.

Con la comunicazione diretta ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione. In questo schema le funzioni primitive `send()` e `receive()` si definiscono come segue:

```
send(P, messaggio), invia messaggio al processo P;  
receive(Q, messaggio), riceve messaggio dal processo Q.
```

All'interno di questo schema un canale di comunicazione ha le seguenti caratteristiche:

- tra ogni coppia di processi che intendono comunicare si stabilisce automaticamente un canale; i processi devono conoscere solo la reciproca identità;
- un canale è associato esattamente a due processi;
- esiste esattamente un canale tra ciascuna coppia di processi.

Questo schema ha una *simmetria* nell'indirizzamento, vale a dire che per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda. Una variante di questo schema si avvale dell'*asimmetria* nell'indirizzamento: soltanto il trasmittente nomina il ricevente, mentre il ricevente non ha bisogno di nominare il trasmittente. In questo schema le primitive `send()` e `receive()` si definiscono come segue:

```
• send(P, messaggio), invia messaggio al processo P;  
• receive(id, messaggio), riceve un messaggio da qualsiasi processo; nella variabile id si ottiene il nome del processo con cui è avvenuta la comunicazione.
```

Entrambi gli schemi, simmetrico e asimmetrico, hanno lo svantaggio di una limitata modularità delle risultanti definizioni dei processi. La modifica del nome di un processo può infatti implicare la necessità di un riesame di tutte le altre definizioni dei processi, individuando tutti i riferimenti al vecchio nome allo scopo di sostituirli con il nuovo. In generale, tali cablature (*hard coding*) di informazioni nel codice sono meno vantaggiose di soluzioni indirette, descritte nel seguito.

Con la comunicazione indiretta i messaggi s'inviano a delle porte o mailbox, che li ricevono. Una mailbox si può considerare in modo astratto come un oggetto in cui i processi possono introdurre e prelevare messaggi, ed è identificata in modo unico. Per

l'identificazione di una porta le code di messaggi posix usano per esempio un valore intero. In questo schema un processo può comunicare con altri processi tramite un certo numero di mailbox e due processi possono comunicare solo se condividono una mailbox. Le primitive `send()` e `receive()` si definiscono come segue:

- `send(A, messaggio)`, invia messaggio alla mailbox A;
- `receive(A, messaggio)`, riceve un messaggio dalla mailbox A.

In questo schema un canale di comunicazione ha le seguenti caratteristiche:

- tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa mailbox;
- un canale può essere associato a più di due processi;
- tra ogni coppia di processi comunicanti possono esserci più canali diversi, ciascuno corrispondente a una mailbox.

A questo punto, si supponga che i processi P_1 , P_2 e P_3 condividano la mailbox A. Il processo P_1 invia un messaggio ad A, mentre sia P_2 sia P_3 eseguono una `receive()` da A. Sorge il problema di sapere quale processo riceverà il messaggio.

La soluzione dipende dallo schema prescelto:

- si fa in modo che un canale sia associato al massimo a due processi;
- si consente l'esecuzione di un'operazione `receive()` a un solo processo alla volta;
- si consente al sistema di decidere arbitrariamente quale processo riceverà il messaggio (che sarà ricevuto da P_2 o da P_3 , ma non da entrambi). Il sistema può anche definire un algoritmo per selezionare quale processo riceverà il messaggio (specificando per esempio uno schema, detto *round robin*, secondo il quale i processi ricevono i messaggi a turno) e può comunicare l'identità del ricevente al trasmittente.

Una mailbox può appartenere al processo o al sistema. Se appartiene a un processo, cioè fa parte del suo spazio d'indirizzi, occorre distinguere tra il proprietario, che può soltanto ricevere messaggi tramite la mailbox, e l'utente, che può solo inviare messaggi alla mailbox. Poiché ogni mailbox ha un unico proprietario, non può sorgere confusione su chi debba ricevere un messaggio inviato a una determinata mailbox. Quando un processo che possiede una mailbox termina, questa scompare, e qualsiasi processo che invii un messaggio alla mailbox di un processo già terminato ne deve essere informato.

Invece, una mailbox posseduta dal sistema operativo ha una vita autonoma, è indipendente e non è legata ad alcun processo particolare. Il sistema operativo offre un meccanismo che permette a un processo le seguenti operazioni:

- creare una nuova mailbox;
- inviare e ricevere messaggi tramite la mailbox;
- rimuovere una mailbox.

Il processo che crea una nuova mailbox è il proprietario predefinito della mailbox, e inizialmente è l'unico processo che può ricevere messaggi attraverso questa mailbox. Tuttavia, il diritto di proprietà e il diritto di ricezione si possono passare ad altri processi per mezzo di idonee chiamate di sistema. Naturalmente questa disposizione potrebbe dar luogo all'esistenza di più riceventi per ciascuna mailbox.

3.6.2 Sincronizzazione

La comunicazione tra processi avviene attraverso chiamate delle primitive `send()` e `receive()`. Ci sono diverse possibilità nella definizione di ciascuna primitiva. Lo scambio di messaggi può essere sincrono (o bloccante) oppure asincrono (o non bloccante). (Per tutto il testo incontreremo i concetti di comportamento sincrono e asincrono in relazione a vari algoritmi del sistema operativo).

- Invio sincrono. Il processo che invia il messaggio si blocca nell'attesa che il processo ricevente, o la mailbox, riceva il messaggio.
- Invio asincrono. Il processo invia il messaggio e riprende la propria esecuzione.
- Ricezione sincrona. Il ricevente si blocca nell'attesa dell'arrivo di un messaggio.
- Ricezione asincrona. Il ricevente riceve un messaggio valido o un valore nullo.

È possibile anche avere diverse combinazioni di `send()` e `receive()`. Se le primitive `send()` e `receive()` sono entrambe bloccanti si parla di rendezvous tra mittente e ricevente. È banale dare soluzione al problema del produttore e del consumatore tramite le primitive bloccanti `send()` e `receive()`. Il produttore, infatti, si limita a invocare `send()` e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore richiama `receive()`, bloccandosi fino all'arrivo di un messaggio.

```
message next_produced;

while (true) {

    /* produce un elemento in next_produced */

    send(next_produced);
```

```
}
```

Figura 3.14 Processo produttore con l'utilizzo dello scambio di messaggi.

```
message next_consumed;  
  
while (true) {  
  
    receive(next_consumed);  
  
    /* consuma l'elemento in next_consumed */  
  
}
```

Figura 3.15 Processo consumatore con l'utilizzo dello scambio di messaggi.

Le Figure 3.14 e 3.15 mostrano queste operazioni.

3.6.3 Code di messaggi

Se la comunicazione è diretta o indiretta, i messaggi scambiati tra processi comunicanti risiedono in code temporanee. Fondamentalmente esistono tre modi per realizzare queste code.

- Capacità zero. La coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio.
- Capacità limitata. La coda ha lunghezza finita n , quindi al suo interno possono risiedere al massimo n messaggi. Se la coda non è piena, quando s'invia un nuovo messaggio, quest'ultimo è posto in fondo alla coda (il messaggio viene copiato oppure si tiene un puntatore a quel messaggio). Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale ha tuttavia una capacità limitata; se è pieno, il trasmittente deve fermarsi nell'attesa che ci sia spazio disponibile nella coda.
- Capacità illimitata. La coda ha una lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

Il caso con capacità zero è talvolta chiamato *sistema a scambio di messaggi senza buffering*; gli altri due, *sistemi con buffering automatico*.

3.7 Esempi di sistemi IPC

In questo paragrafo analizzeremo quattro sistemi per la comunicazione fra processi: la api posix basata sulla memoria condivisa, lo scambio di messaggi nel sistema operativo Mach, la comunicazione fra processi in Windows, che ha la caratteristica interessante di usare la memoria condivisa come meccanismo per supportare alcune forme di message passing e, infine, le pipe, uno dei primi meccanismi ipc sui sistemi unix.

3.7.1 Memoria condivisa in posix

Lo standard posix prevede svariati meccanismi per la ipc, compresa la memoria condivisa e lo scambio di messaggi. Qui illustreremo la api posix per la condivisione della memoria.

La memoria condivisa posix è organizzata utilizzando i file mappati in memoria, che associano la regione di memoria condivisa a un file. Un processo deve prima creare un oggetto memoria condivisa con la chiamata di sistema `shm_open()`, come segue:

```
fd = shm_open (name, O_CREAT | O_RDWR, 0666);
```

Il primo parametro specifica il nome dell'oggetto memoria condivisa. I processi che desiderano accedere a questa memoria condivisa devono fare riferimento all'oggetto mediante questo nome. I parametri successivi specificano che l'oggetto memoria condivisa deve essere creato se non esiste ancora (`O_CREAT`) e che l'oggetto è aperto in lettura e scrittura (`O_RDWR`). L'ultimo parametro definisce le autorizzazioni di directory dell'oggetto memoria condivisa. Una chiamata a `shm_open()` con esito positivo restituisce un intero, il descrittore di file per l'oggetto memoria condivisa.

Una volta che l'oggetto è creato, viene utilizzata la funzione `ftruncate()` per specificare la dimensione dell'oggetto in byte. La chiamata

```
ftruncate(fd, 4096);
```

imposta la dimensione dell'oggetto a 4.096 byte.

Infine, la funzione `mmap()` crea un file mappato in memoria che contiene l'oggetto memoria condivisa e restituisce un puntatore al file mappato in memoria che viene utilizzato per accedere all'oggetto memoria condivisa.

I programmi mostrati nelle Figure 3.16 e 3.17 usano la memoria condivisa per implementare il modello produttore-consumatore. Il produttore definisce un oggetto memoria condivisa e scrive sulla memoria condivisa, il consumatore legge dalla memoria condivisa.

Il produttore, mostrato nella Figura 3.16, crea un oggetto memoria condivisa denominato `os` e scrive la famigerata stringa "Hello World!" sulla memoria condivisa. Il programma mappa in memoria un oggetto memoria condivisa del formato specificato e consente di scrivere sull'oggetto. (Ovviamente, al produttore serve solo la scrittura). Il flag `MAP_SHARED` specifica che le modifiche apportate all'oggetto memoria condivisa saranno visibili a tutti i processi che condividono l'oggetto. Si noti che la scrittura avviene chiamando la funzione `sprintf()` e scrivendo la stringa formattata nella posizione puntata da `ptr`. Dopo ogni scrittura dobbiamo incrementare il puntatore secondo il numero di byte scritti.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
```

```

int main()
{
    /* dimensione, in byte, dell'oggetto memoria condivisa */

    const int SIZE 4096;

    /* nome dell'oggetto memoria condivisa */

    const char *name = "OS";

    /* stringa scritta nella memoria condivisa */

    const char *message_0 = "Hello";

    const char *message_1 = "World!";

    /* descrittore del file di memoria condivisa */

    int fd;

    /* puntatore all'oggetto memoria condivisa */

    char *ptr;

    /* crea l'oggetto memoria condivisa */

    fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configura la dimensione dell'oggetto memoria condivisa */

    ftruncate(fd, SIZE);

    /* mappa in memoria l'oggetto memoria condivisa */

    ptr = (char *)

    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    /* scrive sull'oggetto memoria condivisa */

    sprintf(ptr,"%s",message_0);

    ptr += strlen(message_0);

    sprintf(ptr,"%s",message_1);

    ptr += strlen(message_1);

    return 0;
}

```

Figura 3.16 Processo produttore che illustra l'api per la memoria condivisa posix.

Il processo consumatore, mostrato nella Figura 3.17, legge e visualizza il contenuto della memoria condivisa. Il consumatore invoca anche la funzione `shm_unlink()`, che rimuove il segmento di memoria condivisa dopo l'accesso del consumatore. Una trattazione più dettagliata della mappatura della memoria è presente nel Paragrafo 13.5.

```

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/mman.h>

int main()

{

/* dimensione, in byte, dell'oggetto memoria condivisa */

const int SIZE 4096;

/* nome dell'oggetto memoria condivisa */

const char *name = "OS";

/* descrittore del file di memoria condivisa */

int fd;

/* puntatore all'oggetto memoria condivisa */

void *ptr;

/* apre l'oggetto memoria condivisa */

fd = shm_open(name, O_RDONLY, 0666);

/* mappa in memoria l'oggetto memoria condivisa */

ptr = (char *)

mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

/* legge dall'oggetto memoria condivisa */

printf("%s", (char *)ptr);

/* rimuove l'oggetto memoria condivisa */

shm_unlink(name);

return 0;

}

```

Figura 3.17 Processo consumatore che illustra l'api per la memoria condivisa posix.

3.7.2 Mach

Come esempio di scambio di messaggi consideriamo ora il sistema operativo Mach. Mach è stato progettato appositamente per i sistemi distribuiti, ma si è dimostrato adatto anche per sistemi desktop e mobili, come dimostrato dalla sua inclusione nei sistemi operativi macos e ios, sottolineata nel Capitolo 2.

Il kernel Mach supporta la creazione e la distruzione di molteplici *task*, che sono simili ai processi, ma hanno multipli thread di controllo e meno risorse associate. La maggior parte delle comunicazioni in Mach, compresa tutta la comunicazione tra task, viene eseguita via messaggi. I messaggi vengono inviati e ricevuti da caselle di posta chiamate *porte*. Le porte hanno dimensioni finite e sono unidirezionali: per la comunicazione bidirezionale, un messaggio viene inviato a una porta e la risposta viene inviata a una porta di risposta distinta. Ogni porta può avere più trasmettenti, ma solo un ricevitore. Mach utilizza le porte per rappresentare risorse come i task, i thread, la memoria e i processori, mentre lo scambio di messaggi fornisce un approccio orientato agli oggetti per interagire con queste risorse e con i servizi di sistema. Lo scambio di messaggi può avvenire tra due porte sullo stesso host oppure su host diversi nel caso di un sistema distribuito.

A ciascuna porta è associata una raccolta di permessi (*port right*) che determinano le caratteristiche che un task deve avere per interagire con la porta. Per esempio, affinché un task possa ricevere un messaggio da una data porta, deve avere i permessi `MACH_PORT_RIGHT_RECEIVE` per quella porta. Il task che crea una porta ne diventa il proprietario e solo a tale task è consentito ricevere messaggi da tale porta. Il proprietario di una porta può gestirne i permessi, come avviene comunemente quando si stabilisce una porta di risposta. Per esempio, si supponga che il task *T1* sia proprietario della porta *P1* e invii un messaggio alla porta *P2*, di proprietà del task *T2*. Se *T1* si aspetta di ricevere una risposta da *T2*, deve concedere a *T2* i permessi `MACH_PORT_RIGHT_SEND` per la porta *P1*. I permessi sono assegnati a livello di task, quindi tutti i thread appartenenti allo stesso task condividono gli stessi permessi di utilizzo della porta. Due thread appartenenti allo stesso task possono quindi comunicare facilmente scambiandosi messaggi attraverso la porta a livello di thread associata a ciascuno di essi.

Quando si genera un task, vengono anche create due porte speciali: una *porta del task*, chiamata Task Self, e una *porta di notifica*, chiamata Notify. Il kernel possiede i permessi di ricezione per la porta Task Self, in modo da consentire a un task di inviare messaggi al kernel. Inoltre, il kernel può notificare l'occorrenza di un evento alla porta Notify di un task (su cui, ovviamente, il task ha ricevuto i permessi di ricezione).

La chiamata della funzione `mach_port_allocate()` crea una nuova porta, alloca lo spazio per la sua coda di messaggi e definisce i permessi per la porta. Ogni permesso rappresenta un *nome* distinto per la stessa porta ed è possibile accedere a una porta solo tramite il suo nome. I nomi delle porte sono semplici valori interi e si comportano in modo simile ai descrittori di file unix. L'esempio seguente illustra la creazione di una porta con l'utilizzo di `mach_port_allocate()`.

```
mach_port_t port; // nome della porta

mach_port_allocate (
    mach_task_self(), // task che fa riferimento a se stesso
    MACH_PORT_RIGHT_RECEIVE, // permesso per la porta
    &port); // nome della porta
```

Ogni task ha anche accesso a una porta bootstrap, che gli consente di registrare una porta creata con un server bootstrap a livello di sistema. Quando una porta è registrata con il server bootstrap, gli altri task la possono cercare nel registro e ottenerne i permessi per l'invio di messaggi alla porta.

La porta ha inizialmente una coda di messaggi vuota e i messaggi si copiano nella porta nell'ordine in cui sono ricevuti: tutti i messaggi hanno la stessa priorità. Mach garantisce che più messaggi in arrivo dallo stesso trasmettente siano accodati nell'ordine d'arrivo (*first-in, first-out, fifo*), ma non garantisce un ordinamento assoluto. Per esempio, i messaggi inviati da due trasmettenti possono essere accodati in un ordine qualsiasi.

I messaggi Mach sono formati dai seguenti due campi.

- Un'intestazione di messaggio di dimensione fissa contenente metadati relativi al messaggio, tra cui la sua dimensione e le porte di origine e di destinazione. Normalmente il thread trasmettente attende una risposta; il nome della porta del trasmettente è passato al task ricevente, che lo impiega come un "indirizzo di risposta".
- Un corpo di dimensioni variabili contenente dati.

I messaggi possono essere *semplici* o *complessi*. Un messaggio semplice contiene dati utente ordinari e non strutturati che non sono interpretati dal kernel. Un messaggio complesso può contenere puntatori a posizioni di memoria contenenti dati (noti come dati "out-of-line") o può essere utilizzato per trasferire i diritti di porta a un altro task. I puntatori a dati out-of-line sono particolarmente utili quando un messaggio deve trasferire grandi blocchi di dati. Un messaggio semplice richiederebbe, infatti, di copiare i dati e confezionare il messaggio, mentre la trasmissione di dati out-of-line richiede solo un puntatore che faccia riferimento alla posizione di memoria in cui sono memorizzati i dati.

La funzione `mach_msg()` è l'api standard per l'invio e la ricezione di messaggi: il valore di uno dei suoi parametri può essere impostato a `MACH_SEND_MSG` o `MACH_RECV_MSG` per indicare se si tratta di un'operazione di invio o ricezione, rispettivamente. Illustriamo ora come questa funzione viene utilizzata quando un task client invia un messaggio semplice a un task server. Supponiamo che ci siano due porte *client* e *server* associate rispettivamente ai task client e server. Il codice mostrato nella Figura 3.18 mostra il task client che costruisce un'intestazione e invia un messaggio al server, e il task server che riceve il messaggio inviato dal client.

```
#include<mach/mach.h>

struct message {
    mach_msg_header_t header;
    int data;
};

mach_port_t client;
mach_port_t server;

/* Codice del client */

struct message message;
// costruisce l'intestazione
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;
// invia il messaggio
mach_msg(&message.header, // intestazione del messaggio
MACH_SEND_MSG, // invia un messaggio
sizeof(message), // dimensione del messaggio spedito
0, // dimensione massima del messaggio ricevuto - non necessario
MACH_PORT_NULL, // nome della porta di ricezione - non necessario
MACH_MSG_TIMEOUT_NONE, // nessun timeout
MACH_PORT_NULL // nessuna porta di notifica
);

/* Codice del server */

struct message message;
// riceve un messaggio
mach_msg(&message.header, // intestazione del messaggio
MACH_RCV_MSG, // riceve un messaggio
0, // dimensione del messaggio spedito
sizeof(message), // dimensione massima del messaggio ricevuto
server, // nome della porta di ricezione
```

```

MACH_MSG_TIMEOUT_NONE, // nessun timeout

MACH_PORT_NULL // nessuna porta di notifica

);

```

Figura 3.18 Esempio di scambio di messaggi in Mach.

La funzione `mach_msg()` viene invocata dai programmi utente per eseguire uno scambio di messaggi. A sua volta, `mach_msg()` richiama la funzione `mach_msg_trap()`, una chiamata di sistema al kernel Mach. All'interno del kernel, `mach_msg_trap()` chiama la funzione `mach_msg_overwrite_trap()`, che gestisce il trasferimento effettivo del messaggio.

Le operazioni di invio e ricezione sono piuttosto flessibili. Per esempio, quando s'invia un messaggio a una porta la cui coda non è piena, il messaggio viene copiato nella coda e il task trasmittente prosegue la sua esecuzione. Se la coda della porta è piena, il trasmittente ha diverse opzioni (specificabili tramite parametri della funzione `mach_msg()`):

1. Attendere indefinitamente che nella porta ci sia spazio.
2. Attendere al massimo n millisecondi.
3. Non attendere, ma ritornare immediatamente.
4. Memorizzare temporaneamente il messaggio. Un messaggio viene consegnato al sistema operativo anche se la porta che dovrebbe riceverlo è piena. Quando il messaggio può effettivamente essere messo nella porta, s'invia un avviso al trasmittente; per una porta piena può restare in sospeso un solo messaggio di questo tipo, in qualunque momento, per un dato thread trasmittente.

L'ultima possibilità si usa per i task che svolgono servizi (*server task*). Dopo aver portato a termine una richiesta, questi task possono aver bisogno d'inviare un'unica risposta al task che aveva richiesto il servizio, ma devono anche proseguire con altre richieste di servizi, anche se la porta di risposta per un client è piena.

Il problema principale dei sistemi a scambio di messaggi è generalmente costituito dalle scarse prestazioni della copia dei messaggi dalla porta del trasmittente alla porta del ricevente. Il sistema di messaggi di Mach tenta di evitare le operazioni di copiatura usando tecniche di gestione della memoria virtuale (Capitolo 10). Fondamentalmente Mach mappa lo spazio d'indirizzi contenente il messaggio del trasmittente nello spazio d'indirizzi del ricevente. Il messaggio non è mai effettivamente copiato, quindi le prestazioni del sistema migliorano notevolmente, anche se solo nel caso di messaggi scambiati all'interno di uno stesso sistema.

3.7.3 Windows

Il sistema operativo Windows è un esempio di moderno progetto che impiega la modularità per aumentare la funzionalità e diminuire il tempo necessario alla realizzazione di nuove caratteristiche. Windows gestisce più ambienti operativi o *sottosistemi*, con cui i programmi applicativi comunicano attraverso un meccanismo a scambio di messaggi; tali programmi si possono dunque considerare *client* del *server* costituito dal sottosistema.

La funzione di scambio di messaggi di Windows è detta chiamata di procedura locale avanzata (*advanced local procedure call*, alpc) e si usa per la comunicazione tra due processi presenti nello stesso calcolatore. È simile al meccanismo standard della chiamata di procedura remota, ma è ottimizzata per questo sistema (le chiamate di procedura remota sono trattate in dettaglio nel Paragrafo 3.8.2). Come in Mach, nel sistema Windows s'impiega un oggetto porta per stabilire e mantenere una connessione tra due processi. Windows usa due tipi di porte: le porte di connessione e le porte di comunicazione.

I processi server pubblicano gli oggetti porte di connessione che sono visibili a tutti i processi. Quando un client vuole i servizi di un sottosistema, apre un handle all'oggetto porta di connessione del server e invia una richiesta di connessione a quella porta. Il server crea quindi un canale e restituisce un handle al client. Il canale è costituito da una coppia di porte di comunicazione private: una per i messaggi client-server, l'altra per i messaggi server-client. Inoltre, i canali di comunicazione supportano un meccanismo di callback che permette a client e server di accettare le richieste anche quando sarebbero normalmente in attesa di una risposta.

Quando viene creato un canale alpc viene scelta una tra le seguenti tre tecniche di scambio di messaggi.

1. Per piccoli messaggi (fino a 256 byte) la coda di messaggi della porta viene utilizzata come deposito intermedio e i messaggi vengono copiati da un processo all'altro.
2. Messaggi più grandi devono essere fatti passare attraverso un oggetto sezione, che è una regione di memoria condivisa associata al canale.
3. Quando la quantità di dati è troppo grande per essere contenuta in un oggetto sezione è disponibile una api che consente ai processi server di leggere e scrivere direttamente nello spazio degli indirizzi di un client.

Il client deve decidere, quando attiva il canale, se dovrà inviare messaggi lunghi: in tal caso richiede la creazione di un oggetto sezione. Allo stesso modo, se il server stabilisce che le risposte sono lunghe, provvede alla creazione di un oggetto sezione. Per usare gli oggetti sezione s'invia un breve messaggio contenente un puntatore e le informazioni sulle sue dimensioni. Questo metodo è un po' più complicato del primo metodo sopra descritto, ma evita la copiatura dei dati. La struttura delle chiamate di procedura locale avanzate in Windows è illustrata nella Figura 3.19.

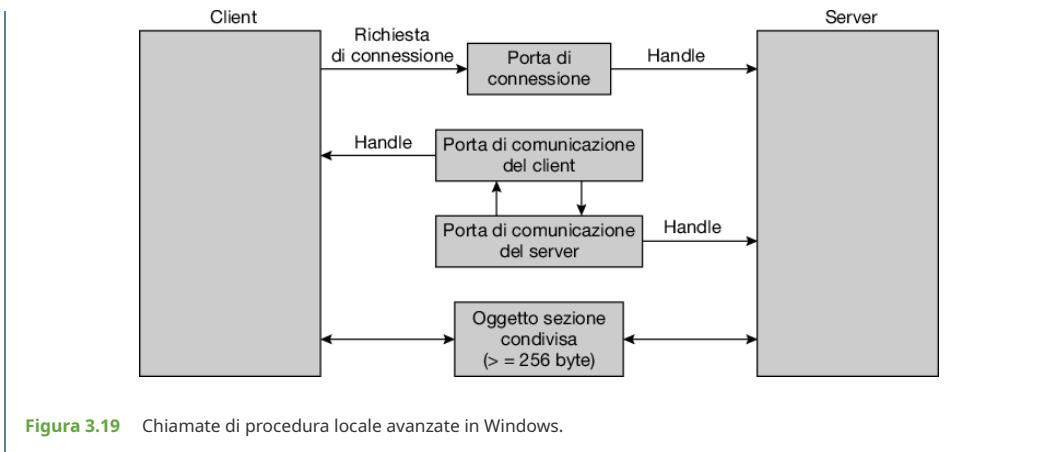


Figura 3.19 Chiamate di procedura locale avanzate in Windows.

È importante notare che il meccanismo alpc di Windows non è parte della api Windows e quindi non è accessibile ai programmati di applicazioni. Le applicazioni che utilizzano l'api Windows devono comunque usare rpc standard; nel caso in cui la chiamata si riferisca a un processo residente sulla stessa macchina del chiamante, il sistema la implementa tramite una alpc. Inoltre molti servizi del kernel utilizzano alpc per comunicare con processi client.

3.7.4 Pipe

Una pipe agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi (ipc) nei primi sistemi unix e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro, sebbene con qualche limitazione. Quando si implementa una pipe devono essere prese in considerazione quattro questioni.

1. La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?
2. Se è ammessa la comunicazione bidirezionale, essa è di tipo *half duplex* (i dati possono viaggiare in un'unica direzione alla volta) o *full duplex* (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
3. Deve esistere una relazione (del tipo *padre-figlio*) tra i processi in comunicazione?
4. Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

Nei paragrafi seguenti esploriamo due tipi comuni di pipe utilizzate sia in unix sia in Windows: le pipe convenzionali e le named pipe.

3.7.4.1 Pipe convenzionali

Le pipe convenzionali permettono a due processi di comunicare secondo una modalità standard chiamata del produttore-consumatore. Il produttore scrive a una estremità del canale (l'estremità dedicata alla scrittura, o *write-end*) mentre il consumatore legge dall'altra estremità (l'estremità dedicata alla lettura, o *read-end*). Le pipe convenzionali sono quindi unidirezionali, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la comunicazione bidirezionale devono essere utilizzate due *pipe*, ognuna delle quali manda i dati in una direzione. Illustreremo la costruzione di pipe convenzionali sia in unix sia in Windows. In entrambi i programmi di esempio un processo scrive sulla pipe il messaggio `Greetings`, mentre l'altro lo legge dall'altra estremità della pipe.

Nei sistemi unix le pipe convenzionali sono costruite utilizzando la funzione

```
pipe(int fd[])
```

Essa crea una pipe alla quale si può accedere tramite i descrittori del file `int fd[1]:fd[0]` è l'estremità dedicata alla lettura, mentre `fd[1]` è l'estremità dedicata alla scrittura. Il sistema unix considera una pipe come un tipo speciale di file; si può così accedere alle pipe tramite le usuali chiamate di sistema `read()` e `write()`.

Non si può accedere a una pipe al di fuori del processo che la crea. Solitamente un processo padre crea una pipe e la utilizza per comunicare con un processo figlio generato con il comando `fork()`. Come già indicato nel Paragrafo 3.3.1, il processo figlio eredita i file aperti dal processo padre. Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre. La Figura 3.20 illustra la relazione del descrittore di file `fd` rispetto ai processi padre e figlio. Come mostrato, ogni scrittura del padre sull'estremità dedicata alla scrittura della pipe, `fd[1]`, può essere letta dal figlio sull'estremità dedicata alla lettura, `fd[0]`.

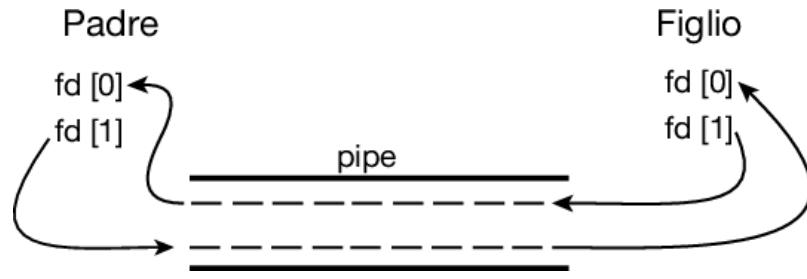


Figura 3.20 Descrittori di file per una pipe convenzionale.

Nel programma unix mostrato nella Figura 3.21 (che prosegue nella Figura 3.22) il processo padre crea una pipe e in seguito esegue una chiamata `fork()`, generando un processo figlio. Ciò che succede dopo la chiamata `fork()` dipende da come i dati devono fluire nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale. Sebbene il programma mostrato nella Figura 3.21 non richieda questa azione è importante assicurare che un processo che legge dalla pipe possa rilevare l'end-of-file (`read()` restituisce 0) quando chi scrive ha chiuso la sua estremità della pipe.

```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;
```

Figura 3.21 Pipe convenzionali in unix.

```
/* crea la pipe */
```

```

if (pipe(fd) == -1) {

    fprintf(stderr,"Pipe failed");

    return 1;
}

/* crea tramite fork un processo figlio */

pid = fork();

if (pid < 0) ( /* errore */

    fprintf(stderr, "Fork Failed");

    return 1;
}

if (pid > 0) ( /* processo padre */

    /* chiude l'estremità inutilizzata della pipe */

    close(fd[READ_END]);

    /* scrive sulla pipe */

    write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

    /* chiude l'estremità della pipe dedicata alla scrittura */

    close(fd[WRITE_END]);
}

else ( /* processo figlio */

    /* chiude l'estremità inutilizzata della pipe */

    close(fd[WRITE_END]);

    /* legge dalla pipe */

    read(fd[READ_END], read_msg, BUFFER_SIZE);

    printf("read %s",read_msg);

    /* chiude l'estremità della pipe dedicata alla lettura */

    close(fd[READ_END]);
}

return 0;
}

```

Figura 3.22 Continuazione della Figura 3.21.

Nei sistemi Windows le pipe convenzionali sono denominate pipe anonime e si comportano analogamente alle loro equivalenti in unix: sono unidirezionali e utilizzano relazioni del tipo padre-figlio tra i processi coinvolti nella comunicazione. Inoltre, l'attività di lettura e scrittura sulla pipe può essere eseguita con le usuali funzioni `ReadFile()` e `WriteFile()`. L'api Windows per creare le pipe è

la funzione `CreatePipe()`, che riceve in ingresso quattro parametri. I parametri forniscono handle distinti per (1) leggere e (2) scrivere sulla pipe, oltre a (3) una istanza della struttura `STARTUPINFO`, usata per specificare che il processo figlio erediti gli handle della pipe. Inoltre, può essere specificata (4) la dimensione della pipe (in byte).

La Figura 3.23 (che prosegue nella Figura 3.24) illustra un processo padre che crea una pipe anonima per comunicare con il proprio figlio. A differenza dei sistemi unix, dove un processo figlio eredita automaticamente una pipe creata dal proprio padre, Windows richiede al programmatore di specificare quali attributi saranno ereditati dal processo figlio. Ciò avviene innanzitutto inizializzando la struttura `SECURITY_ATTRIBUTES` per permettere che gli handle siano ereditati e poi reindirizzando lo standard input o lo standard output del processo figlio verso l'handle di scrittura o di lettura della pipe, rispettivamente. Dato che il figlio leggerà dalla pipe, il padre deve reindirizzare lo standard input del figlio verso l'handle di lettura della pipe stessa. Inoltre, dal momento che le pipe sono half duplex, è necessario proibire al figlio di ereditare l'handle di scrittura della pipe. La creazione del processo figlio avviene come nel programma nella Figura 3.10, fatta eccezione per il quinto parametro che in questo caso viene impostato al valore true per indicare che il processo figlio eredita degli handle specifici dal proprio padre. Prima di scrivere sulla pipe, il padre ne chiude l'estremità di lettura, che è inutilizzata. Il processo figlio che legge dalla pipe è mostrato nella Figura 3.25. Prima di leggere dalla pipe, questo programma ottiene l'handle di lettura invocando `GetStdHandle()`.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#define BUFFER_SIZE 25
int main(VOID)
{
    HANDLE ReadHandle, WriteHandle;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    char message[BUFFER_SIZE] = "Greetings";
    DWORD written;
```

Figura 3.23 Pipe anonne in Windows (processo genitore).

```
/* imposta gli attributi di sicurezza in modo che le pipe siano ereditate */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};

/* alloca la memoria */
ZeroMemory(&pi, sizeof(pi));

/* crea la pipe */

if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0))
{
    fprintf(stderr, "Create Pipe Failed");

    return 1;
```

```

}

/* prepara la struttura START_INFO per il processo figlio */

GetStartupInfo(&si);

si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* reindirizza lo standard input verso l'estremità della pipe dedicata alla lettura
 */

si.hStdInput = ReadHandle;

si.dwFlags = STARTF_USESTDHANDLES;

/* non permette al processo figlio di ereditare l'estremità della pipe dedicata alla
scrittura */

SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* crea il processo figlio */

CreateProcess(NULL, "child.exe", NULL, NULL,
TRUE, /* inherit handles */

0, NULL, NULL, &si, &pi);

/* chiude l'estremità inutilizzata della pipe */

CloseHandle(ReadHandle);

/* il padre scrive sulla pipe */

if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))

fprintf(stderr, "Error writing to pipe.");

/* chiude l'estremità della pipe dedicata alla scrittura */

CloseHandle(WriteHandle);

/* attende la terminazione del processo figlio */

WaitForSingleObject(pi.hProcess, INFINITE);

CloseHandle(pi.hProcess);

CloseHandle(pi.hThread);

return 0;

}

```

Figura 3.24 Continuazione della Figura 3.23.

```
#include <stdio.h>
```

```

#include <windows.h>

#define BUFFER_SIZE 25

int main(VOID)
{
    HANDLE ReadHandle;
    CHAR buffer[BUFFER_SIZE];
    DWORD read;

    /* riceve l'handle di lettura della pipe */

    ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

    /* il figlio legge dalla pipe */

    if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))

        printf("child read %s",buffer);

    else

        fprintf(stderr, "Error reading from pipe");

    return 0;
}

```

Figura 3.25 Pipe anonime in Windows (processo figlio).

Si noti bene che le pipe convenzionali richiedono una relazione di parentela padre-figlio tra i processi comunicanti, sia in unix sia in Windows. Ciò significa che queste pipe possono essere utilizzate soltanto per la comunicazione tra processi in esecuzione sulla stessa macchina.

3.7.4.2 *Named pipe*

Le pipe convenzionali offrono un meccanismo semplice di comunicazione tra una coppia di processi. Tuttavia, le pipe convenzionali esistono solo mentre i processi stanno comunicando fra loro. Sia in unix sia in Windows, una volta che i processi hanno finito di comunicare e terminano, le pipe convenzionali cessano di esistere.

Le named pipe costituiscono uno strumento di comunicazione molto più potente; la comunicazione può essere bidirezionale, e la relazione di parentela padre-figlio non è necessaria. Una volta che si sia creata la named pipe, diversi processi possono utilizzarla per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori. In più, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. Sia unix sia Windows mettono a disposizione le named pipe, nonostante ci siano grandi differenze nei dettagli dell'implementazione. Di seguito, prendiamo in esame le named pipe in ciascuno dei due sistemi.

Nei sistemi unix le named pipe sono dette fifo. Una volta create, esse appaiono come normali file all'interno del file system. Una fifo viene creata mediante una chiamata di sistema `mkfifo()` e viene poi manipolata con le usuali chiamate di sistema `open()`, `read()`, `write()` e `close()`; essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system. Nonostante le fifo permettano la comunicazione bidirezionale, l'unica tipologia di trasmissione consentita è quella half duplex. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due fifo. Per utilizzare le fifo i processi comunicanti devono risiedere sulla stessa macchina: se è richiesta la comunicazione tra più macchine devono essere impiegate le socket (si veda il Paragrafo 3.8.1).

Rispetto alle loro controparti in unix, le named pipe su un sistema Windows offrono un meccanismo di comunicazione più ricco. È permessa la comunicazione full duplex e i processi comunicanti possono risiedere sia sulla stessa macchina sia su macchine diverse. Inoltre, attraverso una fifo di unix possono essere trasmessi solo dati byte-oriented, mentre i sistemi Windows permettono la trasmissione di dati sia byte-oriented sia message-oriented. Le named pipe vengono create con la funzione `CreateNamedPipe()` e un client può connettersi a una named pipe tramite `ConnectNamedPipe()`. La comunicazione attraverso le named pipe avviene grazie alle funzioni `ReadFile()` e `WriteFile()`.

Le pipe sono usate molto spesso dalla riga di comando di unix in situazioni nelle quali l'output di un comando serve da input per un altro comando. Per esempio, il comando `ls` di unix elenca il contenuto di una directory. Per directory particolarmente grandi, l'output può scorrere su diverse schermate. Il comando `less` gestisce l'output mostrando solo una schermata alla volta; l'utente deve premere la barra spaziatrice per muoversi da una schermata all'altra. Creando una pipe tra i comandi `ls` e `less` (in esecuzione come singoli processi) si fa in modo che l'output di `ls` venga inviato all'input di `less`, permettendo così all'utente di vedere il contenuto di una grande directory una schermata alla volta.

Dalla riga di comando si può costruire una pipe utilizzando il carattere `|`. Il comando completo è quindi

```
ls | less
```

In questo scenario, il comando `ls` funge da produttore, e il suo output è consumato dal comando `less`.

I sistemi Windows offrono un comando `more` per la shell dos con una funzionalità analoga al corrispettivo di unix `less`. Anche la shell dos utilizza il carattere `|` per creare una pipe. L'unica differenza è costituita dal fatto che, per restituire il contenuto di una directory, dos utilizza il comando `dir` anziché `ls`. Il comando equivalente in dos è quindi

```
dir | more
```

3.8 Comunicazione nei sistemi client-server

Nel Paragrafo 3.4 ci siamo soffermati su come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione in sistemi client/server (Paragrafo 1.10.3). Consideriamo qui altre due strategie: socket e chiamate di procedura remota (rpc). Come vedremo, le rpc non sono soltanto utili per la computazione client-server, ma sono anche utilizzate, nel caso di Android, come forma di comunicazione tra processi in esecuzione sullo stesso sistema.

3.8.1 Socket

Una *socket* è definita come l'estremità di un canale di comunicazione. Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo ip concatenato a un numero di porta. In generale, le socket impiegano un'architettura client-server; il server attende le richieste dei client, stando in ascolto a una porta specificata; quando il server riceve una richiesta, accetta la connessione proveniente dalla socket del client, e si stabilisce la comunicazione. I server che svolgono servizi specifici (come ssh, ftp e http) stanno in ascolto su porte ben note (i server ssh alla porta 23, i server ftp alla porta 21, e i server Web, o http, alla porta 80). Tutte le porte al di sotto del *valore* 1024 sono considerate *ben note* (*well known*) e si usano per realizzare servizi standard.

Quando un processo client richiede una connessione, il calcolatore che lo ospita assegna una porta specifica, che consiste di un numero arbitrario maggiore di 1024. Si supponga per esempio che un processo client presente nel calcolatore x con indirizzo ip 146.86.5.20 voglia stabilire una connessione con un server web (in ascolto alla porta 80) all'indirizzo 161.25.19.8; il calcolatore x potrebbe assegnare al client, per esempio, la porta 1625. La connessione sarebbe composta di una coppia di socket: (146.86.5.20:1625) nel calcolatore x e (161.25.19.8:80) nel server web. La Figura 3.26 mostra questa situazione. La consegna dei pacchetti al processo giusto avviene secondo il numero della porta di destinazione.

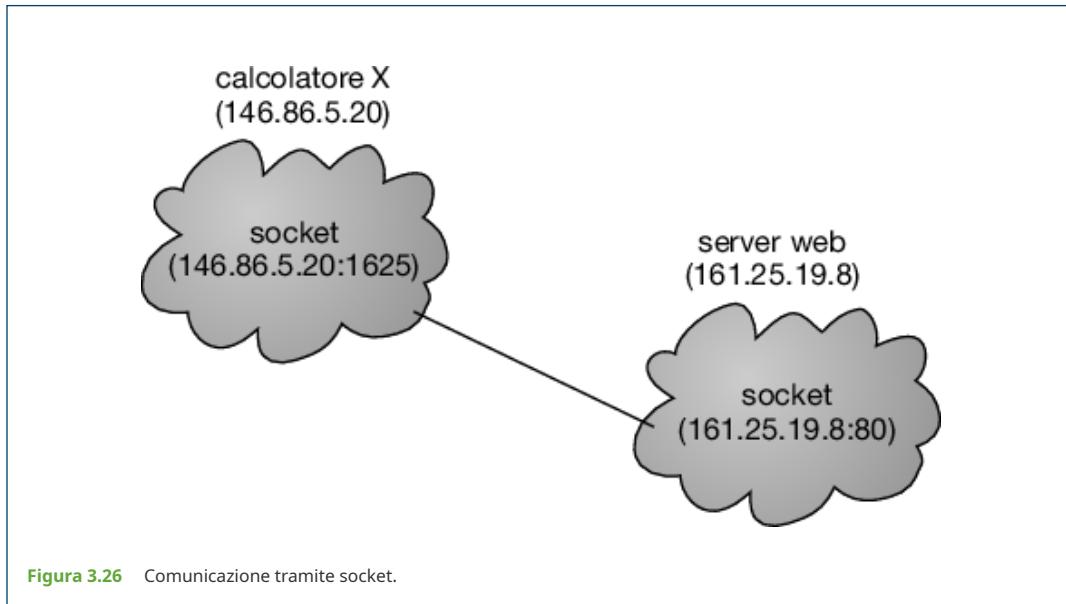


Figura 3.26 Comunicazione tramite socket.

Tutte le connessioni devono essere uniche; quindi, se un altro processo, nel calcolatore x, voglia stabilire un'altra connessione con lo stesso server web, riceve un numero di porta maggiore di 1024 e diverso da 1625. Ciò assicura che ciascuna connessione sia identificata da una distinta coppia di socket.

Sebbene la maggior parte degli esempi di programmazione di questo testo sia scritta in C, le socket sono illustrate usando il linguaggio Java, poiché offre un'interfaccia alle socket più semplice e dispone di una ricca libreria di utilità di networking.

Il linguaggio Java prevede tre tipi differenti di socket: quelle orientate alla connessione (tcp) sono realizzate con la classe `Socket`; quelle senza connessione (udp) usano la classe `DatagramSocket`; il terzo tipo di socket è basato sulla classe `MulticastSocket`; si tratta di una sottoclassificazione della classe `DatagramSocket` che permette l'invio simultaneo dei dati a diversi destinatari (*multicast*).

Nel nostro esempio un server usa socket tcp orientate alla connessione per fornire l'ora e la data correnti ai client. Il server si pone in ascolto alla porta 6013 – potrebbe essere un altro numero arbitrario, purché maggiore di 1024. Quando giunge una richiesta di connessione, il server restituisce la data e l'ora al client.

Il codice del server è mostrato nella Figura 3.27. Il server crea una `ServerSocket` che ascolta alla porta 6013. Il server si pone in ascolto tramite la chiamata bloccante `accept()` e rimane in attesa fino all'arrivo della richiesta di un client. A quel punto, `accept()` restituisce la socket che il server può usare per comunicare con il client.

```
import java.net.*;
import java.io.*;

public class DateServer

{

    public static void main(String[] args) {

        try {

            ServerSocket sock = new ServerSocket(6013);

            /* si pone in ascolto di richieste di connessione */

            while (true) {

                Socket client = sock.accept();

                PrintWriter pout = new

                PrintWriter(client.getOutputStream(), true);

                /* scrive la Data sulla socket */

                pout.println(new java.util.Date().toString());

                /* chiude la socket */

                /* ritorna in ascolto di nuove richieste */

                client.close();

            }

        }

        catch (IOException ioe) {

            System.err.println(ioe);

        }

    }

}
```

Figura 3.27 Server che fornisce al client la data corrente.

Ecco alcuni dettagli relativi all'uso da parte del server della socket connessa al client. Il server crea da principio un oggetto di classe `PrintWriter` che gli permette di scrivere sulla socket tramite i metodi `print()` e `println()`. Esso manda quindi la data e l'ora al

client scrivendo sulla socket tramite `println()`; a questo punto, chiude la socket di comunicazione con il client e torna in attesa di nuove richieste.

Un client comunica con il server creando una socket e collegandosi per suo tramite alla porta su cui il server è in ascolto. L'implementazione è mostrata nella Figura 3.28. Il client crea una `Socket` e richiede una connessione al server alla porta 6013 dell'indirizzo ip 127.0.0.1. Stabilita la connessione, il client può leggere dalla socket tramite le ordinarie istruzioni di i/o. Dopo aver ricevuto la data dal server, il client chiude la socket e termina. L'indirizzo ip 127.0.0.1, noto come loopback, è peculiare: è usato da una macchina per riferirsi a se stessa. Tramite questo stratagemma, un client e un server residenti sulla stessa macchina sono in grado di comunicare tramite il protocollo tcp/ip. L'indirizzo ip 127.0.0.1 si potrebbe sostituire con l'indirizzo ip di una qualunque macchina che ospiti il server che fornisce la data. È anche possibile usare un nome simbolico, come `www.westminstercollege.edu`.

```
import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            /* si collega alla porta su cui ascolta il server */

            Socket sock = new Socket("127.0.0.1", 6013);

            InputStream in = sock.getInputStream();

            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            /* legge la data dalla socket */

            String line;

            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            /* chiude la socket */

            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Figura 3.28 Client che riceve dal server la data corrente.

La comunicazione tramite socket è diffusa ed efficiente, ma è considerata una forma di comunicazione di basso livello fra sistemi distribuiti. Infatti, le socket permettono unicamente la trasmissione di un flusso non strutturato di byte: è responsabilità del client e del

server interpretare e organizzare i dati in strutture più complesse. Nel prossimo paragrafo viene illustrato un metodo di comunicazione di livello più alto, le chiamate di procedure remote (rpc)

3.8.2 Chiamate di procedure remote

Uno tra i più diffusi tipi di servizio remoto è il paradigma della rpc, progettata per astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete. Per molti aspetti è simile al meccanismo ipc descritto nel Paragrafo 3.4, ed è generalmente costruita su un sistema di questo tipo. Tuttavia in questo caso, poiché in un sistema distribuito si eseguono i processi su sistemi distinti, per offrire un servizio remoto occorre impiegare uno schema di comunicazione basato sullo scambio di messaggi.

Contrariamente ai messaggi ipc, i messaggi scambiati per la comunicazione rpc sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone rpc, in ascolto a una porta del sistema remoto, e contengono un identificatore della funzione da eseguire e i parametri da passare a tale funzione. Nel sistema remoto si esegue questa funzione e s'invia ogni risultato al richiedente in un messaggio distinto.

La porta è semplicemente un numero inserito all'inizio del messaggio. Mentre un singolo sistema ha normalmente un solo indirizzo di rete, all'interno dell'indirizzo può avere molte porte che servono a distinguere i numerosi servizi che può fornire in rete. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente; per esempio, per permettere ad altri di ottenere un elenco dei suoi attuali utenti, un sistema deve possedere un demone in ascolto a una porta, per esempio la porta 3027, che realizzzi una siffatta rpc. Qualsiasi sistema remoto può ottenere l'informazione richiesta, vale a dire l'elenco degli utenti, inviando un messaggio rpc alla porta 3027 del server; i dati si ricevono in un messaggio di risposta.

La semantica delle rpc permette a un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Il sistema delle rpc nasconde i dettagli necessari che consentono la comunicazione, fornendo uno stub al lato client. Esiste in genere uno stub per ogni diversa procedura remota. Quando il client la invoca, il sistema delle rpc richiama l'appropriato stub, passando i parametri della procedura remota. Lo stub individua la porta del server e struttura i parametri; la strutturazione dei parametri (*marshaling*) implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Lo stub quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo stub nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica. Sui sistemi Windows, il codice dello stub è compilato a partire da una specifica, scritta nel Microsoft Interface Definition Language (midl), utilizzato per definire le interfacce tra i programmi client e server.

Il marshaling dei parametri risolve il problema delle differenze nella rappresentazione dei dati sulle macchine client e server. Si consideri la rappresentazione a 32 bit dei numeri interi; alcuni sistemi, noti come *big-endian*, usano l'indirizzo di memoria minore per contenere il byte più significativo; altri, noti come *little-endian*, lo usano per contenere il byte meno significativo. Nessuna delle due opzioni è migliore di per sé, si tratta di una scelta arbitraria nella definizione dell'architettura del computer. Per risolvere tali differenze molti sistemi di rpc definiscono una rappresentazione dei dati indipendente dalla macchina. Uno di questi sistemi di rappresentazione è noto come rappresentazione esterna dei dati (*external data representation*, *xdr*). Nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato *xdr*; nel server, i dati nel formato *xdr* si convertono nel formato della macchina server.

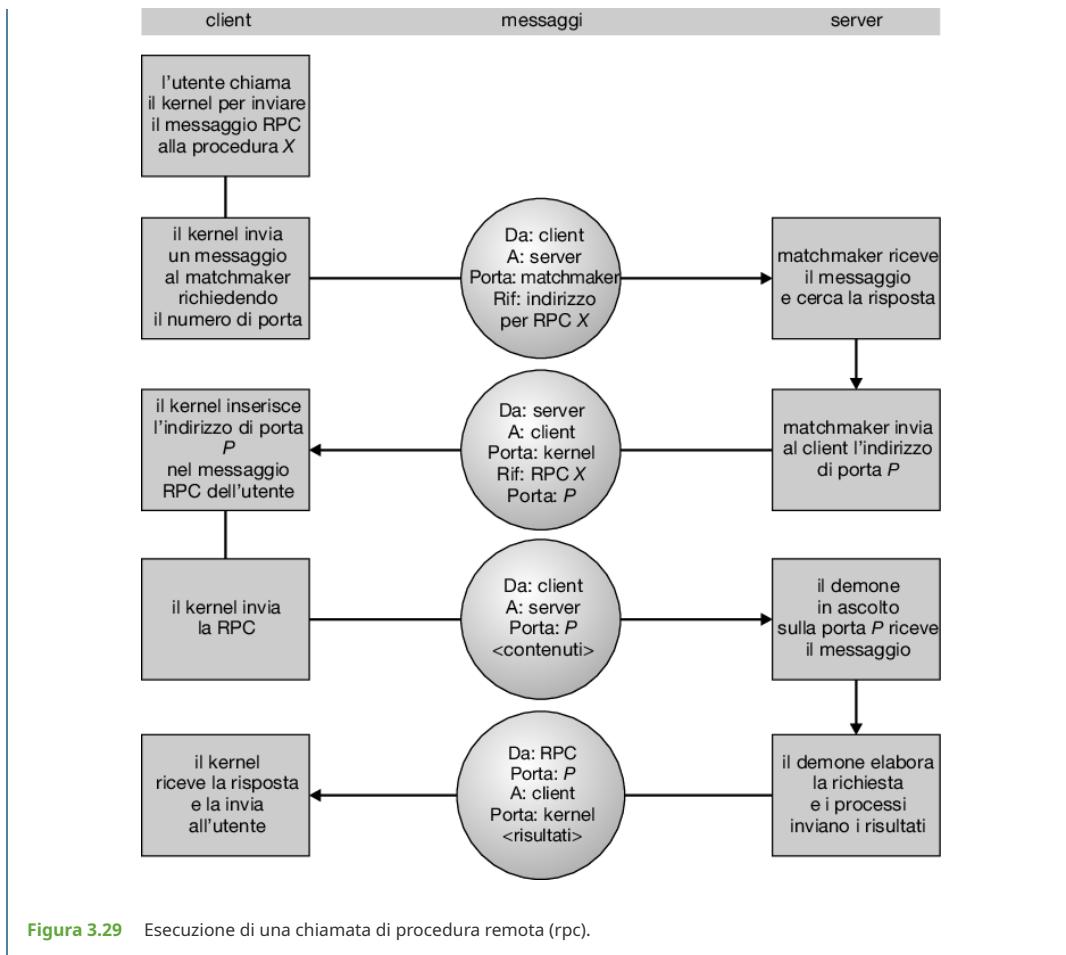
Un'altra questione importante riguarda la semantica delle chiamate. Infatti, mentre le chiamate locali falliscono in circostanze estreme, le rpc possono non riuscire, o risultare duplicate e dunque eseguite più volte, semplicemente a causa di comuni errori della rete. Un modo per affrontare il problema è che il sistema operativo garantisca che i messaggi vengono elaborati *esattamente una volta*, e non *al massimo una volta*. Questo tipo di semantica è comune per le chiamate locali, ma è più difficile da implementare per le rpc.

Si consideri prima la semantica "al massimo una volta". Essa può essere implementata marcando ogni messaggio con la sua ora di emissione (*timestamp*). Il server dovrà mantenere l'archivio di tutti gli orari di emissione dei messaggi già ricevuti e trattati, o perlomeno un archivio che sia abbastanza ampio da identificare tutti i messaggi duplicati. I messaggi in entrata con orario di emissione già presente nell'archivio sono ignorati. Il client avrà allora la sicurezza che la procedura remota sarà eseguita al massimo una volta, anche nel caso di un invio di più copie di uno stesso messaggio.

Per implementare la semantica "esattamente una volta" (*exactly once*), occorre eliminare il rischio che il server non riceva mai la richiesta. Per ottenere questo risultato, il server deve implementare la semantica "al massimo una volta" (*at most once*), ma integrarla con l'invio al client di un riscontro che attesti l'avvenuta esecuzione della procedura. Riscontri di questo tipo sono molto diffusi nella comunicazione tramite reti. Il client dovrà inviare periodicamente la richiesta rpc finché non ottenga il relativo riscontro.

Un altro argomento importante riguarda la comunicazione tra server e client. Con le ordinarie chiamate di procedure, durante la fase di collegamento, caricamento o esecuzione di un programma (Capitolo 9), ha luogo una forma di associazione che sostituisce il nome della procedura chiamata con l'indirizzo di memoria della procedura stessa. Lo schema delle rpc richiede una corrispondenza di questo genere tra il client e la porta del server, ma come fa il client a conoscere i numeri di porta sul server? Nessun sistema dispone d'informazioni complete sull'altro, poiché essi non condividono memoria.

Per risolvere questo problema s'impiegano per lo più due metodi. Con il primo, l'informazione sulla corrispondenza tra il client e la porta del server si può predeterminare fissando gli indirizzi delle porte: una rpc si associa nella fase di compilazione a un numero di porta fisso; il server non può modificare né il numero di porta né il servizio richiesto. Con il secondo metodo la corrispondenza si può effettuare dinamicamente tramite un meccanismo di *rendezvous*. Generalmente il sistema operativo fornisce un demone di rendezvous (*matchmaker*) a una porta di rpc fissata. Un client invia un messaggio, contenente il nome della rpc, al demone di rendezvous per richiedere l'indirizzo della porta della rpc che deve eseguire. Il demone risponde col numero di porta, e la richiesta d'esecuzione della rpc si può inviare a quella porta fino al termine del processo (o fino alla caduta del server). Questo metodo richiede un ulteriore carico a causa della richiesta iniziale, ma è più flessibile del primo metodo. La Figura 3.29 illustra un esempio d'interazione.



Lo schema della rpc è utile nella realizzazione di un file system distribuito (Capitolo 19); un sistema di questo tipo si può realizzare come un insieme di demoni e client di rpc. I messaggi s'indirizzano alla porta del file system distribuito su un server in cui deve avvenire l'operazione sui file. I messaggi contengono le operazioni da svolgere sui dischi: `read()`, `write()`, `rename()`, `delete()` o `status()`, corrispondenti alle normali chiamate di sistema che si usano per i file. Il messaggio di risposta contiene i dati risultanti da quella chiamata, che il demone del file system distribuito esegue su incarico del client. Un messaggio può, per esempio, contenere una richiesta di trasferimento di un intero file a un client, oppure semplici richieste di blocchi. Nel secondo caso per trasferire un intero file possono essere necessarie parecchie richieste di questo tipo.

3.8.2.1 Chiamate di procedure remote in Android

Sembra che le rpc siano di solito associate all'elaborazione client-server, in un sistema distribuito possono anche essere usate come una forma di comunicazione tra processi in esecuzione sullo stesso sistema. Il sistema operativo Android ha un ricco insieme di meccanismi ipc contenuti nel suo ambiente binder, tra cui le rpc che consentono a un processo di richiedere servizi a un altro processo.

Android definisce il componente applicativo (*application component*) come il blocco elementare di base in grado di fornire funzionalità a un'applicazione Android; ogni app può combinare più componenti applicativi. Uno di questi componenti è il servizio (*service*), che non ha un'interfaccia utente, ma resta in esecuzione in background mentre esegue lunghe operazioni o lavora per processi remoti. Per esempio, sono servizi la riproduzione di musica in background e il recupero di dati tramite una connessione di rete per conto di un altro processo, per prevenirne il blocco in attesa che i dati vengono scaricati. Quando un'app client richiama il metodo `bindService()` di un servizio, tale servizio diventa vincolato (bound) e disponibile per fornire comunicazioni client-server utilizzando lo scambio di messaggi o le rpc.

Un servizio bound deve estendere la classe `Service` e deve implementare il metodo `onBind()`, che viene invocato quando un client chiama `bindService()`. Nel caso dello scambio di messaggi, il metodo `onBind()` restituisce un servizio `Messenger`, utilizzato per inviare messaggi dal client al servizio. Il servizio `Messenger` è solo a senso unico; se il servizio deve inviare una risposta al client, il client deve fornire anche un servizio `Messenger`, contenuto nel campo `replyTo` dell'oggetto `Message` inviato al servizio. Il servizio può quindi inviare messaggi di risposta al client.

Per fornire le rpc, il metodo `onBind()` deve restituire un'interfaccia che rappresenta i metodi nell'oggetto remoto che i client utilizzano per interagire con il servizio. Questa interfaccia è scritta con la normale sintassi Java e utilizza l'Android Interface Definition Language (aidl) per creare stub che fungano da interfaccia client ai servizi remoti.

Descriviamo brevemente il procedimento da seguire per fornire un servizio remoto generico denominato `remoteMethod()` utilizzando aidl e il binder. L'interfaccia per il servizio remoto appare come segue:

```
/* RemoteService.aidl */

interface RemoteService
{
    boolean remoteMethod (int x, double y);
}
```

Questo file, `RemoteService.aidl`, verrà utilizzato dal kit di sviluppo Android per generare un'interfaccia `.java`, nonché uno stub che funge da interfaccia rpc per questo servizio. Il server deve implementare l'interfaccia generata dal file `.aidl` e l'implementazione di questa interfaccia verrà chiamata quando il client invoca `remoteMethod()`.

Quando un client chiama `bindService()`, sul server viene invocato il metodo `onBind()` che restituisce al client lo stub per l'oggetto `RemoteService`. Il client può quindi richiamare il metodo remoto come segue:

```
RemoteService service;
. . .
service.remoteMethod(3, 0.14);
```

Internamente, l'ambiente binder di Android gestisce il marshaling dei parametri, trasferendo i parametri convertiti tra i processi e invocando l'implementazione del servizio, nonché restituendo eventuali valori di ritorno al processo client.

3.9 Sommario

- Un processo è un programma in esecuzione e lo stato dell'attività corrente di un processo è rappresentato dal contatore di programma e da altri registri.
- La struttura di un processo in memoria è formata da quattro sezioni differenti: (1) testo, (2) dati, (3) heap e (4) stack.
- Un processo cambia il suo stato durante l'esecuzione. Esistono quattro stati in cui un processo può trovarsi: (1) pronto (ready), (2) in esecuzione (running), (3) in attesa (waiting) e (4) terminato.
- Un blocco di controllo del processo (pcb) è la struttura dati del kernel che rappresenta un processo in un sistema operativo.
- Il ruolo dello scheduler dei processi è selezionare un processo disponibile da eseguire su una cpu.
- Un sistema operativo esegue un cambio di contesto quando passa da un processo a un altro.
- Le chiamate di sistema `fork()` e `CreateProcess()` vengono utilizzate per creare processi su sistemi unix e Windows, rispettivamente.
- Quando si utilizza la memoria condivisa per la comunicazione tra processi, due (o più) processi condividono la stessa regione di memoria, posix fornisce un'api per la memoria condivisa.
- Due processi possono comunicare utilizzando lo scambio di messaggi. Il sistema operativo Mach utilizza lo scambio di messaggi come forma principale di comunicazione tra processi e anche Windows rende disponibile una forma di scambio di messaggi.
- Una pipe fornisce a due processi un canale di comunicazione. Esistono due tipi di pipe, le pipe convenzionali e le named pipe. Le pipe convenzionali sono progettate per la comunicazione tra processi in relazione genitore-figlio, mentre le named pipe sono più generali e consentono a diversi processi di comunicare tra loro.
- I sistemi unix rendono disponibili le pipe convenzionali attraverso la chiamata di sistema `pipe()`. Le pipe convenzionali hanno un'estremità dedicata alla lettura e una dedicata alla scrittura. Un processo genitore può, per esempio, inviare dati alla pipe usando la sua estremità di scrittura, e il processo figlio può leggere i dati dalla sua estremità di lettura. In unix le named pipe si chiamano fifo.
- Anche i sistemi Windows forniscono due forme di pipe: le pipe anonime e le named pipe. Le pipe anonime sono simili alle pipe convenzionali in unix, sono unidirezionali e richiedono una relazione genitore-figlio tra i processi in comunicazione. Le named pipe offrono una forma di comunicazione tra processi più ricca rispetto alle fifo di unix.
- Due forme tipiche di comunicazione client-server sono le socket e le chiamate di procedure remote (rpc). Le socket consentono a due processi su macchine diverse di comunicare attraverso una rete. Le rpc astraggono il concetto di chiamata di funzione (o procedura) in modo tale che una funzione possa essere invocata da un altro processo, probabilmente in esecuzione su un computer distinto.
- Il sistema operativo Android, per mezzo del suo ambiente binder, utilizza le rpc come forma di comunicazione tra processi.

Esercizi di ripasso

3.1 Con riferimento al programma mostrato nella Figura 3.30, descrivete l'output prodotto alla LINEA A.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* processo figlio */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* processo padre */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINEA A */
        return 0;
    }
}
```

Figura 3.30 Quale output sarà prodotto alla LINEA A?

3.2 Considerando anche il processo padre iniziale, quanti processi vengono creati dal programma della Figura 3.31?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
```

```

/* crea mediante fork un processo figlio */

fork();

/* crea un altro processo figlio */

fork();

/* e ne crea un altro ancora */

fork();

return 0;

}

```

Figura 3.31 Quanti processi vengono creati?

3.3 Le versioni originali del sistema operativo Apple ios non fornivano alcuno strumento di elaborazione concorrente. Discutete tre principali complicazioni che l'elaborazione concorrente aggiunge a un sistema operativo.

3.4 Alcuni sistemi di elaborazione dispongono di multipli set di registri. Descrivete ciò che avviene quando si ha un cambio di contesto nel caso in cui il contesto successivo sia già caricato in un determinato set di registri. Che cosa succede se il contesto successivo è nella memoria (invece che in un set di registri) e tutti i registri sono in uso?

3.5 Quando un processo crea un nuovo processo utilizzando l'istruzione `fork()`, quale dei seguenti stati è condiviso tra il processo padre e il processo figlio?

- a. Stack
- b. Heap
- c. Segmenti di memoria condivisa

3.6 A proposito del meccanismo rpc, considerate la semantica “esattamente una volta” (*exactly once*). L'algoritmo che implementa questa semantica funziona correttamente anche se il messaggio ack che restituisce al client va perso a causa di un problema di rete? Descrivete la sequenza di messaggi scambiati e indicate se la semantica “esattamente una volta” viene ancora preservata.

3.7 Assumendo che un sistema distribuito sia soggetto a malfunzionamento del server, quali meccanismi sarebbero richiesti per garantire la semantica “esattamente una volta” per l'esecuzione di rpc?

Esercizi

3.8 Descrivete le azioni intraprese dal kernel nell'esecuzione di un cambio di contesto tra processi.

3.9 Costruite un albero di processi simile a quello nella Figura 3.7. Per ottenere informazioni sui processi su sistemi unix o Linux utilizzate il comando `ps -ael`. Utilizzate il comando `man ps` per ottenere più informazioni sul comando `ps`. Il task manager di Windows non mostra l'id del processo padre, ma lo strumento *process monitor*, disponibile su technet.microsoft.com, è in grado di mostrare l'albero dei processi.

3.10 Spiegate il ruolo del processo `init` (o `systemd`) su sistemi unix e Linux, con riferimento alla terminazione dei processi.

3.11 Compreso il processo padre iniziale, quanti processi vengono creati dal programma mostrato nella Figura 3.32?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int i;
    for (i = 0; i < 4; i++)
        fork();
    return 0;
}
```

Figura 3.32 Quanti processi vengono creati?

3.12 Dite in quali circostanze la linea di codice contrassegnata `printf ("LINE J")` nella Figura 3.33 sarà raggiunta.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    pid_t pid;
    /* crea un processo figlio */
    pid = fork();
```

```

if (pid < 0) { /* si è verificato un errore */

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0) { /* processo figlio */

execvp("/bin/ls","ls",NULL);

printf("LINE J");

}

else { /* processo padre */

/* il padre attende che il figlio termini */

wait(NULL);

printf("Child Complete");

}

return 0;

}

```

Figura 3.33 Quando sarà raggiunta la linea printf("LINE J")?

3.13 In riferimento al programma nella Figura 3.34 identificate i valori dei `pid` alle linee A, B, C e D. (Assumete che i `pid` effettivi del padre e del figlio siano rispettivamente 2600 e 2603).

```

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int main()

{

pid_t pid, pid1;

/* crea mediante fork un processo figlio */

pid = fork();

if (pid < 0) { /* errore */

fprintf(stderr, "Fork Failed");

return 1;

}

```

```

else if (pid == 0) { /* processo figlio */

    pid1 = getpid();

    printf("child: pid = %d",pid); /* A */

    printf("child: pid1 = %d",pid1); /* B */

}

else { /* processo padre */

    pid1 = getpid();

    printf("parent: pid = %d",pid); /* C */

    printf("parent: pid1 = %d",pid1); /* D */

    wait(NULL);

}

return 0;

}

```

Figura 3.34 Quali sono i valori dei pid?

3.14 Fornite un esempio di situazione nella quale le pipe convenzionali siano più adatte delle named pipe e un esempio di situazione nella quale le named pipe siano invece più indicate delle pipe convenzionali.

3.15 In riferimento al meccanismo rpc, descrivete i possibili effetti negativi della mancata implementazione della semantiche “al massimo una volta” o di quella “esattamente una volta”. Considerate una possibile applicazione di un meccanismo che non le implementi.

3.16 Descrivete l’output del programma nella Figura 3.35 alle LINEE X e Y.

```

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()

{

    int i;

    pid_t pid;

    pid = fork();

    if (pid == 0) {

```

```

        for (i = 0; i < SIZE; i++) {

            nums[i] *= -i;

            printf("CHILD: %d ",nums[i]); /* LINEA X */

        }

    }

else if (pid > 0) {

    wait(NULL);

    for (i = 0; i < SIZE; i++)

        printf("PARENT: %d ",nums[i]); /* LINEA Y */

}

return 0;

}

```

Figura 3.35 Quale output sarà prodotto alla LINEA X e alla LINEA Y?

3.17 Analizzate vantaggi e svantaggi delle tecniche elencate di seguito, considerando sia il punto di vista del sistema sia quello del programmatore.

- a. Comunicazione sincrona e asincrona.
- b. Gestione automatica o esplicita del buffer.
- c. Trasmissione per copia e trasmissione per riferimento.
- d. Messaggi a lunghezza fissa e variabile.

3.18 Scrivete, in un sistema unix o Linux, un programma C che crei un processo figlio che alla fine diventi un processo zombie. Questo processo zombie deve rimanere nel sistema per almeno 10 secondi. Gli stati dei processi possono essere ottenuti con il comando

```
ps -1
```

Gli stati sono riportati nella colonna **s**, i processi con stato **z** sono processi zombie. L'identificatore di processo (pid) del processo figlio viene mostrato nella colonna **PID**, quello del genitore nella colonna **PPID**.

Il modo più semplice per verificare che il processo figlio sia davvero uno zombie è probabilmente quello di eseguire il programma che avete scritto in background (utilizzando l'opzione **&**) e quindi utilizzare il comando **ps -1** per determinare se il figlio è un processo zombie. Poiché non vogliamo troppi processi zombie presenti nel sistema, è necessario rimuovere quello che si è creato. Il modo più semplice per farlo è quello di terminare il genitore con il comando **kill**. Per esempio, se l'id di processo del genitore è 4884, dobbiamo scrivere

```
kill -9 4884
```

3.19 Il pid manager di un sistema operativo è responsabile della gestione degli identificatori di processo. Quando un processo viene creato, il gestore dei pid gli assegna un pid univoco, che viene restituito al gestore quando il processo termina la sua esecuzione, in modo che il gestore possa in seguito riassegnare il pid ad altri processi. Gli identificatori di processo vengono ampiamente descritti

nel Paragrafo 3.3.1. L'aspetto più importante da comprendere è che gli identificatori di processo devono essere univoci; non possono esistere due processi attivi con lo stesso pid.

Utilizzate le seguenti costanti per individuare l'intervallo di valori possibili dei pid:

```
# define MIN_PID 300  
  
# define MAX_PID 5000
```

È possibile utilizzare qualsiasi struttura dati per rappresentare la disponibilità di identificatori di processo. Una strategia è quella di seguire l'esempio di Linux e utilizzare una bitmap in cui il valore 0 alla posizione i indica che il pid di valore i è disponibile e un valore pari a 1 indica che il pid di valore i è attualmente in uso.

Implementate le seguenti api per ottenere e rilasciare un pid:

- `int allocate_map(void)` – Crea e inizializza una struttura dati per rappresentare i pid; restituisce -1 in caso di insuccesso e 1 in caso di successo.
- `int allocate_pid(void)` – Alloca e restituisce un pid, restituisce -1 se non è possibile assegnare un pid (tutti i pid sono in uso).
- `void release_pid(int pid)` – Rilascia un pid.

3.20 La congettura di Collatz ha a che fare con quello che succede quando si prende un qualsiasi numero intero n positivo e si applica il seguente algoritmo:

$$n = \begin{cases} n / 2, & \text{se } n \text{ è pari} \\ 3 \times n + 1, & \text{se } n \text{ è dispari} \end{cases}$$

La congettura afferma che quando questo algoritmo viene applicato iterativamente, per ogni intero positivo in ingresso alla fine si raggiungerà il valore 1. Per esempio, se $n = 35$, la sequenza prodotta è

35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Scrivete un programma C che utilizzi la chiamata di sistema `fork()` per generare questa sequenza nel processo figlio. Il numero di partenza sarà fornito dalla riga di comando. Per esempio, se il valore 8 viene passato come parametro da riga di comando, il processo figlio restituirà 8, 4, 2, 1. Poiché i processi padre e figlio hanno ognuno le proprie copie dei dati, la sequenza verrà necessariamente restituita dal figlio. Fate in modo che il genitore invochi la chiamata `wait()` per attendere che il processo figlio termini prima di uscire dal programma. Eseguite il necessario controllo degli errori per garantire che dalla riga di comando venga passato un numero intero positivo.

3.21 Nel Problema 3.20 il processo figlio deve restituire la sequenza di numeri generati dall'algoritmo specificato dalla congettura di Collatz, perché il genitore e il figlio hanno ognuno le proprie copie dei dati. Un altro approccio per la realizzazione di questo programma è quello di creare un oggetto di memoria condivisa tra il padre e il figlio. Questa tecnica permette al figlio di scrivere il contenuto della sequenza nell'oggetto memoria condivisa. La sequenza può quindi essere restituita dal genitore nel momento in cui il figlio completa la sua attività. Poiché la memoria è condivisa, tutte le modifiche che il figlio applica si rifletteranno nel processo padre.

Questo programma sarà strutturato utilizzando memoria posix come descritto nel Paragrafo 3.7.1. Il processo padre procederà attraverso le seguenti fasi:

- Costruire l'oggetto memoria condivisa (`shm_open()`, `ftruncate()`, `mmap()`).
- Creare il processo figlio e attendere la sua terminazione.
- Restituire il contenuto della memoria condivisa.
- Rimuovere l'oggetto memoria condivisa.

Uno dei problemi che possono verificarsi con l'utilizzo di processi cooperanti riguarda la sincronizzazione. In questo esercizio i processi padre e figlio devono essere coordinati in modo che il genitore non restituisca la sequenza finché il figlio non termini l'esecuzione. I due processi vengono sincronizzati utilizzando la chiamata di sistema `wait()`: il processo padre invocherà una `wait()` e verrà così sospeso fino al termine dell'esecuzione del processo figlio.

3.22 Il Paragrafo 3.8.1 descrive i numeri di porta inferiori a 1024 come numeri ben noti, perché forniscono servizi standard. La porta 17 offre il servizio *quote-of-the-day* (citazione del giorno). Quando un client si connette alla porta 17 di un server, il server risponde restituendo la citazione per quel giorno.

Modificate il server di data mostrato nella Figura 3.27 in modo che restituisca una citazione del giorno al posto della data corrente. Le citazioni devono essere in caratteri ascii stampabili e devono contenere meno di 512 caratteri, ma sono ammesse più righe. Poiché la porta 17 è ben nota e quindi non disponibile, mettete il server in ascolto sulla porta 6017. Il client mostrato nella Figura 3.26 può essere utilizzato per leggere le citazioni restituite dal server.

3.23 Un haiku è una poesia di tre righe in cui la prima riga contiene cinque sillabe, la seconda riga contiene sette sillabe e la terza riga contiene cinque sillabe. Scrivete un server di haiku che resta in ascolto sulla porta 5575: quando un client si connette a questa porta il server risponde con un haiku. Il client mostrato nella Figura 3.26 può essere utilizzato per leggere le stringhe restituite dal server di haiku.

3.24 Un server eco è un server che restituisce ai client esattamente ciò che essi gli inviano. Se un client, per esempio, invia al server la stringa *Ciao !*, il server risponderà con gli stessi dati: ossia, invierà al client la stringa *Ciao !*. Scrivete un server eco usando la api Java descritta nel Paragrafo 3.8.1. Il server rimarrà in attesa dei client tramite il metodo `accept()`. Dopo aver accettato una connessione, il client eseguirà il ciclo seguente:

- leggerà i dati dalla socket connessa con il client, ponendoli in un buffer;
- scriverà i contenuti del buffer sulla socket connessa con il client.

Il server uscirà dal ciclo una volta stabilito che il client ha chiuso la connessione. Il server nella Figura 3.27 impiega la classe Java `java.io.BufferedReader`, che estende la classe `java.io.Reader`, usata per leggere flussi di caratteri. Il server eco, però, potrebbe ricevere dati di altro tipo dal client – per esempio dati binari. La classe `java.io.InputStream` tratta i dati come byte, e non come caratteri. È quindi necessario che il server eco impieghi una classe che estenda `java.io.InputStream`.

Il metodo `read()` di `java.io.InputStream` restituisce `-1` quando il client ha chiuso la connessione.

3.25 Utilizzando pipe convenzionali, scrivete un programma nel quale un processo manda una stringa a un secondo processo, il quale cambia le lettere maiuscole del messaggio in minuscole, e viceversa, per poi restituire il risultato al primo processo. Per esempio, se il primo processo manda il messaggio *Hi There*, il secondo restituirà *hI tHERE*. Ciò richiede l'utilizzo di due pipe, una per mandare il messaggio originale dal primo al secondo processo e l'altra per mandare il messaggio modificato dal secondo processo al primo. Potete scrivere il programma utilizzando pipe in Windows o in unix.

3.26 Scrivete un programma `filecopy` per la copia di file, usando le pipe convenzionali. Questo programma riceverà in ingresso due parametri: il primo è il nome del file che deve essere copiato, il secondo è il nome del file copia. Il programma creerà una pipe convenzionale e scriverà i contenuti del file da copiare nella pipe. Il processo figlio leggerà il file dalla pipe e lo scriverà nel suo file di destinazione. Se per esempio invochiamo il programma come segue:

```
filecopy input.txt copy.txt
```

il file `input.txt` sarà scritto sulla pipe. Il processo figlio leggerà i contenuti del file e li scriverà nel file di destinazione `copy.txt`. Potete scrivere il programma utilizzando pipe in Windows o in unix.

Progetto 1 – Shell di unix e cronologia

Questo progetto consiste nel realizzare in linguaggio C un'interfaccia shell che accetta comandi utente ed esegue ogni comando in un processo separato. Questo progetto può essere completato in qualsiasi sistema Linux, unix o Mac os X.

Un'interfaccia shell fornisce all'utente un prompt e resta in attesa di un comando. L'esempio seguente mostra il prompt `osh>` e il successivo comando dell'utente: `cat prog.c`. (Questo comando consente di visualizzare il file `prog.c` sul terminale mediante il comando unix `cat`).

```
osh> cat prog.c
```

Una tecnica per implementare un'interfaccia shell è di fare in modo che il processo padre legga prima di tutto ciò che l'utente scrive sulla linea di comando (in questo caso `cat prog.c`) e quindi crei un processo figlio separato che esegua il comando. Se non diversamente specificato, il processo padre attende che il figlio termini prima di continuare. Questo è simile come funzionalità alla creazione di un nuovo processo illustrata nella Figura 3.9. Tuttavia, le shell unix di solito permettono anche che il processo figlio resti in esecuzione in background o in maniera concorrente. Per fare ciò, si aggiunge una `&` alla fine del comando.

Così, se riscriviamo il comando precedente come

```
osh> cat prog.c &
```

i processi padre e figlio resteranno in esecuzione concorrentemente.

Il processo figlio viene creato utilizzando la chiamata di sistema `fork()` e il comando dell'utente viene eseguito utilizzando una delle chiamate di sistema della famiglia `exec()` (come descritto nel Paragrafo 3.3.1).

Un programma in C che fornisce le operazioni generali di una shell a riga di comando è mostrato nella Figura 3.36. La funzione `main()` presenta il prompt `osh>` e definisce le azioni da intraprendere dopo che l'input dell'utente è stato letto. La funzione `main()` continua a ciclare finché `should_run` resta uguale a 1; quando l'utente immette `exit` al prompt, il programma imposterà `should_run` a 0 e terminerà.

```
#include <stdio.h>

#include <unistd.h>

#define MAX_LINE 80 /* Lunghezza massima di un comando */

int main(void)

{

    char *args[MAX_LINE/2 + 1]; /* Argomenti della riga di comando */

    int should_run = 1; /* Flag per determinare quando uscire dal programma */

    while (should_run) {

        printf("osh>");

        fflush(stdout);

        /**
         * Dopo aver letto l'input dell'utente i passi sono:

         * (1) creare un processo figlio usando la fork()

         * (2) il processo figlio invoca la execvp()

         * (3) se il comando include la &, il padre invoca la wait()

         */

    }

    return 0;

}
```

Figura 3.36 Schema di una semplice shell.

Questo progetto è organizzato in due parti: (1) creazione del processo figlio ed esecuzione del comando nel processo figlio, (2) modifica della shell per consentire di mantenere una cronologia.

Parte I – Creazione di un processo figlio

Il primo compito è quello di modificare la funzione `main()` nella Figura 3.36 in modo da creare un processo figlio che esegua il comando specificato dall'utente. Ciò richiederà l'analisi di ciò che l'utente ha inserito, la sua suddivisione in token separati e la memorizzazione dei token in un array di stringhe di caratteri (`args` nella Figura 3.36). Per esempio, se l'utente immette il comando `ps -ael` al prompt `osh>`, i valori memorizzati nell'array `args` sono:

```
args[0] = "ps"  
args[1] = "-ael"  
args[2] = NULL
```

L'array args sarà passato alla funzione `execvp()` che ha il seguente prototipo:

```
execvp (char *command, char *params[]);
```

Qui, `command` rappresenta il comando da eseguire e `params` memorizza i parametri per questo comando. In questo progetto, la funzione `execvp()` va invocata come `execvp(args[0], args)`. Assicuratevi di controllare se l'utente ha incluso un `&` per determinare se il processo padre deve o meno attendere la terminazione del figlio.

Parte II – Creazione della cronologia

Il compito successivo è quello di modificare il programma dell'interfaccia shell in modo da fornire una funzionalità di cronologia per consentire all'utente di accedere ai comandi immessi più di recente. Utilizzando la cronologia, l'utente sarà in grado di accedere a un massimo di 10 comandi. I comandi saranno numerati a partire da 1, e la numerazione proseguirà oltre il 10. Per esempio, se l'utente ha inserito 35 comandi, i 10 comandi più recenti saranno numerati da 26 a 35.

L'utente sarà in grado di elencare la cronologia dei comandi digitando

```
history
```

al prompt `osh>`. Si supponga, per esempio, che la cronologia dei comandi sia costituita, dal più al meno recente, da:

```
ps, ls -l, top, cal, who, date
```

Il comando `history` restituirà:

```
6 ps  
5 ls -l  
4 top  
3 cal  
2 who  
1 date
```

Il vostro programma deve supportare due tecniche per il recupero dei comandi dalla cronologia:

1. Quando l'utente inserisce `!!` viene eseguito il comando più recente.
2. Quando l'utente immette un singolo `!` seguito da un numero intero N , viene eseguito l' N -esimo comando nella cronologia.

Proseguendo il nostro esempio precedente, se l'utente inserisce `!!` verrà eseguito il comando `ps`; se l'utente immette `!3`, verrà eseguito il comando `cal`. Ogni comando eseguito in questo modo deve essere visualizzato sullo schermo dell'utente. Il comando deve anche essere ricollocato nel buffer della cronologia come ultimo comando eseguito.

Il programma deve effettuare una gestione di base degli errori. Se non ci sono comandi nella cronologia all'immissione di `!!` deve essere visualizzato un messaggio `Nessun comando nella cronologia`. Se non c'è nessun comando corrispondente al numero inserito dopo un singolo `!` il programma deve visualizzare il messaggio `Comando non presente nella cronologia`.

Progetto 2 – Modulo del kernel di Linux per l’elenco dei task

In questo progetto verrà scritto un modulo del kernel che elenca tutti i task correnti in un sistema Linux. Assicurarsi prima di iniziare questo progetto di rivedere il progetto di programmazione del Capitolo 2, che si occupa di creazione di moduli del kernel Linux. Il progetto può essere realizzato utilizzando la macchina virtuale Linux fornita con questo testo.

Parte I – Iterazione lineare sui task

Come illustrato nel Paragrafo 3.1, il blocco di controllo di un processo (pcb) in Linux è rappresentato dalla struttura `task_struct`, che si trova nel file di include `<linux/sched.h>`. In Linux la macro `for_each_process()` permette di realizzare in modo semplice l’iterazione su tutti i task in corso nel sistema:

```
#include <linux/sched.h>

struct task_struct *task;

for each process(task) {

    /* A ogni iterazione task punta al prossimo task */

}
```

I vari campi di `task_struct` possono essere visualizzati mentre il programma cicla per mezzo della macro `for each process()`.

Parte I – Esercizio

Progettate un modulo del kernel che consenta di scorrere tutti i task del sistema utilizzando la macro `for_each_process()`. In particolare, si visualizzi il nome (noto come nome del file eseguibile), lo stato e l’id di processo di ogni task. (Dovrete probabilmente leggere la struttura `task_struct` in `<linux/sched.h>` per ottenere i nomi di questi campi). Scrivete questo codice nel punto di ingresso del modulo in modo che i suoi contenuti verranno visualizzati nel buffer di log del kernel, che può essere letto con il comando `dmesg`. Per verificare che il codice funzioni correttamente, confrontate il contenuto del buffer di log del kernel con l’output del seguente comando, che elenca tutti i task del sistema:

```
ps -el
```

I due valori dovrebbero essere molto simili. Poiché i task sono dinamici, tuttavia, è possibile che alcuni task appaiano in una lista ma non nell’altra.

Parte II – Iterazione sui task con un albero di ricerca in profondità

La seconda parte di questo progetto prevede l’iterazione su tutti i task nel sistema utilizzando una ricerca in profondità (dfs: Depth-First Search). A titolo di esempio: l’iterazione dfs sui processi nella Figura 3.7 è 1, 8415, 8416, 9298, 9204, 2, 6, 200, 3028, 3610, 4005.

Linux mantiene il suo albero dei processi come una serie di liste. Esaminando la struttura `task_struct` in `<linux/sched.h>`, vediamo due oggetti `struct list_head`:

```
children
```

```
e
```

```
sibling
```

Questi oggetti sono puntatori a una lista dei figli del task e a una lista dei suoi fratelli. Linux mantiene anche riferimenti al task `init` (`struct task_struct init_task`). Utilizzando queste informazioni e le macro per operare sulle liste, siamo in grado di scorrere i figli di `init` come segue:

```
struct task_struct *task;

struct list_head *list;

list_for_each(list, &init_task->children) {

    task = list_entry(list, struct task_struct, sibling);

    /* task punta al successivo figlio nella lista */

}
```

La macro `list_for_each()` riceve due parametri, entrambi di tipo `struct list_head`:

- un puntatore alla testa della lista da scorrere;
- un puntatore al nodo di testa della lista da scorrere.

A ogni passo di `list_for_each()` il primo parametro è impostato alla struttura `list` del figlio successivo. Questo valore viene poi utilizzato per ottenere ogni struttura nella lista usando la macro `list_entry()`.

Parte II – Esercizio

Progettate un modulo del kernel che itera su tutte le attività del sistema, a partire dal task `init`, utilizzando un albero dfs. Proprio come nella prima parte di questo progetto, stampate il nome, lo stato e il pid di ogni task. Eseguite questa iterazione nel modulo di ingresso del kernel in modo che il suo output venga visualizzato nel buffer di log.

Se si stampano tutti i task del sistema è possibile vedere molti più task rispetto a quelli ottenuti con il comando `ps -ael`, poiché alcuni thread appaiono come figli, ma non si presentano come processi ordinari. Pertanto, per verificare l'output dell'albero dfs, utilizzate il comando

```
ps -eLF
```

Questo comando elenca tutti i task del sistema, inclusi i thread. Per verificare di aver effettivamente eseguito l'iterazione dfs in maniera appropriata, si dovranno esaminare le relazioni tra i vari task restituiti dal comando `ps`.

CAPITOLO 4

Thread e concorrenza

Nel modello introdotto nel Capitolo 3 si è assunto che un processo sia un programma in esecuzione con un unico percorso di controllo. Quasi tutti i sistemi operativi moderni permettono tuttavia che un processo possa avere più percorsi di controllo chiamati *thread*. Nei sistemi multicore, dotati di più cpu, diventa sempre più importante individuare le opportunità di eseguire più thread in parallelo.

In questo capitolo sono introdotti diversi concetti, con le relative criticità, associati ai sistemi elaborativi *multithread*, tra i quali un'approfondita descrizione delle api per le librerie di thread di Pthreads, Windows e Java. Inoltre, si esplorano diverse nuove tecniche che permettono di astrarre il concetto di creazione di thread e consentono agli sviluppatori di concentrarsi sull'individuazione delle possibilità di parallelismo, lasciando alle caratteristiche del linguaggio e alle api la gestione dei dettagli relativi alla creazione e alla gestione dei thread. Si esaminano molti aspetti legati alla programmazione multithread e il modo in cui essa influenza la progettazione dei sistemi operativi; infine, si analizza il modo in cui alcuni sistemi operativi moderni, come Windows e Linux, gestiscono i thread a livello kernel.

4.1 Introduzione

Un thread è l'unità di base d'uso della cpu e comprende un identificatore di thread (id), un contatore di programma, un insieme di registri e una pila (*stack*). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche processo pesante (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di svolgere più compiti in modo concorrente. La Figura 4.1 mostra la differenza tra un processo tradizionale, a singolo thread, e uno multithread.

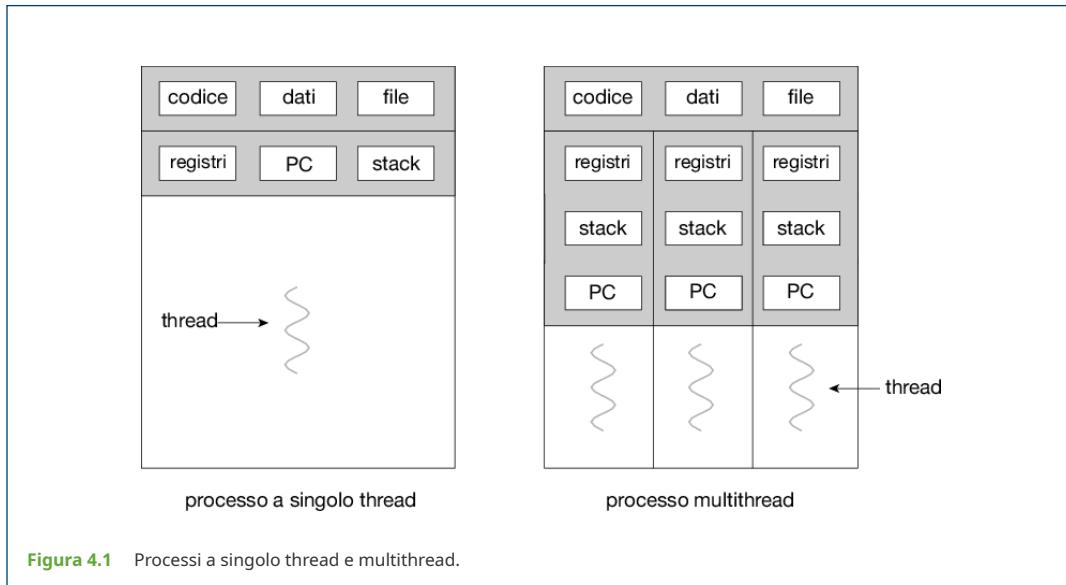


Figura 4.1 Processi a singolo thread e multithread.

4.1.1 Motivazioni

La maggior parte delle applicazioni per i moderni computer è multithread. Di solito, un'applicazione si codifica come un processo a sé stante comprendente più thread di controllo. Alcuni esempi di applicazioni multithread sono i seguenti.

- Un'applicazione che crea miniature di foto (thumbnails) da una raccolta di immagini può utilizzare un thread distinto per generare una miniatura di ciascuna immagine.
- Un web browser può avere un thread per rappresentare sullo schermo immagini e testo e un altro thread per scaricare i dati dalla rete.
- Un word processor può avere un thread per la rappresentazione grafica, uno per la risposta all'input da tastiera e uno per la correzione ortografica e grammaticale eseguita in background.

Le applicazioni possono anche essere progettate per sfruttare le capacità di elaborazione sui sistemi multicore. Tali applicazioni possono eseguire diverse attività che utilizzano intensivamente la cpu in parallelo sui diversi core di elaborazione.

In alcune situazioni una singola applicazione deve poter gestire molti compiti simili tra loro. Per esempio, un server web accetta dai client richieste di pagine web, immagini, suoni e altro. Per un server web intensamente utilizzato potrebbero esservi molti (forse migliaia) client che vi accedono in modo concorrente; se il server web fosse eseguito come un processo tradizionale a singolo thread, esso sarebbe in grado di soddisfare un solo client alla volta e un client potrebbe dover aspettare molto a lungo prima che la sua richiesta venga servita.

Una soluzione è eseguire il server come un singolo processo che accetta richieste. Quando ne riceve una, il server crea un processo separato per eseguirla. In effetti, questo metodo di creazione di processi era molto usato prima che si diffondesse la possibilità di gestione dei thread. La creazione dei processi è molto onerosa, sia a livello di tempi sia di risorse; se il nuovo processo si deve occupare degli stessi compiti del processo corrente, non c'è alcuna ragione di accettare l'intero carico che la sua creazione comporta. Generalmente è più conveniente impiegare un processo multithread. Nel caso del server web, se il processo è multithread, il server creerà un thread distinto per ricevere le richieste dei client; in presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla. Il tutto è illustrato nella Figura 4.2.

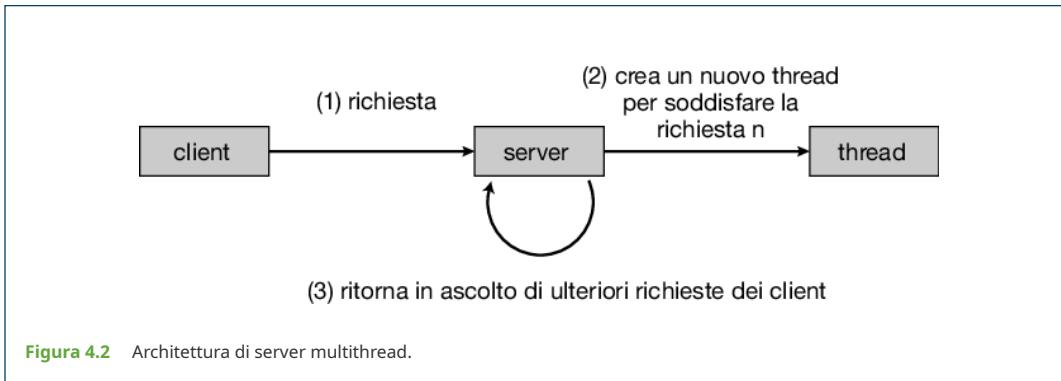


Figura 4.2 Architettura di server multithread.

La maggior parte dei kernel dei sistemi operativi è multithread. Per esempio, durante l'avvio del sistema Linux vengono creati diversi thread a livello kernel e ogni thread esegue un'attività specifica, per esempio la gestione dei dispositivi, della memoria o delle interruzioni. Su un sistema Linux in esecuzione si può usare il comando `ps -ef` per visualizzare i thread a livello kernel. Esaminando l'output di questo comando si può notare il thread `kthreadd` (con `pid = 2`), che funge da genitore di tutti gli altri thread a livello kernel.

Molte applicazioni, tra cui gli algoritmi di ordinamento e gli algoritmi per alberi e grafi, possono trarre vantaggio dal multithreading. Inoltre, i programmatore specializzati nella risoluzione di problemi che richiedono un uso intensivo della cpu in ambito data mining, grafica e intelligenza artificiale possono sfruttare la potenza dei moderni sistemi multicore progettando soluzioni che utilizzano l'esecuzione in parallelo.

4.1.2 Vantaggi

I vantaggi della programmazione multithread si possono classificare in quattro categorie principali.

1. Tempo di risposta. Rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta all'utente. Questa caratteristica è particolarmente utile nella progettazione di interfacce utente. Per esempio, si consideri quello che succede quando un utente fa clic su un pulsante che provoca l'esecuzione di un'operazione che richiede diverso tempo. Un'applicazione a thread singolo resterebbe bloccata fino al completamento dell'operazione. Al contrario, se l'operazione viene eseguita in un thread separato, l'applicazione rimane attiva per l'utente.
2. Condivisione delle risorse. I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa e lo scambio di messaggi. Queste tecniche devono essere esplicitamente messe in atto dal programmatore. Tuttavia, i thread condividono per default la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
3. Economia. Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza nell'overhead richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione dei thread richiede in generale meno tempo e meno memoria, e il cambio di contesto tra thread è più rapido.
4. Scalabilità. I vantaggi della programmazione multithread sono ancora maggiori nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Invece un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo. Esporeremo questo tema nel prossimo paragrafo.

4.2 Programmazione multicore

Nei primi tempi della progettazione dei calcolatori, in risposta alla necessità di una maggiore potenza di calcolo, si è passati dai sistemi a singola cpu ai sistemi multi-cpu. Una simile tendenza, più recente, nel progetto dell'architettura dei sistemi consiste nel montare diversi *core* di elaborazione su un unico chip; ogni unità appare al sistema operativo come un processore separato (Paragrafo 1.3.2). Sia che i core appartengano allo stesso chip o a più chip, noi chiameremo questi sistemi multicore o multiprocessore. La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi multiprocessori e aiuta a sfruttare al meglio la concorrenza. Si consideri un'applicazione con quattro thread. In un sistema con un singolo core, “esecuzione concorrente” significa solo che l'esecuzione dei thread è avvicendata nel tempo (*interleaved*) (Figura 4.3), perché la cpu è in grado di eseguire un solo thread alla volta. Su un sistema multicore, invece, “esecuzione concorrente” significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core (Figura 4.4).

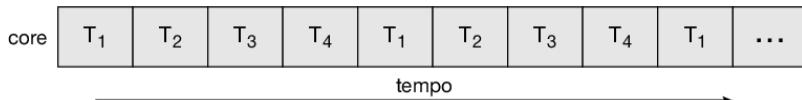


Figura 4.3 Esecuzione concorrente su un sistema a singolo core.

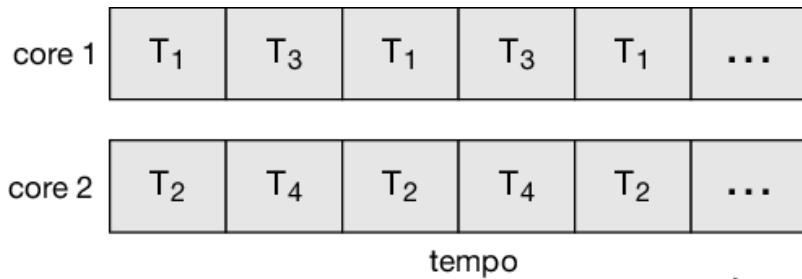


Figura 4.4 Esecuzione parallela su un sistema multicore.

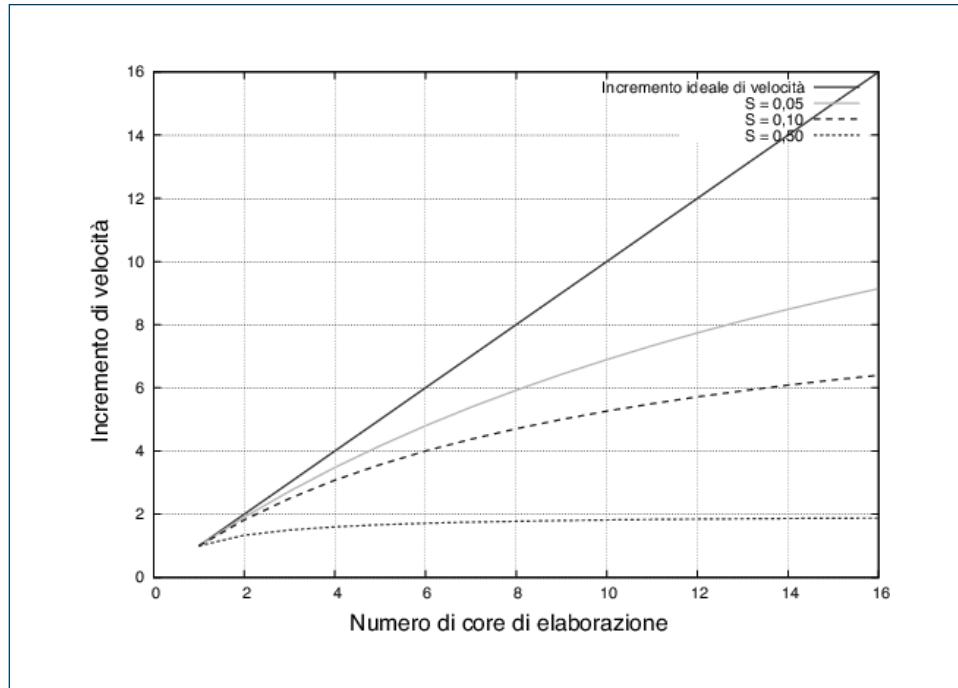
Si noti la distinzione tra *parallelismo* e *concorrenza*. Un sistema concorrente supporta più task permettendo a ciascuno di progredire nell'esecuzione. Un sistema parallelo, invece, può eseguire simultaneamente più di un task. È dunque possibile avere concorrenza senza parallelismo. Prima dell'avvento dei processori smp e delle architetture multicore, la maggior parte dei sistemi era dotata di un singolo processore. Gli scheduler della cpu erano progettati per fornire l'illusione di parallelismo, mediante una rapida commutazione tra processi nel sistema, consentendo in tal modo a ogni processo di fare progressi. Tali processi erano eseguiti in maniera concorrente, ma non in parallelo.

LEGGE DI AMDAHL

La legge di Amdahl è una formula che permette di determinare i potenziali guadagni in termini di prestazioni ottenuti dall'aggiunta di ulteriori core di elaborazione, nel caso di applicazioni che contengono sia componenti seriali (non parallele) sia componenti parallele. Indicando con S la porzione di applicazione che deve essere realizzata serialmente su un sistema con N core di elaborazione, la formula è la seguente:

$$\text{incremento di velocità} \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Come esempio, supponiamo di avere un'applicazione che sia al 75% parallela e al 25% seriale. Se eseguiamo l'applicazione su un sistema con due core di elaborazione, possiamo ottenere un incremento di velocità pari a 1,6. Se aggiungiamo due core (per un totale di quattro core), l'incremento di velocità è pari a 2,28. Il grafico che segue illustra la legge di Amdahl in differenti scenari.



Un fatto interessante circa la legge di Amdahl è che per N che tende all'infinito, l'incremento di velocità converge a $1/S$. Per esempio, se il 50 per cento di un'applicazione viene eseguita in maniera seriale, il massimo aumento di velocità è di 2 volte, indipendentemente dal numero di core di elaborazione che aggiungiamo. Questo è il principio fondamentale che sta dietro la legge di Amdahl: la porzione seriale di un'applicazione può avere un effetto dominante sulle prestazioni ottenibili con l'aggiunta di ulteriori core.

4.2.1 Le sfide della programmazione

La tendenza verso i sistemi multicore tiene costantemente sotto pressione i progettisti di sistemi operativi e i programmati di applicazioni, affinché utilizzino al meglio core multipli. I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diversi core per permettere un'esecuzione parallela come quella mostrata nella Figura 4.4. Per i programmati di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread. In generale, possiamo individuare nelle cinque aree seguenti le principali sfide della programmazione dei sistemi multicore.

1. Identificazione dei task. Consiste nell'esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possono essere eseguiti in parallelo su core distinti.
2. Bilanciamento. Nell'identificare i task eseguibili in parallelo, i programmati devono far sì che i vari task eseguano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena di utilizzare core separati.

3. Suddivisione dei dati. Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati su core distinti.
4. Dipendenze dei dati. I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In casi in cui un task dipende dai dati forniti da un altro, i programmatore devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze. Esamineremo queste strategie nel Capitolo 6.
5. Test e debugging. Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di questi programmi è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

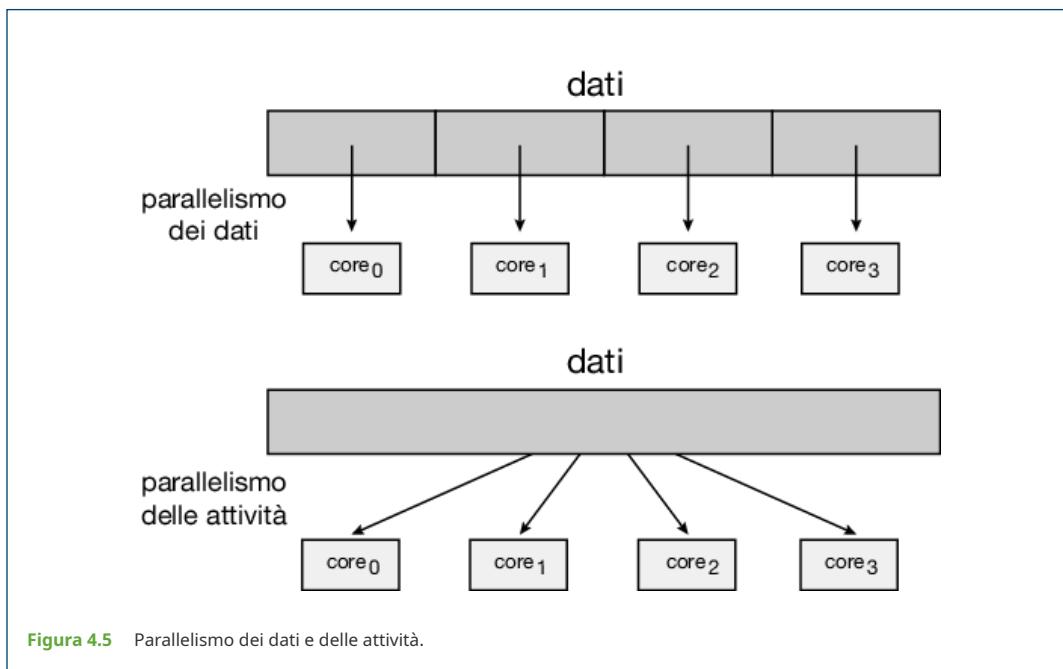
Molti sviluppatori di software sostengono, in ragione dei problemi appena esposti, che l'avvento dei sistemi multicore richiederà in futuro un approccio interamente nuovo al progetto dei sistemi software. (Allo stesso modo, molti docenti di informatica ritengono che lo sviluppo del software debba essere insegnato ponendo maggiore enfasi sulla programmazione parallela).

4.2.2 Tipi di parallelismo

In generale, esistono due tipi di parallelismo: parallelismo dei dati e parallelismo delle attività. Il parallelismo dei dati riguarda la distribuzione di sottosinsiemi dei dati su più core di elaborazione e l'esecuzione della stessa operazione su ogni core. Si consideri, per esempio, l'operazione di somma dei valori contenuti in un vettore di dimensione N . In un sistema con un singolo core, un thread sommerebbe semplicemente gli elementi da $[0]$ a $[N - 1]$. In un sistema dual-core, invece, il thread A , in esecuzione sul core 0, potrebbe sommare gli elementi da $[0]$ a $[N/2 - 1]$, mentre il thread B , in esecuzione sul core 1, potrebbe sommare gli elementi da $[N/2]$ a $[N - 1]$. I due thread sarebbero in esecuzione in parallelo su core di elaborazione distinti.

Il parallelismo delle attività prevede la distribuzione di attività (thread), e non di dati, su più core. Ogni thread realizza un'operazione distinta e thread differenti possono operare sugli stessi dati o su dati diversi. Si consideri ancora il nostro esempio. A differenza della situazione precedente, un esempio di parallelismo delle attività potrebbe coinvolgere due thread, ciascuno dei quali esegue un'unica operazione statistica sull'array di elementi. I thread operano ancora in parallelo su core separati, ma ciascuno sta eseguendo un'operazione diversa.

Fondamentalmente, quindi, il parallelismo dei dati comporta la distribuzione dei dati su più core e il parallelismo delle attività la distribuzione dei task su più core, come mostrato nella Figura 4.5. Tuttavia, parallelismo dei dati e parallelismo delle attività non sono mutuamente esclusivi, e le applicazioni possono utilizzare un ibrido di queste due strategie.



4.3 Modelli di supporto al multithreading

I thread possono essere distinti in thread a livello utente e thread a livello kernel: i primi sono gestiti sopra il livello del kernel e senza il suo supporto; i secondi, invece, sono gestiti direttamente dal sistema operativo. Praticamente tutti i sistemi operativi moderni supportano i thread nel kernel, compresi Windows, Linux e macos.

In ogni caso, deve esistere una relazione tra i thread a livello utente e i thread a livello kernel, come mostrato nella Figura 4.6. In questo paragrafo analizziamo tre opzioni comuni: il modello da molti a uno, il modello da uno a uno e il modello da molti a molti.

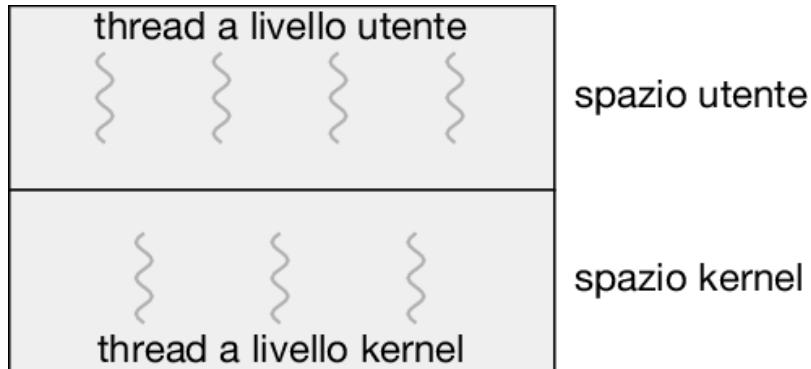


Figura 4.6 Thread a livello utente e a livello kernel.

4.3.1 Modello da molti a uno

Il modello da molti a uno (Figura 4.7) fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. La gestione dei thread risulta efficiente perché viene effettuata da una libreria di thread nello spazio utente (le librerie di supporto ai thread verranno discusse nel Paragrafo 4.4). Tuttavia l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poiché un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multicore; la libreria green threads, disponibile per Solaris e adottata nelle prime versioni di Java, usa questo modello. Tuttavia, pochissimi sistemi utilizzano ancora questo modello a causa della sua incapacità di trarre vantaggio dalla presenza di più core (che costituiscono lo standard nei sistemi elaborativi moderni).

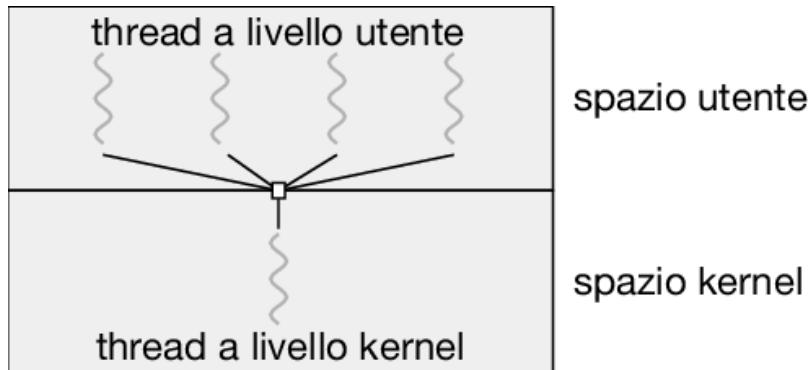


Figura 4.7 Modello da molti a uno.

4.3.2 Modello da uno a uno

Il modello da uno a uno (Figura 4.8) mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore. L'unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel. Poiché il carico dovuto alla creazione di un thread a livello kernel può sovraccaricare le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread supportabili dal sistema. I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.

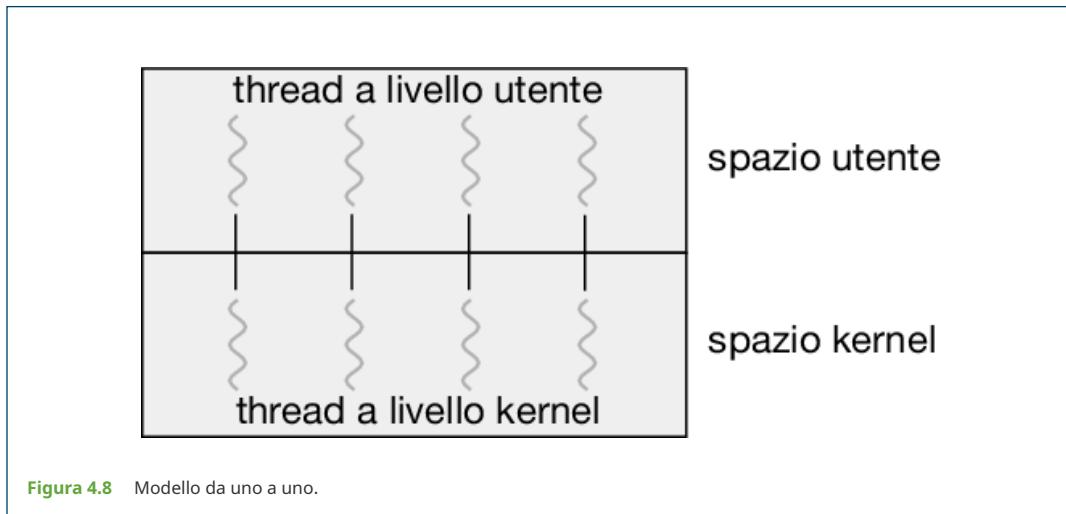


Figura 4.8 Modello da uno a uno.

4.3.3 Modello da molti a molti

Il modello da molti a molti (Figura 4.9) mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel; quest'ultimo può essere specifico per una certa applicazione o per un particolare calcolatore (a un'applicazione potrebbero essere assegnati più thread a livello kernel in un'architettura dotata di otto core rispetto a quanti ne verrebbero assegnati in una con quattro core).

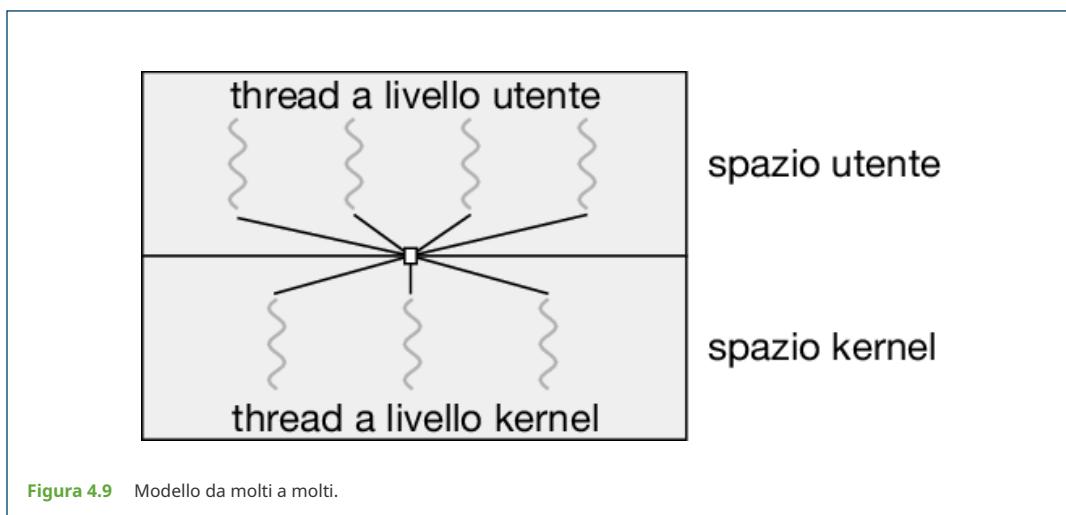
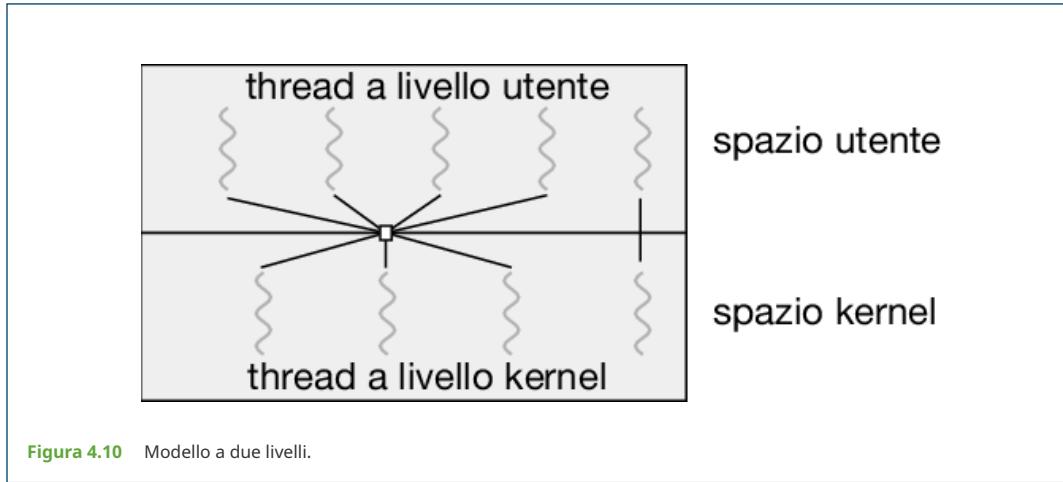


Figura 4.9 Modello da molti a molti.

Consideriamo l'effetto di questo modello sulla concorrenza. Nonostante il modello da molti a uno permetta ai programmati di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta. Il modello da uno a uno permette una maggiore concorrenza, ma i programmati devono stare attenti a non creare troppi thread all'interno di un'applicazione (in qualche caso si possono avere limitazioni sul numero di thread che si possono creare). Il modello da molti a molti non ha alcuno di questi difetti: i programmati possono creare liberamente i thread che ritengono necessari e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata di sistema bloccante, il kernel può schedulare un altro thread.

Una variante del modello da molti a molti mantiene la corrispondenza fra più thread utente e un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread del kernel. Questa variante è anche chiamata modello a due livelli (Figura 4.10).



Anche se il modello da molti a molti sembra essere il più flessibile tra quelli trattati è difficile metterlo in atto nella pratica. Inoltre, con un numero sempre crescente di core di elaborazione presenti nella maggior parte dei sistemi, la limitazione del numero di thread a livello kernel è diventata meno importante e, di conseguenza, la maggior parte dei sistemi operativi utilizza ora il modello da uno a uno. Tuttavia, come vedremo nel Paragrafo 4.5, alcune librerie per la concorrenza richiedono agli sviluppatori di identificare i task che vengono poi mappati sui thread usando il modello da molti a molti.

4.4 Librerie dei thread

Una libreria dei thread fornisce al programmatore una api per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due. Nel primo, la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio utente. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio utente e non in una chiamata di sistema.

Il secondo metodo consiste nell'implementare una libreria a livello kernel, supportata direttamente dal sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della api per la libreria provoca, generalmente, una chiamata di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: Pthreads di posix, Windows e Java. Pthreads, estensione dello standard posix, può essere realizzata sia come libreria a livello utente sia a livello kernel. La libreria di thread Windows è una libreria a livello kernel per i sistemi Windows. La api per la creazione dei thread in Java è gestibile direttamente dai programmi Java. Tuttavia, data la peculiarità di funzionamento della jvm, quasi sempre eseguita sopra un sistema operativo che la ospita, la api di Java per i thread è solitamente implementata per mezzo di una libreria dei thread del sistema ospitante. Perciò, i thread di Java sui sistemi Windows sono in effetti implementati mediante la api Windows; sui sistemi unix e Linux, invece, si adopera spesso Pthreads.

Nel threading di posix e di Windows tutti i dati dichiarati a livello globale, ovvero al di fuori di ogni funzione, sono condivisi tra tutti i thread appartenenti allo stesso processo. Poiché Java non ha alcuna nozione di dati globali, l'accesso ai dati condivisi deve essere assegnato in modo esplicito tra i thread. I dati locali di una funzione sono in genere memorizzati nello stack. Dal momento che ogni thread ha un proprio stack, ogni thread ha la propria copia di dati locali.

Nel seguito affrontiamo i fondamenti della generazione dei thread in queste tre librerie. Come esempio dimostrativo progetteremo un programma multithread che calcola la somma dei primi N interi non negativi in un thread separato, in simboli:

$$sum = \sum_{i=1}^N i$$

Se, per esempio, $N = 5$, si avrebbe $sum = 15$, la somma dei numeri da 0 a 5. Ciascuno dei tre programmi funzionerà inserendo nella riga di comando l'indice superiore N della sommatoria; inserendo 8, quindi, si otterrà come risultato la somma dei valori interi da 0 a 8.

Prima di procedere con i nostri esempi di creazione di thread, introduciamo due strategie generali per la creazione di più thread: il *threading asincrono* e il *threading sincrono*. Nel threading asincrono, una volta che il genitore crea un thread figlio, riprende la sua esecuzione, in modo che genitore e figlio restino in esecuzione concorrentemente. Ogni thread viene eseguito in modo indipendente rispetto agli altri thread e il thread genitore non ha bisogno di conoscere quando suo figlio termina. Poiché i thread sono indipendenti, vi è di solito poca condivisione dei dati tra i thread. Il threading asincrono è la strategia utilizzata nel server multithread illustrato nella Figura 4.2 ed è anche comunemente utilizzato per il progetto di interfacce utente reattive (*responsive*).

Il threading sincrono si verifica quando il thread genitore crea uno o più figli e attende che tutti terminino prima di riprendere l'esecuzione. Tale strategia è detta fork-join. In questo caso, i thread creati dal genitore svolgono il lavoro in maniera concorrente, ma il genitore non può continuare fino al completamento di questo lavoro. Una volta che un thread ha completato il suo lavoro esso termina e si unisce (*join*) con il genitore. Solo dopo che tutti i figli si sono uniti al genitore, questo può riprendere l'esecuzione. In genere, il threading sincrono comporta una significativa condivisione dei dati tra i thread. Per esempio, il thread genitore può combinare i risultati calcolati dai suoi figli. Tutti i seguenti esempi utilizzano il threading sincrono.

4.4.1 Pthreads

Col termine Pthreads ci si riferisce allo standard posix (ieee 1003.1c) che definisce una api per la creazione e la sincronizzazione dei thread. Non si tratta di una *implementazione*, ma di una *specifica* del comportamento dei thread; i progettisti di sistemi operativi possono realizzare la specifica come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; per la maggior parte si tratta di sistemi di tipo unix, tra cui Linux e macos. Anche se Windows non supporta nativamente Pthreads, sono disponibili alcune implementazioni di terze parti per Windows.

Il programma C multithread nella Figura 4.11 esemplifica la api Pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito. Nei programmi Pthreads, i nuovi thread iniziano l'esecuzione a partire da una funzione specificata. Nel programma in esame si tratta della funzione `runner()`. All'inizio dell'esecuzione del programma c'è un unico thread di controllo che parte da `main()`; dopo una fase d'inizializzazione, `main()` crea un secondo thread che inizia l'esecuzione dalla funzione `runner()`. Entrambi i thread condividono la variabile globale `sum`.

```
#include <pthread.h>
#include <stdio.h>
```

```

#include <stdlib.h>

int sum; /* questo dato è condiviso dai thread */

void *runner(void *param); /* i thread chiamano questa funzione */

int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore del thread */

    pthread_attr_t attr; /* insieme di attributi del thread */

    /* imposta gli attributi predefiniti del thread */

    pthread_attr_init(&attr);

    /* crea il thread */

    pthread_create(&tid,&attr,runner,argv[1]);

    /* attende la terminazione del thread */

    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* Il thread viene eseguito in questa funzione */

void *runner(void *param)
{
    int i, upper = atoi(param);

    sum = 0;

    for (i = 1; i <= upper; i++)

        sum += i;

    pthread_exit(0);
}

```

Figura 4.11 Programma multithread in linguaggio C che impiega la api Pthreads.

Esaminiamo il programma più attentamente. Tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione `pthread.h`. La dichiarazione di variabili `pthread_t tid` specifica l'identificatore per il thread da creare. Ogni thread ha un insieme di attributi che includono la dimensione dello stack e informazioni di scheduling. La dichiarazione `pthread_attr_t attr` riguarda la struttura dati per gli attributi del thread, i cui valori si assegnano con la chiamata di funzione `pthread_attr_init(&attr)`. Poiché non sono stati esplicitamente forniti valori per gli attributi, si usano quelli predefiniti. (Nel Capitolo 5 saranno esaminati alcuni degli attributi di scheduling offerti dalle api Pthreads). La chiamata di funzione `pthread_create()` crea un nuovo thread. Oltre all'identificatore del thread e ai suoi attributi, si passa anche il nome della funzione da

cui il nuovo thread inizierà l'esecuzione, in questo caso la funzione `runner()`, e il numero intero fornito come parametro alla riga di comando, e individuato da `argv[1]`.

A questo punto il programma ha due thread: il thread iniziale (o genitore), in `main()`; il thread che esegue la somma (o figlio), in `runner()`. Il programma segue dunque la strategia fork-join descritta in precedenza: dopo aver creato il figlio, il thread padre ne attende il completamento chiamando la funzione `pthread_join()`. Il secondo thread termina quando invoca la funzione `pthread_exit()`. Quando il thread che esegue la somma termina, il thread padre produce in uscita il valore condiviso `sum` della sommatoria.

Questo programma di esempio crea un solo thread. Con il crescente predominio dei sistemi multicore, la scrittura di programmi che contengono più thread è diventata sempre più comune. Un metodo semplice per l'attesa su più thread usando `pthread_join()` è di racchiudere l'operazione all'interno di un semplice ciclo `for`. Per esempio, è possibile effettuare il join di dieci thread utilizzando il codice Pthread mostrato nella Figura 4.12.

```
#define NUM_THREADS 10

/* array di thread da unire */

pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)

pthread_join(workers[i], NULL);
```

Figura 4.12 Codice Pthread per effettuare il join di 10 thread.

4.4.2 Thread in Windows

La tecnica usata dalla libreria Windows per la creazione dei thread può richiamare, per molti versi, quella di Pthreads. Illustriamo la api Windows nel programma C mostrato dalla Figura 4.13. Si noti che per utilizzare la api Windows è necessario includere il file d'intestazione `windows.h`.

```
#include <windows.h>

#include <stdio.h>

DWORD Sum; /* il dato è condiviso tra i thread */

/* il thread viene eseguito in questa funzione separata */

DWORD WINAPI Summation(LPVOID Param)

{

    DWORD Upper = *(DWORD*)Param;

    for (DWORD i = 0; i <= Upper; i++)

        Sum += i;

    return 0;

}
```

```

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    Param = atoi (argv [1]);

    /* crea il thread */

    ThreadHandle = CreateThread(
        NULL, /* attributi di sicurezza di default */
        0, /* dimensione di default dello stack */
        Summation, /* funzione del thread */
        &Param, /* parametri alla funzione del thread */
        0, /* flag di crezione di default */
        &ThreadId); /* restituisce l'identificatore del thread */

    /* adesso aspetta la fine del thread */

    WaitForSingleObject(ThreadHandle,INFINITE);

    /* chiude l'handle del thread */

    CloseHandle(ThreadHandle);

    printf("sum = %d\n",Sum);

}

```

Figura 4.13 Programma multithread in C con l'utilizzo dell'api Windows.

Come nella versione di Pthreads della Figura 4.11, i dati condivisi da thread separati – nella fattispecie, `Sum` – sono globali (il tipo `DWORD` è un intero di 32 bit privo di segno). Dobbiamo inoltre definire la funzione `Summation()` che il nuovo thread eseguirà. A questa funzione è passato un puntatore a `void`, che in Windows è `lpvoid`. Il thread che esegue questa funzione imposta la variabile globale `Sum` al valore della sommatoria da 0 fino al parametro passato a `Summation()`.

Nella api Windows i nuovi thread si generano tramite la funzione `CreateThread()`, che – proprio come in Pthreads – accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione dello stack e un indicatore (`flag`) per segnalare se il thread debba avere inizio nello stato d'attesa. Ci serviremo, nel programma, dei valori di default di questi attributi, che inizialmente non pongono il thread in stato d'attesa, bensì lo rendono eseguibile dallo scheduler della cpu. Una volta creato il nuovo thread, il thread iniziale deve attenderne il completamento prima di produrre in uscita il valore di `Sum`, poiché esso è computato dal nuovo thread. Come si ricorderà, nel programma Pthread (Figura 4.11) il thread iniziale era posto in attesa della terminazione del nuovo thread tramite la funzione `pthread_join()`. La chiamata equivalente nella api Windows è `WaitForSingleObject()`, che causa la sospensione del thread iniziale fintanto che il nuovo thread non abbia terminato.

In situazioni che richiedono l'attesa della terminazione di più thread viene utilizzata la funzione `WaitForMultipleObjects()`, alla quale vengono passati quattro parametri:

1. il numero di oggetti da attendere;
2. un puntatore al vettore di oggetti;
3. un flag che indica se tutti gli oggetti sono stati segnalati;
4. la durata del timeout (o il valore `INFINITE`).

Per esempio, se `THandles` è un array di `HANDLE` di thread di dimensione N , il thread principale può attendere che tutti i suoi figli terminino con la seguente istruzione:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

4.4.3 Thread Java

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di funzionalità per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo – persino un semplice programma, costituito soltanto da un metodo `main()`, è eseguito dalla JVM come un singolo thread. I thread Java sono disponibili su tutti i sistemi che dispongono di una JVM, tra cui Windows, Linux e macOS. L'API Java per i thread è anche disponibile per le applicazioni Android.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e "sovrascrivere" (override) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`.

Questa interfaccia definisce un singolo metodo astratto con segnatura `public void run()`. Il codice nel metodo `run()` di una classe che implementa `Runnable` sarà eseguito in un thread distinto. Un esempio è mostrato di seguito:

```
class Task implements Runnable {  
    public void run() {  
        System.out.println("I am a thread.");  
    }  
}
```

La creazione di un thread in Java richiede di creare un oggetto `Thread` e passare all'oggetto un'istanza di una classe che implementa `Runnable`, per poi invocare il metodo `start()` dell'oggetto `Thread`, come si può vedere nel seguente esempio:

```
Thread worker = new Thread(new Task());  
  
worker.start();
```

L'invocazione del metodo `start()` del nuovo oggetto `Thread` ha il duplice effetto di:

1. allocare la memoria e inizializzare un nuovo thread nella JVM;
2. chiamare il metodo `run()`, cosa che rende il thread eseguibile dalla JVM. Si osservi come il metodo `run()` non sia mai chiamato per via diretta, ma solo tramite la chiamata del metodo `start()`.

Ricordiamo che i thread genitori nelle librerie Pthreads e Windows usano, rispettivamente, `phtread_join()` e `WaitForSingleObject()` per attendere la conclusione del thread che esegue la somma prima di procedere. Il metodo `join()` in Java fornisce una simile funzionalità. (Si noti che `join()` può sollevare una `InterruptedException`, che nel codice abbiamo deciso di non gestire).

```
try {  
  
    worker.join();  
}
```

```
    }

    catch (InterruptedException ie) { }
```

Se il genitore deve attendere la terminazione di diversi thread, il metodo `join()` può essere racchiuso in un ciclo `for` simile a quello mostrato per Pthreads nella Figura 4.12.

ESPRESSIONI LAMBDA IN JAVA

A partire dalla versione 1.8 del linguaggio, Java ha introdotto le espressioni Lambda, che forniscono una sintassi molto più pulita per la creazione di thread. Piuttosto che definire una classe separata che implementa `Runnable`, è possibile utilizzare un'espressione Lambda:

```
Runnable task = () -> {

    System.out.println("I am a thread.");

};

Thread worker = new Thread(task);

worker.start();
```

Le espressioni Lambda, così come funzioni simili chiamate **chiusure** (closure), sono una delle principali caratteristiche dei linguaggi di programmazione funzionale e sono state rese disponibili in diversi linguaggi non funzionali, compresi Python, C++ e C#. Come vedremo negli esempi successivi di questo capitolo, le espressioni Lambda permettono in molti casi di scrivere applicazioni parallele con una sintassi semplice.

4.4.3.1 Java Executor

Java ha supportato la creazione di thread mediante l'approccio che abbiamo descritto finora sin dalle sue origini. Tuttavia, a partire dalla versione 1.5 e dalla relativa api, Java ha introdotto diverse nuove funzionalità per la concorrenza che forniscono agli sviluppatori un controllo superiore sulla creazione di thread e sulla loro comunicazione. Questi strumenti sono disponibili nel package `java.util.concurrent`. Piuttosto che creare esplicitamente oggetti `Thread`, la creazione del thread è ora organizzata attorno all'interfaccia `Executor`:

```
public interface Executor

{

    void execute(Runnable command);

}
```

Le classi che implementano questa interfaccia devono definire il metodo `execute()`, passato a un oggetto `Runnable`. Gli sviluppatori Java, invece di creare un oggetto `Thread` distinto e richiamare il suo metodo `start()`, possono utilizzare `Executor`, come segue:

```
Executor service = new Executor;
```

```
service.execute(new Task());
```

L'ambiente `Executor` si basa sul modello produttore-consamatore; vengono generati task che implementano l'interfaccia `Runnable` e i thread che eseguono questi task li consumano. Il vantaggio di questo approccio è che non solo separa la creazione del thread dalla sua esecuzione, ma fornisce anche un meccanismo per la comunicazione tra task concorrenti.

La condivisione dei dati tra thread appartenenti allo stesso processo avviene facilmente in Windows e Pthreads, poiché i dati condivisi vengono semplicemente dichiarati globalmente. Essendo un linguaggio puramente orientato agli oggetti, Java non possiede una tale nozione di dato globale e, sebbene sia possibile passare parametri a una classe che implementa `Runnable`, i thread Java non possono restituire risultati. Per soddisfare questa esigenza, il package `java.util.concurrent` definisce l'interfaccia `Callable`, che si comporta in modo simile a `Runnable` tranne per il fatto che è possibile restituire un risultato. I risultati restituiti dai task `Callable` sono noti come oggetti `Future`. Un risultato può essere recuperato con il metodo `get()` definito nell'interfaccia `Future`. Il programma mostrato nella Figura 4.14 illustra il calcolo di una sommatoria utilizzando queste funzionalità Java.

```
import java.util.concurrent.*;

class Summation implements Callable<Integer>

{
    private int upper;

    public Summation(int upper) {
        this.upper = upper;
    }

    /* Il thread viene eseguito in questo metodo */

    public Integer call() {

        int sum = 0;

        for (int i = 1; i <= upper; i++)
            sum += i;

        return new Integer(sum);
    }
}

public class Driver

{
    public static void main(String[] args) {
        int upper = Integer.parseInt(args[0]);
        ExecutorService pool = Executors.newSingleThreadExecutor();

        Future<Integer> result = pool.submit(new Summation(upper));
    }
}
```

```

try {

    System.out.println("sum = " + result.get());

}

} catch (InterruptedException | ExecutionException ie) { }

}

```

Figura 4.14 Esempio di utilizzo dell'api Java Executor.

La classe `Summation` implementa l'interfaccia `Callable`, che specifica il metodo `call()`, il cui codice viene eseguito in un thread separato. Per eseguire questo codice, creiamo un oggetto `newSingleThreadExecutor` (fornito come metodo statico nella classe `Executors`) di tipo `ExecutorService` e lo passiamo a un task `Callable` usando il suo metodo `submit()`. (La differenza principale tra i metodi `execute()` e `submit()` è che il primo non restituisce alcun risultato, mentre il secondo restituisce un `Future`). Una volta che inviamo il task `Callable` al thread, restiamo in attesa del risultato chiamando il metodo `get()` dell'oggetto `Future` restituito.

A prima vista questo modello per la creazione di thread appare più complicato della semplice creazione di un thread e della sua unione (`join`) nel momento della sua terminazione. Tuttavia, questa piccola complicazione aggiuntiva porta dei benefici, per esempio, come abbiamo visto, l'utilizzo di `Callable` e `Future` consente ai thread di restituire risultati. Inoltre, questo approccio separa la creazione di un thread dai risultati che produce: invece di aspettare che un thread termini per poter recuperare i risultati, il genitore attende che i risultati diventino disponibili. Infine, come vedremo nel Paragrafo 4.5.1, questo ambiente di sviluppo può essere combinato con altre funzionalità per creare strumenti robusti per la gestione di un numero elevato di thread.

LA JVM E IL SISTEMA OPERATIVO OSPITE

La macchina virtuale del linguaggio Java (jvm) è solitamente implementata sulla base di un sistema operativo sottostante (Figura 18.10). Questo assetto permette alla jvm di nascondere i dettagli del sistema operativo e di offrire un ambiente astratto e coerente che consente ai programmi scritti in Java di essere eseguiti su qualsiasi piattaforma che disponga di una jvm. Le specifiche della jvm non prescrivono come i thread Java debbano corrispondere ai servizi del sistema operativo sottostante, lasciando i dettagli all'implementazione. Il sistema operativo Windows, per esempio, adotta il modello da uno a uno; perciò, ogni thread Java di una jvm installata su questo sistema corrisponde a un thread a livello kernel. Oltre a ciò, può esistere una relazione tra la libreria dei thread di Java e la libreria dei thread del sistema operativo residente. Le varie versioni della jvm per la famiglia di sistemi operativi Windows, per esempio, potrebbero ricorrere alla api Windows al fine di implementare i thread di Java; i sistemi Linux e macos potrebbero impiegare la api Pthreads.

4.5 Threading implicito

Con la continua crescita di elaborazione multicore si profilano all'orizzonte applicazioni contenenti centinaia, o anche migliaia, di thread. La progettazione di tali applicazioni non è un'impresa semplice: i programmatori devono affrontare non solo le sfide descritte nel Paragrafo 4.2, ma anche ulteriori difficoltà. Queste difficoltà, che riguardano la correttezza dei programmi, sono trattate nei Capitoli 6 e 8.

Un modo per affrontare tali ostacoli e gestire al meglio la progettazione di applicazioni parallele e concorrenti è il trasferimento della creazione e della gestione del threading dagli sviluppatori di applicazioni ai compilatori e alle librerie di runtime.

Questa strategia, chiamata *threading implicito*, è diventata oggi molto comune. In questo paragrafo esploriamo tre approcci alternativi per la progettazione di programmi multithread in grado di sfruttare i processori multicore attraverso il threading implicito. Come vedremo, queste strategie richiedono in genere agli sviluppatori di applicazioni di identificare *task*, e non thread, che possano essere eseguiti in parallelo. Un task viene solitamente codificato come una funzione, che viene mappata dalla libreria di runtime su un thread separato, in genere utilizzando il modello da molti a molti (Paragrafo 4.3.3). Il vantaggio di questo approccio è che gli sviluppatori devono solo individuare i task parallelizzabili, mentre le librerie determinano i dettagli specifici della creazione e della gestione dei thread.

4.5.1 Gruppi di thread

Nel Paragrafo 4.1 abbiamo descritto un server web multithread in cui, per ogni richiesta ricevuta, il server crea un thread distinto per fornire il servizio richiesto. Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, tuttavia un server multithread presenta diversi problemi. Il primo riguarda il tempo richiesto per la creazione del thread prima di poter soddisfare la richiesta, considerando anche il fatto che questo thread sarà terminato non appena avrà completato il proprio lavoro. La seconda questione è più problematica: se si permette che tutte le richieste concorrenti siano servite da un nuovo thread, non si è posto un limite al numero di thread concorrentemente attivo nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di cpu o la memoria. L'impiego dei gruppi di thread (*thread pool*) è una possibile soluzione a questo problema.

L'idea generale è quella di creare un certo numero di thread alla creazione del processo e organizzarli in un gruppo (*pool*) in cui attenda di eseguire il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread del gruppo – se ce n'è uno disponibile – e gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nel gruppo d'attesa. I gruppi di thread funzionano bene quando i task possono essere eseguiti in maniera asincrona. I vantaggi sono i seguenti:

1. il servizio di una richiesta tramite un thread esistente è più rapido dell'attesa della creazione di un nuovo thread;
2. un gruppo di thread limita il numero di thread esistenti a un certo istante; ciò è particolarmente rilevante per sistemi che non possono sostenere un elevato numero di thread concorrenti;
3. separare il task da svolgere dalla meccanica della sua creazione ci permette di utilizzare diverse strategie per l'esecuzione di tale task. Per esempio, si potrebbe pianificare l'esecuzione del task dopo un ritardo di tempo o periodicamente.

Il numero di thread di un gruppo si può determinare tramite euristiche che considerano fattori come il numero di cpu nel sistema, la quantità di memoria fisica e il numero atteso di richieste concorrenti da parte dei client. Architetture più raffinate per la gestione dei gruppi di thread possono correggere dinamicamente il numero di thread di un gruppo secondo l'utilizzazione del sistema. Queste architetture hanno l'ulteriore vantaggio di utilizzare gruppi più piccoli – comportando quindi un minore impegno di memoria – quando il carico del sistema è basso. Introdurremo una di queste architetture, Grand Central Dispatch di Apple, nei prossimi paragrafi.

La api Windows mette a disposizione diverse funzioni legate ai gruppi di thread, il cui uso è simile alla creazione di un thread tramite la funzione `Thread_Create()` presentata al Paragrafo 4.4.2. Si definisce una funzione da eseguire in un nuovo thread, per esempio:

```
DWORD WINAPI PoolFunction(PVOID Param) {  
  
    /*Questa funzione esegue come thread separato.*/  
  
}
```

Un puntatore a `PoolFunction()` è poi passato a una delle apposite funzioni nella api per i gruppi di thread, il che avvierà uno dei thread del gruppo. Una di tali apposite funzioni è `QueueUserWorkItem()`, che accetta tre parametri:

- `LPTHREAD_START_ROUTINE Function` – un puntatore alla funzione da eseguire in un nuovo thread;
- `PVOID Param` – il parametro passato a `Function`;
- `ULONG Flags` – indica come il gruppo di thread debba creare e gestire l'esecuzione del nuovo thread.

Un esempio di chiamata è:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

A seguito di questa invocazione, uno dei thread del gruppo richiamerà `PoolFunction()` per conto del programmatore. In questo esempio, `PoolFunction()` non riceve parametri, e il valore 0 di `Flags` indica l'assenza di indicazioni particolari per la creazione del thread.

Altre funzioni della api Windows per i gruppi di thread offrono servizi di invocazione periodica di funzioni, o invocazioni legate al completamento di un'operazione di i/o asincrona.

GRUPPI DI THREAD (THREAD POOL) IN ANDROID

Nel Paragrafo 3.8.2.1 abbiamo parlato delle RPC nel sistema operativo Android. Ricordiamo da quel paragrafo che Android utilizza l'interfaccia AIDL (Android Interface Definition Language), uno strumento per specificare l'interfaccia remota utilizzata dai client per interagire con il server. AIDL fornisce anche un *thread pool*. Un servizio remoto che utilizza il thread pool può gestire più richieste simultanee, servendo ogni richiesta con un thread distinto del gruppo.

4.5.1.1 Thread pool in Java

Il pacchetto `java.util.concurrent` include un'api per diverse varietà architetturali di gruppi di thread. Ci concentreremo sui seguenti tre modelli:

1. Single thread executor—`newSingleThreadExecutor()`: crea un gruppo di dimensione 1.
2. Fixed thread executor—`newFixedThreadPool(int size)`: crea un gruppo con un numero specificato di thread.
3. Cached thread executor—`newCachedThreadPool()`: crea un gruppo di thread illimitato, riutilizzando i thread in esecuzioni successive.

Abbiamo in effetti già visto l'uso di un gruppo di thread Java nel Paragrafo 4.4.3, dove, nel programma d'esempio della Figura 4.14, abbiamo creato un `newSingleThreadExecutor`. In quel paragrafo abbiamo affermato che l'ambiente executor di Java può essere utilizzato per costruire strumenti di threading più robusti. Descriviamo ora come possa essere utilizzato per creare gruppi di thread.

Un gruppo di thread viene creato utilizzando uno dei metodi factory della classe `Executors`:

- `static ExecutorService newSingleThreadExecutor()`
- `static ExecutorService newFixedThreadPool(int size)`
- `static ExecutorService newCachedThreadPool()`

Ciascuno di questi metodi crea e restituisce un'istanza dell'oggetto che implementa l'interfaccia `ExecutorService`. `ExecutorService` estende l'interfaccia di `Executor`, permettendoci di invocare il metodo `execute()` su questo oggetto. Inoltre, `ExecutorService` fornisce metodi per la gestione della terminazione del gruppo di thread.

L'esempio mostrato nella Figura 4.15 crea un gruppo di cached thread e sottomette i task che devono essere eseguiti da un thread del gruppo utilizzando il metodo `execute()`. Quando viene invocato il metodo `shutdown()`, il gruppo di thread rifiuta task aggiuntivi e termina una volta che i task esistenti hanno completato l'esecuzione.

```
import java.util.concurrent.*;

public class ThreadPoolExample {

    public static void main(String[] args) {
        int numTasks = Integer.parseInt(args[0].trim());
        /* Crea il gruppo di thread */

        ExecutorService pool = Executors.newCachedThreadPool();

        /* Esegue ogni attività con un thread distinto del gruppo */
    }
}
```

```

        for (int i = 0; i < numTasks; i++)

        pool.execute(new Task());



        /* Termina il gruppo quando tutti i thread hanno completato l'attività */

        pool.shutdown();

    }

}

```

Figura 4.15 Creazione di un gruppo di thread in Java.

4.5.2 Fork join

La strategia di creazione dei thread trattata nel Paragrafo 4.4 è spesso nota come modello fork-join. Ricordiamo che secondo questo metodo il thread padre crea (*fork*) uno o più thread figli, attende che i figli terminino e si uniscano a esso (*join*) e a quel punto può recuperare e combinare i risultati ottenuti dai figli. Questo modello sincrono è spesso indicato come uno strumento per la creazione esplicita di thread, ma è anche un eccellente candidato per il threading implicito. In quest'ultima situazione, illustrata nella Figura 4.16, i thread non sono costruiti direttamente durante la fase di fork; piuttosto, occorre definire task paralleli. Una libreria gestisce il numero di thread che vengono creati ed è responsabile dell'assegnazione dei task ai thread. In un certo senso, questo modello fork-join è una versione sincrona di un thread pool in cui una libreria determina il numero effettivo di thread da creare, per esempio, utilizzando l'euristica descritta nel Paragrafo 4.5.1.

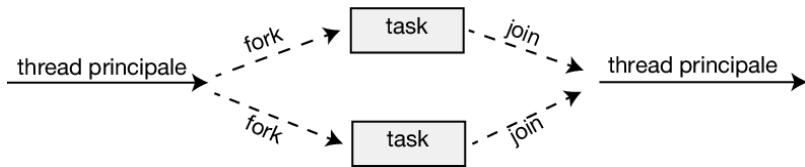


Figura 4.16 Parallelismo fork-join.

4.5.2.1 Fork join in Java

Java ha introdotto una libreria fork-join nella versione 1.7 dell'api, progettata per essere utilizzata con algoritmi ricorsivi *divide et impera* come Quicksort e Mergesort. Quando si implementano algoritmi *divide et impera* mediante questa libreria, differenti task vengono diramati (*fork*) durante il passo ricorsivo e vengono assegnati a sottoinsiemi più piccoli del problema originale. Gli algoritmi devono essere progettati in modo che questi task separati possano essere eseguiti contemporaneamente. A un certo punto, la dimensione del problema assegnato a un task è abbastanza piccola da poter essere risolta direttamente senza richiedere la creazione di nuovi task. Di seguito viene mostrato l'algoritmo ricorsivo generale che sta dietro il modello fork-join di Java:

```

Task(problem)

if problem is small enough

    solve the problem directly

else

    subtask1 = fork(new Task(subset of problem)

```

```

subtask2 = fork(new Task(subset of problem)

result1 = join(subtask1)

result2 = join(subtask2)

return combined results

```

La Figura 4.17 illustra graficamente il modello.

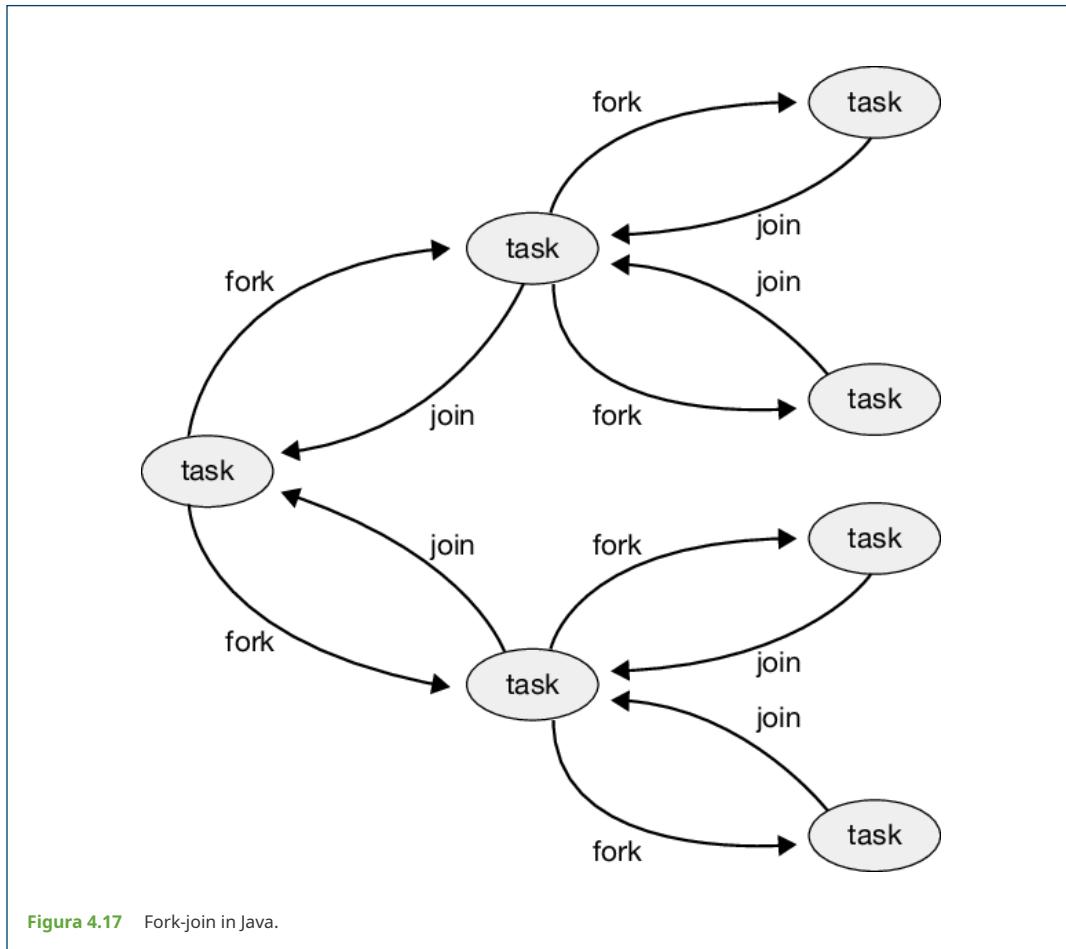


Figura 4.17 Fork-join in Java.

Illustriamo ora la strategia fork-join di Java progettando un algoritmo *divide et impera* che sommi tutti gli elementi di un vettore di numeri interi. Nella versione 1.7 dell'api Java è stato introdotto un nuovo gruppo di thread, chiamato `ForkJoinPool`, a cui possono essere assegnati task che ereditano la classe astratta `ForkJoinTask` (che per ora assumeremo essere la classe `SumTask`). Il codice che segue crea un oggetto `ForkJoinPool` e lancia il task iniziale tramite il suo metodo `invoke()`:

```

ForkJoinPool pool = new ForkJoinPool();

// array contiene i numeri interi da sommare

int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);

```

```
int sum = pool.invoke(task);
```

Al termine, la chiamata iniziale di `invoke()` restituisce la somma dell'array.

La classe `SumTask`, mostrata nella Figura 4.18, implementa un algoritmo *divide et impera* che somma il contenuto dell'array usando fork-join. I nuovi task vengono creati utilizzando il metodo `fork()`; il metodo `compute()` specifica il calcolo eseguito da ciascun task. Il metodo `compute()` viene invocato finché non sia in grado di calcolare direttamente la somma del sottoinsieme che gli è stato assegnato. La chiamata a `join()` si blocca fino a quando il task viene completato, dopo di che restituisce i risultati calcolati durante la `compute()`.

```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>

{

    static final int THRESHOLD = 1000;

    private int begin;

    private int end;

    private int[] array;

    public SumTask(int begin, int end, int[] array) {

        this.begin = begin;

        this.end = end;

        this.array = array;

    }

    protected Integer compute() {

        if (end - begin < THRESHOLD) {

            int sum = 0;

            for (int i = begin; i <= end; i++)

                sum += array[i];

            return sum;

        }

        else {

            int mid = (begin + end) / 2;

            SumTask leftTask = new SumTask(begin, mid, array);

            SumTask rightTask = new SumTask(mid + 1, end, array);

            leftTask.fork();

            rightTask.fork();

        }

    }

}
```

```

        return rightTask.join() + leftTask.join();

    }

}

}

```

Figura 4.18 Esempio di computazione fork-join in Java.

Si noti che `SumTask`, mostrata nella Figura 4.18, estende `RecursiveTask`. La strategia fork-join di Java è organizzata attorno alla classe astratta di base `ForkJoinTask` e le classi `RecursiveTask` e `RecursiveAction` estendono questa classe. La differenza fondamentale tra queste ultime due classi è che `RecursiveTask` restituisce un risultato (tramite il valore di ritorno specificato in `compute()`) e `RecursiveAction` no. La relazione tra le tre classi menzionate è illustrata nel diagramma uml della Figura 4.19.

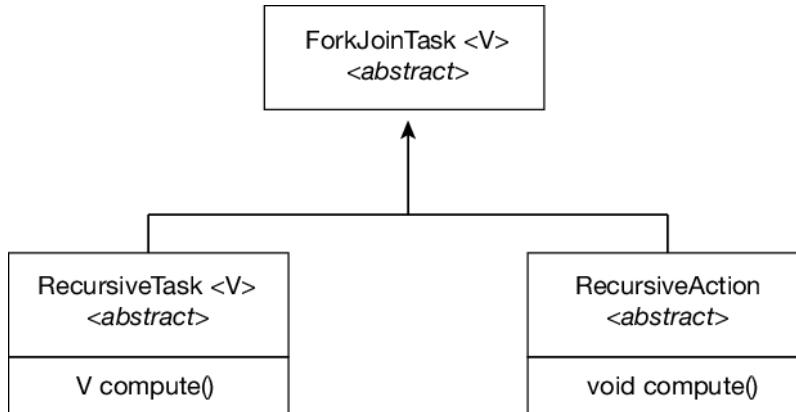


Figura 4.19 Diagramma UML della classe java per il fork-join.

Un aspetto importante da tenere in considerazione è determinare quando il problema è “abbastanza piccolo” da poter essere risolto direttamente, senza necessità di creare ulteriori task. In `SumTask` ciò si verifica quando il numero di elementi da sommare è inferiore al valore `THRESHOLD`, che nella Figura 4.18 abbiamo impostato arbitrariamente a 1.000. Nella pratica, per determinare quando un problema può essere risolto direttamente occorre eseguire attenti cronometraggi, poiché il valore può variare in base all'implementazione.

Ciò che è interessante nel modello fork-join di Java è la gestione dei task, dove la libreria costruisce un gruppo di thread di lavoro e bilancia il carico dei task tra i thread disponibili. In alcune situazioni ci sono migliaia di task, ma solo una manciata di thread esegue il lavoro (per esempio, un thread distinto per ogni cpu). Inoltre, ogni thread in `ForkJoinPool` mantiene una coda dei task che ha generato; se la coda di un thread è vuota, il thread può “rubare” un’attività dalla coda di un altro thread usando un algoritmo detto di work stealing e bilanciando così il carico di lavoro tra tutti i thread.

4.5.3 OpenMP

Openmp è un insieme di direttive del compilatore e una api per programmi scritti in C, C++ o fortran che fornisce il supporto per la programmazione parallela in ambienti a memoria condivisa. Openmp definisce le regioni parallele come blocchi di codice eseguibili in parallelo. Gli sviluppatori di applicazioni inseriscono direttive del compilatore nei punti del codice dove vi sono regioni parallele e queste direttive istruiscono la libreria di runtime Openmp per l'esecuzione della regione in parallelo. Il seguente programma C illustra l'uso di una direttiva del compilatore inserita prima della regione parallela contenente l'istruzione `printf()`:

```

#include <omp.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    /* codice sequenziale */

#pragma omp parallel

    {
        printf("I am a parallel region.");
    }

    /* codice sequenziale */

    return 0;
}

```

Quando Openmp incontra la direttiva

```
# pragma omp parallel
```

crea tanti thread quanti sono i core di elaborazione del sistema. In questo modo, in un sistema dual-core vengono creati due thread, in un sistema quad-core quattro thread, e così via. Tutti i thread eseguono poi contemporaneamente la regione parallela. Quando un thread esce dalla regione parallela viene terminato.

Openmp fornisce diverse direttive aggiuntive per l'esecuzione di porzioni di codice in parallelo, tra cui i cicli parallelizzati. Per esempio, supponiamo di avere due array `a` e `b` di dimensione `N`. Desideriamo sommare i valori in essi contenuti e inserire i risultati nell'array `c`. Possiamo eseguire questo task in parallelo utilizzando il seguente frammento di codice, contenente la direttiva del compilatore per parallelizzare i cicli `for`:

```

#pragma omp parallel for

for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}

```

Openmp divide il compito contenuto nel ciclo `for` fra i thread che ha creato in risposta alla direttiva:

```
# pragma omp parallel for
```

Oltre a fornire direttive per la parallelizzazione, Openmp consente agli sviluppatori di scegliere tra diversi livelli di parallelismo. Si può per esempio impostare manualmente il numero di thread o indicare se i dati sono condivisi tra i thread o sono riservati a un solo thread. Openmp è disponibile su diversi compilatori open-source e commerciali per sistemi Linux, Windows e macos. Esortiamo i lettori interessati a saperne di più su Openmp a consultare la bibliografia alla fine del capitolo.

4.5.4 Grand Central Dispatch

Grand Central Dispatch (gcd), una tecnologia per i sistemi operativi macos e ios di Apple, è una combinazione di estensioni del linguaggio C, una api e una libreria di runtime che permette agli sviluppatori di applicazioni di individuare sezioni di codice da eseguire in parallelo. Come Openmp, gcd gestisce la maggior parte dei dettagli del threading.

gcd pianifica l'esecuzione runtime dei task inserendoli in una coda di dispacciamento (*dispatch queue*). Quando gcd rimuove un task dalla coda, lo assegna a un thread disponibile tra quelli presenti nel gruppo di thread che gestisce. gcd definisce due tipi di code di dispacciamento: *serial* e *concurrent*.

I task posti in una coda seriale vengono prelevati secondo un ordine fifo. Una volta che un task è stato rimosso dalla coda la sua esecuzione deve essere completata prima del prelievo di un altro task. Ogni processo ha una propria coda seriale, chiamata coda principale (*main queue*). Gli sviluppatori possono creare code seriali locali a processi particolari (questo è il motivo per cui le code seriali sono anche note come private dispatch queue, o code di dispacciamento private). Le code seriali sono utili per assicurare l'esecuzione sequenziale delle diverse attività.

Anche i task posti in una coda concorrente vengono rimossi secondo un ordine fifo, ma è possibile prelevare più task alla volta, permettendo così la loro esecuzione in parallelo. Ci sono diverse code concorrenti a livello di sistema (note anche come global dispatch queues, o *code di dispacciamento globali*), suddivise in quattro classi principali in base alla qualità del servizio (qos):

- **QOS_CLASS_USER_INTERACTIVE**: la classe user-interactive rappresenta i task che interagiscono con l'utente, come l'interfaccia utente e la gestione degli eventi, per garantire un'interfaccia utente reattiva. Il completamento di un task appartenente a questa classe richiede di norma poco lavoro.
- **QOS_CLASS_USER_INITIATED**: la classe user-initiated è simile alla classe precedente, perché le attività sono associate a un'interfaccia utente reattiva; tuttavia, i task avviati dall'utente possono richiedere tempi di elaborazione più lunghi. Per esempio, l'apertura di un file o di un url è un task avviato dall'utente. I task di questa classe devono essere completati affinché l'utente possa continuare a interagire con il sistema, ma non è necessario servirli con la stessa tempestività dei task nella coda user-interactive.
- **QOS_CLASSUTILITY**: La classe utility rappresenta i task che richiedono un tempo più lungo per essere completati, ma non richiedono risultati immediati. Questa classe include attività come l'importazione di dati.
- **QOS_CLASS_BACKGROUND**: i task appartenenti alla classe background non sono visibili all'utente e non sono sensibili al fattore tempo. Alcuni esempi sono l'indicizzazione di una casella di posta elettronica e l'esecuzione di un backup.

I task inviati alle code di dispacciamento possono essere espressi in due modi diversi.

1. Nei linguaggi C, C++ e Objective-C, gcd definisce un'estensione del linguaggio nota come blocco. Un blocco è semplicemente un'unità di lavoro autocontenuta e viene specificato dal simbolo ^ inserito prima di una coppia di parentesi graffe {}. Il codice tra parentesi descrive l'attività da svolgere. Un semplice esempio di blocco è:

```
^{ printf ( "I am a block" ); }
```

1. Nel linguaggio di programmazione Swift, un task viene definito utilizzando una chiusura (*closure*), simile a un blocco, in quanto esprime un'unità autocontenuta di funzionalità. Sintatticamente, una chiusura Swift è scritta come un blocco, eccetto il simbolo ^ iniziale.

Il seguente segmento di codice Swift illustra come ottenere una coda concorrente per la classe `user-initiated` e inviare un task alla coda utilizzando la funzione `dispatch_async()`:

```
let queue = dispatch_get_global_queue (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue, { print("I am a closure." ) })
```

Internamente, il gruppo di thread di gcd è composto da thread posix. gcd gestisce attivamente il gruppo, consentendo al numero di thread di aumentare o diminuire in base alle richieste dell'applicazione e alla capacità del sistema. gcd è implementato dalla libreria `libdispatch`, che Apple ha rilasciato sotto licenza Apache Commons. In seguito ne è stato effettuato il porting su sistema operativo Freebsd.

4.5.5 Intel Thread Building Blocks

Intel threading building blocks (tbb) è una libreria di template che supporta la progettazione di applicazioni parallele in C++ e non richiede alcun compilatore o supporto linguistico speciale. Gli sviluppatori specificano i task che possono essere eseguiti in parallelo e il task scheduler di tbb mappa questi task sui thread sottostanti. Inoltre, il task scheduler fornisce il bilanciamento del carico ed è consci della presenza della cache, ovvero darà la precedenza alle attività che più probabilmente avranno i loro dati memorizzati nella memoria cache e che quindi verranno eseguite più rapidamente. tbb offre un ricco set di funzionalità, tra cui template per cicli paralleli, operazioni atomiche e lock mutex. Inoltre, fornisce strutture dati concorrenti, tra cui la mappa hash, la coda e il vettore, che possono fungere da versioni thread-safe equivalenti alle strutture dati della libreria di template standard del linguaggio C++.

Consideriamo, per esempio, il parallelismo nei cicli. Si supponga che esista una funzione denominata `apply(float value)` che esegue una certa operazione sul parametro `value`. Se abbiamo un array `v` di dimensione `n` contenente valori `float`, possiamo usare il seguente ciclo `for` (seriale) per passare ogni valore in `v` alla funzione `apply()`:

```
for (int i = 0; i < n; i++) {  
  
    apply(v[i]);  
  
}
```

Uno sviluppatore potrebbe applicare manualmente il parallelismo dei dati (Paragrafo 4.2.2) su un sistema multicore, assegnando una regione diversa dell'array *v* a ciascun core di elaborazione; tuttavia, ciò lega strettamente la tecnica per ottenere il parallelismo all'hardware fisico e l'algoritmo deve essere modificato e ricompilato al variare del numero di core di elaborazione su ciascuna architettura specifica.

In alternativa, uno sviluppatore può utilizzare tbb, che fornisce un template `parallel_for` a due valori,

```
parallel_for (range body)
```

dove *range* indica l'intervallo di elementi su cui avverrà l'iterazione (noto come spazio di iterazione) e *body* specifica l'operazione che verrà eseguita su un sottoinsieme di elementi.

Possiamo riscrivere il ciclo seriale visto sopra usando il template `tbb parallel_for`, come segue:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

I primi due parametri specificano che lo spazio di iterazione va da 0 a *n*-1 (e quindi ha la stessa dimensione dell'array *v*). Il secondo parametro è una funzione lambda del C++ e richiede qualche spiegazione in più. L'espressione `[=](size_t i)` indica il parametro *i*, che assume ciascuno dei valori nello spazio di iterazione (in questo caso da 0 a *n*-1). Ogni valore di *i* viene utilizzato per identificare quale elemento dell'array *v* deve essere passato come parametro alla funzione `apply(v[i])`.

La libreria tbb divide le iterazioni del ciclo in "blocchi" separati e crea un certo numero di task che operano su quei blocchi. (La funzione `parallel_for` consente agli sviluppatori di specificare manualmente la dimensione dei blocchi, se lo desiderano). tbb creerà anche un certo numero di thread e assegnerà i task ai thread disponibili. Questo procedimento è abbastanza simile a quanto avviene con la libreria fork-join in Java. Il vantaggio di questo approccio è che richiede soltanto che gli sviluppatori identifichino quali operazioni siano eseguibili in parallelo (specificando un ciclo `parallel_for`), mentre la libreria gestisce i dettagli impliciti dalla suddivisione del lavoro in task separati che vengono eseguiti in parallelo. Intel tbb ha versioni commerciali e open-source che funzionano su Windows, Linux e macos. Fate riferimento alla bibliografia per ulteriori dettagli su come sviluppare applicazioni parallele utilizzando tbb.

4.6 Problematiche di programmazione multithread

In questo paragrafo si affrontano alcune problematiche legate al progetto di programmi multithread.

4.6.1 Chiamate di sistema fork() ed exec()

Nel Capitolo 3 è stato descritto l'uso della chiamata di sistema `fork()` per la creazione di un nuovo processo tramite la duplicazione di un processo esistente. In un programma multithread la semantica delle chiamate di sistema `fork()` ed `exec()` cambia.

Se un thread in un programma invoca la chiamata di sistema `fork()`, il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante. Alcuni sistemi unix includono entrambe le versioni.

La chiamata di sistema `exec()` di solito funziona nello stesso modo descritto nel Capitolo 3: se un thread invoca la chiamata di sistema `exec()`, il programma specificato come parametro della `exec()` sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della `fork()` dipende dall'applicazione. Se s'invoca la `exec()` immediatamente dopo la `fork()`, la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della `exec()` sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la `exec()` non segue immediatamente la `fork()`, il nuovo processo dovrebbe duplicare tutti i thread del processo genitore.

4.6.2 Gestione dei segnali

Nei sistemi unix si usano i segnali per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

1. all'occorrenza di un particolare evento si genera un segnale;
2. s'invia il segnale a un processo;
3. una volta ricevuto, il segnale deve essere gestito.

Come esempio, un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui sono considerati sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti (come `<control><c>`) oppure la scadenza di un timer. Di solito un segnale asincrono s'invia a un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un gestore predefinito di segnali;
2. tramite un gestore di segnali definito dall'utente.

Per ogni segnale esiste un gestore predefinito del segnale che il kernel esegue quando deve gestire il segnale. La gestione predefinita è sostituibile da un gestore del segnale definito dall'utente, richiamato per gestire il segnale. Sia i segnali sincroni sia quelli asincroni sono gestibili in modi diversi: alcuni si possono semplicemente ignorare (per esempio, il ridimensionamento di una finestra); altri si possono gestire terminando l'esecuzione del programma (per esempio, un accesso illegale alla memoria).

Per i processi a singolo thread la gestione dei segnali è semplice: i segnali vengono sempre inviati al processo. Per i processi multithread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce;
2. inviare il segnale a ogni thread del processo;
3. inviare il segnale a specifici thread del processo;
4. definire un thread specifico per ricevere tutti i segnali diretti al processo.

Il metodo per recapitare un segnale dipende dal tipo di segnale. I segnali sincroni, per esempio, si devono inviare al thread che ha generato l'evento causa del segnale e non ad altri thread nel processo. Se si tratta di segnali asincroni la situazione non è invece così chiara; alcuni segnali asincroni, come il segnale che termina un processo (come `<control><c>`), si devono inviare a tutti i thread.

La funzione standard per l'invio di un segnale in unix è `kill(pid_t pid, int signal)`.

Questa funzione specifica il processo (`pid`) al quale un particolare segnale (`signal`) deve essere recapitato. La maggior parte delle versioni multithread di unix permette che per ciascun thread si indichino i segnali da accettare e quelli da bloccare. Quindi, alcuni segnali asincroni si potrebbero recapitare soltanto ai thread che non li bloccano.

Tuttavia, poiché i segnali vanno gestiti una sola volta, di solito un segnale è recapitato solo al primo thread che non lo blocca. La api Pthreads posix dispone della funzione

```
pthread_kill(pthread_t tid, int signal)
```

che permette di specificare il thread (`tid`) cui recapitare il segnale.

Sebbene Windows non preveda la gestione esplicita dei segnali, questi si possono emulare con le chiamate di procedure asincrone (*asynchronous procedure call*, apc). Le funzioni apc permettono a un thread a livello utente di specificare la funzione da richiamare quando il thread riceve la comunicazione di un particolare evento. Come s'intuisce dal nome, una apc è grosso modo equivalente a un segnale asincrono di unix. Mentre tuttavia in un ambiente multithread unix necessita di un criterio di gestione dei segnali, il sistema delle apc è più semplice, poiché una apc è rivolta a un particolare thread e non a un processo.

4.6.3 Cancellazione dei thread

La cancellazione dei thread è l'operazione che permette di terminare un thread prima che completi il suo compito. Per esempio, se più thread eseguono una ricerca in modo concorrente in una base di dati e un thread riporta il risultato, gli altri thread possono essere cancellati. Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di terminazione di un browser web per interrompere il caricamento di una pagina. Spesso il caricamento di una pagina è gestito da più thread: ogni immagine è caricata da un thread separato; quando l'utente preme il pulsante di terminazione, tutti i thread che stanno caricando la pagina vengono cancellati.

Un thread da cancellare è spesso chiamato thread bersaglio (*target thread*). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

1. cancellazione asincrona. Un thread fa immediatamente terminare il thread bersaglio;
2. cancellazione differita. Il thread bersaglio controlla periodicamente se deve terminare, in modo da effettuare la terminazione in maniera ordinata.

Si presentano difficoltà con la cancellazione nei casi in cui ci siano risorse assegnate a un thread cancellato, o se si cancella un thread mentre sta aggiornando dei dati che condivide con altri thread. Quest'ultimo caso è particolarmente problematico se si tratta di cancellazione asincrona. Il sistema operativo di solito si riappropria delle risorse di sistema usate da un thread cancellato, ma spesso non si riappropria di tutte le risorse. Quindi, la cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La cancellazione differita invece funziona tramite un thread che segnala la necessità di cancellare un certo thread bersaglio; la cancellazione avviene soltanto quando il thread bersaglio verifica se debba essere o meno cancellato. Questo metodo permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi.

In Pthreads la cancellazione del thread viene avviata tramite la funzione `pthread_cancel()`. L'identificatore del thread da cancellare viene passato come parametro alla funzione. Il codice seguente illustra la creazione e la cancellazione di un thread:

```

pthread_t tid;

/* Crea il thread */

pthread_create(&tid, 0, worker, NULL);

. . .

/* Cancella il thread */

pthread_cancel(tid);

/* Attende la terminazione del thread */

pthread_join(tid,NULL);

```

La chiamata della `pthread_cancel()` comporta solamente una richiesta di cancellazione del thread di destinazione: l'effettiva cancellazione dipende da come il thread di destinazione è impostato per la gestione della richiesta. Pthreads supporta tre modalità di cancellazione, ognuna definita da uno stato e da un tipo, come illustrato nella tabella che segue. Un thread può impostare il suo stato di cancellazione e il suo tipo usando una api.

Modalità	Stato	Tipo
Off	Disabilitato	-
Differita	Abilitato	Differito
Asincrona	Abilitato	Asincrono

Come risulta dalla tabella, Pthreads permette di disabilitare o abilitare la cancellazione dei thread. Ovviamente, un thread non può essere cancellato se la cancellazione è disabilitata; tuttavia, le richieste di cancellazione rimangono in sospeso, in modo che il thread possa in seguito abilitare la cancellazione e rispondere alla richiesta.

Il tipo di cancellazione predefinito è la cancellazione differita, secondo cui la cancellazione avviene solo quando un thread raggiunge un punto di cancellazione (*cancellation point*). La maggior parte delle chiamate di sistema bloccanti in posix e nella libreria standard

del C è definita come punto di cancellazione (è possibile elencare queste chiamate utilizzando il comando `man pthreads` su un sistema Linux). Per esempio, la chiamata di sistema `read()` è un punto di cancellazione che consente di cancellare un thread bloccato durante l'attesa di input da `read()`.

Una tecnica per la creazione di un punto di cancellazione consiste nell'invocare la funzione `pthread_testcancel()`. In caso di richiesta di cancellazione in attesa, viene richiamata una funzione nota come gestore della pulizia (*cleanup handler*), che permette di rilasciare tutte le risorse che un thread può aver acquisito prima di eliminarlo.

Il codice seguente illustra la risposta di un thread a una richiesta di annullamento mediante cancellazione differita:

```
while (1) {  
  
    /* fai qualche lavoro per un po' di tempo */  
  
    . . .  
  
    /* verifica se vi sia una richiesta di cancellazione */  
  
    pthread_testcancel();  
  
}
```

A causa dei problemi descritti in precedenza, nella documentazione di Pthreads viene sconsigliata la cancellazione asincrona e per questa ragione noi non la tratteremo. Una nota interessante: su sistemi Linux, la cancellazione dei thread con l'uso dell'api Pthreads è gestita attraverso segnali (Paragrafo 4.6.2).

La cancellazione del thread in Java utilizza una politica simile alla cancellazione differita in Pthreads. Per cancellare un thread Java occorre invocare il metodo `interrupt()`, che imposta lo stato di interruzione di un thread a true:

```
Thread worker;  
  
. . .  
  
/* imposta lo stato di interruzione del thread */  
  
worker.interrupt()
```

Un thread può controllare il suo stato di interruzione invocando `isInterrupted()`, che restituisce il valore booleano dello stato di interruzione di un thread:

```
while (!Thread.currentThread().isInterrupted()) {  
  
. . .  
}
```

4.6.4 Dati locali dei thread

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati dati specifici dei thread (o tls). Per esempio, in un sistema transazionale si può svolgere ciascuna transazione tramite un thread distinto e assegnare un identificatore unico per ogni transazione. Per associare ciascun thread al relativo identificatore si possono usare dati specifici dei thread.

E facile confondere i dati specifici dei thread con le variabili locali. Mentre le variabili locali sono visibili solo durante la chiamata di una singola funzione, i dati specifici sono visibili attraverso tutte le chiamate. Inoltre, quando lo sviluppatore non ha alcun controllo sul processo di creazione dei thread, per esempio quando si utilizza una tecnica implicita come un gruppo di thread, è necessario un approccio alternativo.

In un certo senso, i tls assomigliano ai dati statici, con la differenza che i dati tls sono unici per ogni thread. (In effetti, i tls vengono solitamente dichiarati come `static`). La maggior parte delle librerie di thread e dei compilatori fornisce supporto per tls. Per esempio, Java fornisce una classe `ThreadLocal <T>` con metodi `set()` e `get()` per gli oggetti `ThreadLocal <T>`. Pthreads include il tipo `pthread_key_t` che fornisce una chiave specifica per ogni thread. Questa chiave può essere utilizzata per accedere ai dati tls. Il linguaggio C# di Microsoft richiede semplicemente l'aggiunta dell'attributo `[ThreadStatic]` nella dichiarazione dei dati locali del thread. Il compilatore `gcc` fornisce la keyword `_thread`, che definisce la storage class necessaria per la dichiarazione dei dati locali del thread. Per esempio, se vogliamo assegnare un identificatore univoco a ogni thread, scriviamo la dichiarazione come segue:

```
static _thread int threadID;
```

4.6.5 Attivazione dello scheduler

Un'ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel modello a due livelli e in quello da molti a molti (Paragrafo 4.3.3). È proprio grazie a questa forma di coordinamento che il numero dei thread nel kernel è modificabile dinamicamente, con l'obiettivo di conseguire le migliori prestazioni.

Molti sistemi che implementano o il modello da molti a molti o quello a due livelli collocano una struttura dati intermedia tra i thread del kernel e dell'utente. Questa struttura dati, normalmente nota come processo leggero o lwp (acronimo di *lightweight process*) è mostrata nella Figura 4.. Dal punto di vista della libreria di thread a livello utente, l'lwp si presenta come un *processore virtuale* a cui l'applicazione può richiedere lo scheduling di un thread a livello utente. Ciascun lwp è associato a un thread del kernel, e sono proprio i thread del kernel che il sistema operativo pone in esecuzione sui processori fisici. Se un thread del kernel si blocca (mentre attende il completamento di un'operazione di i/o, per esempio) anche l'lwp si blocca. L'effetto a catena risale fino al thread a livello utente associato all'lwp, che si blocca anch'esso.

Per un'efficiente esecuzione un'applicazione può aver bisogno di un numero impreciso di lwp. Si consideri un processo con prevalenza di elaborazione eseguito da un singolo processore. In questa situazione è eseguibile solo un thread per volta, dunque un lwp è sufficiente. Un'applicazione con prevalenza di i/o potrebbe, tuttavia, richiedere l'esecuzione di molteplici lwp. Di solito è necessario un lwp per ogni chiamata di sistema concorrente bloccante. Supponiamo, per esempio, che giungano allo stesso tempo cinque richieste differenti per la lettura di file. Sono necessari cinque lwp, nel caso in cui tutte le richieste restino in attesa del completamento dell'i/o nel kernel. Se un processo ha soltanto quattro lwp, la quinta richiesta deve attendere che uno degli lwp sia rilasciato dal kernel.

Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come attivazione dello scheduler. Il suo funzionamento è il seguente: il kernel fornisce all'applicazione una serie di processori virtuali (lwp), mentre l'applicazione esegue lo scheduling dei thread dell'utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l'applicazione se si verificano determinati eventi, seguendo una procedura nota come upcall. Le upcall sono gestite dalla libreria dei thread mediante un apposito gestore, eseguito su un processore virtuale. Una situazione capace di innescare una upcall si verifica quando il thread di un'applicazione è sul punto di bloccarsi. In questo caso il kernel, tramite una upcall, informa l'applicazione che un thread è prossimo a bloccarsi, e identifica il thread in oggetto. Il kernel, quindi, assegna all'applicazione un nuovo processore virtuale. L'applicazione esegue un gestore della upcall su questo nuovo processore: il gestore salva lo stato del thread bloccante e rilascia il processore virtuale su cui era stato eseguito. Il gestore della upcall pianifica allora l'esecuzione di un altro thread sul nuovo processore virtuale. Quando si verifica l'evento atteso dal thread bloccante, il kernel fa un'altra upcall alla libreria dei thread per comunicare che il thread bloccato è nuovamente in condizione di essere eseguito. Il gestore di questa upcall necessita anch'esso di un processore virtuale: il kernel può crearne uno *ex novo*, o sottrarlo a un thread utente per prelazione. L'applicazione contrassegna il thread ad allora bloccato come pronto per l'esecuzione, ed esegue lo scheduling di un thread pronto per l'esecuzione su un processore virtuale disponibile.

4.7 Esempi di sistemi operativi

Abbiamo fin qui esaminato una serie di concetti e problematiche relativi ai thread. Concludiamo il capitolo descrivendo come i thread siano implementati nei sistemi Windows e Linux.

4.7.1 Thread di Windows

Un'applicazione per l'ambiente Windows si esegue come un processo separato; ogni processo può contenere uno o più thread. La api Windows per la creazione dei thread è trattata nel Paragrafo 4.4.2. Il sistema Windows impiega il modello da uno a uno, descritto nel Paragrafo 4.3.2, secondo cui ogni thread a livello utente si associa a un thread del kernel.

I componenti generali di un thread includono:

- un identificatore di thread (id), che identifica univocamente il thread;
- un insieme di registri che rappresenta lo stato del processore;
- un contatore di programma;
- uno stack utente, usato quando il thread è eseguito in modalità utente, e uno stack kernel, usato quando il thread è eseguito in modalità kernel;
- un'area di memoria privata, usata da diverse librerie run-time e dinamiche (dll).

L'insieme di registri, gli stack e la memoria privata sono detti contesto del thread. Le strutture dati principali di un thread includono:

- ethread (*executive thread block*);
- kthread (*kernel thread block*);
- teb (*thread environment block*).

I componenti chiave dell'ethread sono un puntatore al processo a cui il thread appartiene e l'indirizzo della funzione in cui il thread inizia l'esecuzione. La struttura ethread contiene anche un puntatore alla corrispondente struttura kthread. Quest'ultima include informazioni per il thread relative allo scheduling e alla sincronizzazione. Inoltre, kthread contiene lo stack kernel (usato quando il thread viene eseguito in modalità kernel) e un puntatore alla struttura teb.

Le strutture ethread e kthread risiedono interamente nello spazio del kernel; ciò implica che solo il kernel vi può accedere. La struttura dati teb appartiene invece allo spazio utente e vi si accede quando il thread è eseguito in modalità utente. Tra gli altri campi, il teb contiene l'identificatore del thread, uno stack per la modalità utente e un vettore di dati specifici del thread. La struttura di un thread di Windows è illustrata nella Figura 4.22.

flag	significato
CLONE_FS	Condivisione delle informazioni sul file system
CLONE_VM	Condivisione dello stesso spazio di memoria
CLONE_SIGHAND	Condivisione dei gestori dei segnali
CLONE_FILES	Condivisione dei file aperti

Figura 4.22 Alcuni dei flag passati quando viene invocata la funzione `clone()`.

4.7.2 Thread di Linux

Come si è visto nel Capitolo 3 Linux offre la chiamata di sistema `fork()` per duplicare un processo, e prevede inoltre la chiamata di sistema `clone()` per generare nuovo thread. Tuttavia Linux non distingue tra *processi* e *thread*, impiegando generalmente al loro posto il termine task in riferimento a un flusso del controllo nell'ambito di un programma.

Quando `clone()` è invocata, riceve come parametro un insieme di indicatori (*flag*), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio.

Alcuni di questi flag sono illustrati nella Figura 4.. Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il task genitore e il task figlio condivideranno le medesime informazioni sul file system (come la directory attiva), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Adoperare `clone()` in questo modo è equivalente a creare thread come descritto in questo capitolo, dal momento che il task genitore condivide la maggior parte delle proprie risorse con il task figlio. Tuttavia, se nessuno dei flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione, e la funzionalità ottenuta diventa simile a quella fornita dalla chiamata di sistema `fork()`.

Questa condivisione a intensità variabile è resa possibile dal modo in cui un task è rappresentato nel kernel di Linux. Per ogni task, nel kernel esiste un'unica struttura dati (e precisamente, `struct task_struct`). Questa struttura, invece di memorizzare i dati del task relativo, utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti: per esempio, strutture dati che rappresentano l'elenco dei file aperti, le informazioni per la gestione dei segnali e la memoria virtuale. Quando si invoca `fork()`, si crea un nuovo task insieme con una *copy* di tutte le strutture dati del task genitore. Anche quando s'invoca la chiamata `clone()` si crea un nuovo task, ma anziché ricevere una copia di tutte le strutture dati, il nuovo task può *puntare* alle strutture dati del task genitore, a seconda dell'insieme di flag passati a `clone()`.

Infine, la chiamata di sistema `clone()` può essere usata per implementare il concetto di contenitore (*container*); un contenitore è una tecnica di virtualizzazione fornita dal sistema operativo che consente di creare diversi sistemi Linux (contenitori), isolati l'uno dall'altro, su un singolo kernel Linux. Proprio come alcuni flag passati alla `clone()` permettono di generare un task che si comporta

come un processo o come un thread a seconda della quantità di condivisione tra i padre e figlio, ci sono altri flag della `clone()` che consentono la creazione di un contenitore Linux. I contenitori saranno trattati in modo più approfondito nel Capitolo 18.

4.8 Sommario

- Un thread rappresenta l'unità di base che utilizza la cpu. I thread appartenenti a uno stesso processo condividono molte risorse del processo, inclusi codice e dati.
- Le applicazioni multithread offrono quattro principali vantaggi: (1) reattività, (2) condivisione delle risorse, (3) economia e (4) scalabilità.
- La concorrenza esiste quando più thread stanno facendo progressi, mentre il parallelismo esiste quando più thread stanno facendo progressi simultaneamente. Su un sistema con una singola cpu è possibile solo la concorrenza; il parallelismo richiede un sistema multicore che fornisce più di una cpu.
- Ci sono diverse sfide nella progettazione di applicazioni multithread, tra cui la divisione e il bilanciamento del carico di lavoro, la suddivisione dei dati tra i diversi thread e l'identificazione delle dipendenze dei dati. I programmi multithread sono inoltre particolarmente difficili da testare ed è difficile eseguirne il debug.
- Il parallelismo dei dati distribuisce sottoinsiemi dei dati su più core di elaborazione ed esegue la stessa operazione su ogni core. Il parallelismo delle attività prevede la distribuzione di attività, e non di dati, su più core. Ogni attività esegue un'operazione distinta.
- Le applicazioni utente creano thread a livello utente che dovranno essere associati ai thread a livello kernel per essere eseguiti su una cpu. Il modello da molti a uno mappa molti thread a livello utente su un thread a livello kernel. Esistono altri approcci, tra cui i modelli da uno a uno e da molti a molti.
- Una libreria di thread fornisce un'api per la creazione e la gestione dei thread. Tra le librerie più note vi sono le librerie Windows, Pthreads e Java. La libreria Windows è utilizzata solo su sistemi Windows, mentre Pthreads è disponibile su sistemi compatibili con posix come unix, Linux e macos. I thread Java possono essere eseguiti su qualsiasi sistema che supporti una macchina virtuale Java.
- Il threading implicito richiede di identificare i task, e non i thread, e di permettere ai framework dei linguaggi o delle api di creare e gestire i thread. Esistono diversi approcci al threading implicito, inclusi i gruppi di thread, il fork-join e Grand Central Dispatch. Il threading implicito sta diventando una tecnica sempre più utilizzata dai programmati nello sviluppo di applicazioni concorrenti e parallele.
- I thread possono essere terminati utilizzando la cancellazione asincrona o la cancellazione differita. La cancellazione asincrona interrompe immediatamente un thread, anche se si trova a metà di un aggiornamento. La cancellazione differita informa un thread che dovrebbe terminare la sua esecuzione, ma gli consente di terminare in modo ordinato. Nella maggior parte dei casi la cancellazione differita è preferibile alla terminazione asincrona.
- A differenza di molti altri sistemi operativi, Linux non distingue tra processi e thread, ma si riferisce a ciascuno come un task. La chiamata di sistema `clone()` di Linux può essere utilizzata per creare task che si comportano in maniera più simile ai processi o più simile ai thread.

Esercizi di ripasso

- 4.1 Fornite tre esempi di programmi nei quali il multithread offra prestazioni migliori rispetto a soluzioni con un singolo thread.
- 4.2 Usando la legge di Amdahl, calcolate il guadagno in termini di velocità di un'applicazione che ha una componente parallela del 60 per cento in caso di utilizzo di (a) due core di elaborazione e (b) quattro core di elaborazione.
- 4.3 Il web server multithread descritto nel Paragrafo 4.1 presenta parallelismo delle attività o dei dati?
- 4.4 Quali sono due differenze tra i thread a livello utente e i thread a livello kernel? In quali circostanze un tipo è meglio dell'altro?
- 4.5 Descrivete le azioni intraprese da un kernel per cambiare contesto tra i thread a livello kernel.
- 4.6 Quali risorse vengono utilizzate quando si crea un thread? Come differiscono da quelle utilizzate quando si crea un processo?
- 4.7 Assumete che un sistema operativo mappi i thread a livello utente sul kernel utilizzando il modello molti a molti e che la mappatura avvenga tramite lwp. Assumete inoltre che il sistema permetta agli sviluppatori di creare dei thread real-time da utilizzare in sistemi real-time. È necessario vincolare un thread real-time a un lwp? Fornite una spiegazione.

Esercizi

4.8 Descrivete due esempi di programmazione in cui il multithreading *non* offre prestazioni migliori rispetto alla corrispondente soluzione a singolo thread.

4.9 In quali circostanze una soluzione basata sulla programmazione multithread che sfrutta thread multipli del kernel offre prestazioni migliori di una soluzione a singolo thread su un sistema monoprocesso?

4.10 Quali tra i seguenti componenti dello stato di un programma sono condivisi tra thread in un processo multithread?

- a. Valori dei registri.
- b. Memoria heap.
- c. Variabili globali.
- d. Stack.

4.11 È possibile che una soluzione multithread, impiegando thread multipli a livello utente, consegua prestazioni migliori su un sistema multiprocessore piuttosto che su un sistema a singolo processore? Motivate la risposta.

4.12 Nel Capitolo 3 abbiamo illustrato il browser Google Chrome e il suo modo di aprire ogni nuovo sito web in un processo separato. Si sarebbero ottenuti gli stessi vantaggi se Chrome fosse stato progettato per aprire ogni nuovo sito web in un thread separato? Motivate la vostra risposta.

4.13 È possibile avere la concorrenza, ma non il parallelismo? Motivate la vostra risposta.

4.14 Usando la legge di Amdahl, calcolate il guadagno in termini di velocità di un'applicazione che ha una componente parallela del 60 per cento in caso di utilizzo per le seguenti applicazioni:

- componente parallela del 40% con (a) otto core di elaborazione e (b) sedici core di elaborazione
- componente parallela del 67% con (a) due core di elaborazione e (b) quattro core di elaborazione
- componente parallela del 90 % con (a) quattro core di elaborazione e (b) otto core di elaborazione

4.15 Determinate se i seguenti problemi presentano un parallelismo dei dati o un parallelismo delle attività (task).

- Usare un thread separato per generare una miniatura di ciascuna foto di una raccolta
- Trasporre una matrice in parallelo
- Una applicazione in rete dove un thread legga dalla rete e un altro thread scriva sulla rete
- L'applicazione della somma dell'array fork-join descritta nel Paragrafo 4.5.2
- Il sistema Grand Central Dispatch

4.16 Un sistema con due processori dual-core ha quattro processori disponibili per lo scheduling. Su questo sistema è in esecuzione un'applicazione con uso intensivo della cpu. Tutta la fase di input, in cui deve essere aperto un unico file, viene gestita all'avvio del programma. Allo stesso modo, tutto l'output è gestito appena prima che il programma termini, mediante il salvataggio dei risultati del programma in un unico file. Tra l'avvio e la terminazione, il programma utilizza esclusivamente la cpu. Il vostro compito è quello di migliorare le prestazioni di questa applicazione rendendola multithread. L'applicazione viene eseguita in un sistema che utilizza il modello da uno a uno (ogni thread utente viene mappato in un thread del kernel).

- Quanti thread creerete per gestire l'input e l'output? Spieghetelo.
- Quanti thread creerete per la porzione di applicazione a uso intensivo della cpu? Spieghetelo.

4.17 Si consideri il seguente frammento di codice:

```
pid_t pid;

pid = fork();      if (pid == 0)
{ /* processo figlio */          fork();      thread_create(...);      }
fork();
```

- a. Quanti processi distinti vengono creati?
- b. Quanti thread distinti vengono creati?

4.18 Come descritto nel Paragrafo 4.7.2 Linux non fa distinzione tra processi e thread, cosicché un task apparirà più affine a un processo, oppure a un thread, a seconda dell'insieme di flag passati alla chiamata di sistema `clone()`. Tuttavia altri sistemi operativi,

come Windows, trattano processi e thread in maniera diversa. In genere, per descrivere un processo, questi sistemi usano strutture dati contenenti un puntatore per ciascun thread appartenente al processo. Ponete a confronto queste due tecniche per rappresentare i processi e i thread nel kernel.

4.19 Il programma contenuto nella Figura 4.23 utilizza la api Pthreads. Quali dati in uscita verrebbero prodotti dal programma alla LINEA C e alla LINEA P?

```
#include <pthread.h>
#include <stdio.h>
#include <types.h>

int value = 0;

void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
    pid_t pid;
    pthread_t tid;
    pthread_attr_t attr;
    pid = fork();
    if (pid == 0) { /* processo figlio */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: value = %d", value); /* LINEA C */
    }
    else if (pid > 0) { /* processo padre */
        wait(NULL);
        printf("PARENT: value = %d", value); /* LINEA P */
    }
}

void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}
```

Figura 4.23 Programma C dell'Esercizio 4.19.

4.20 Considerate un sistema multicore e un programma multithread scritto con il modello da molti a molti. Ipotizziamo un numero più alto di thread a livello utente nel programma rispetto al numero di processori nel sistema. Analizzate che cosa implichì, in termini di prestazioni, ciascuna delle seguenti possibilità.

- a. Il numero di thread del kernel assegnati al programma è minore del numero di core.
- b. I thread del kernel assegnati al programma sono in numero uguale al numero dei core.
- c. Il numero di thread del kernel assegnati al programma è maggiore del numero di core, ma minore del numero di thread a livello utente.

4.21 Pthread fornisce una api per la gestione della cancellazione dei thread. La funzione `pthread_setcancelstate()` viene utilizzata per impostare lo stato di cancellazione. Il suo prototipo si presenta così:

```
pthread_setcancelstate(int state, int *oldstate)
```

I due possibili valori per lo stato sono `PTHREAD_CANCEL_ENABLE` e `PTHREAD_CANCEL_DISABLE`. Utilizzando il frammento di codice illustrato nella Figura 4.24, fornite esempi di due operazioni che sarebbero adatte da eseguire tra le chiamate di disabilitazione e abilitazione della cancellazione.

```
int oldstate;  
  
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);  
  
/* Quali operazioni eseguire in questo punto? */  
  
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
```

Figura 4.24 Programma C dell'Esercizio 4.21.

4.22 Scrivete un programma multithread che calcoli diversi valori statistici su una lista di numeri. Questo programma riceverà una serie di numeri dalla riga di comando e quindi creerà tre thread di lavoro separati. Il primo thread determinerà la media dei numeri, il secondo determinerà il valore massimo e il terzo determinerà il valore minimo. Si supponga per esempio che al programma vengano passati i numeri interi:

90 81 78 95 79 72 85

Il programma risponderà:

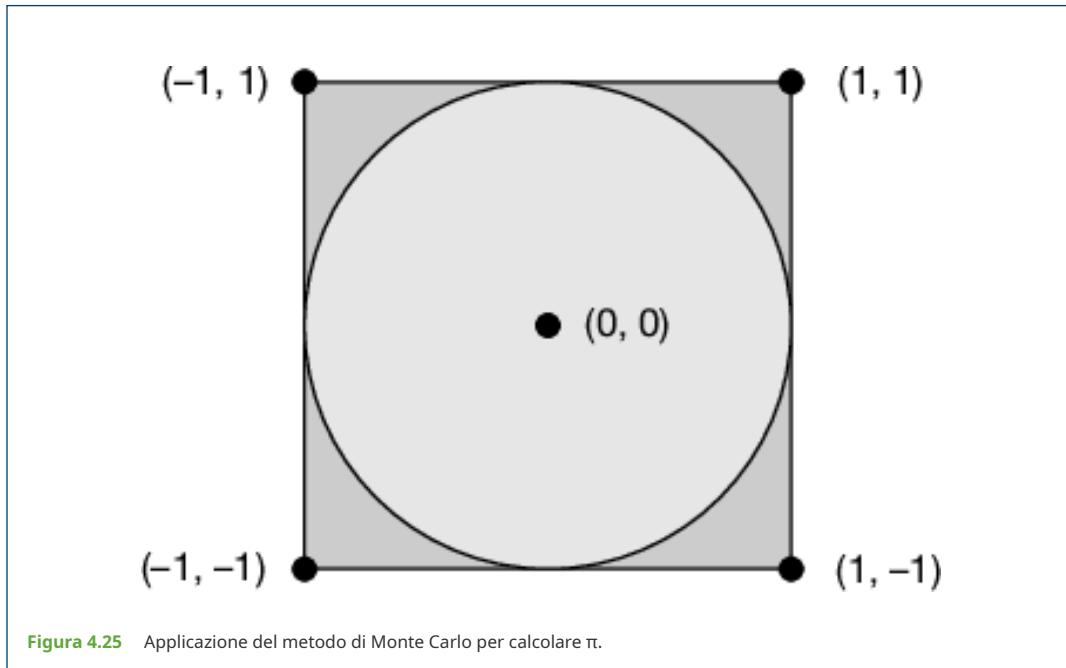
```
Il valore medio è 82  
Il valore minimo è 72  
Il valore massimo è 95
```

Le variabili che rappresentano la media, il minimo e il massimo saranno salvate a livello globale. I thread di lavoro potranno impostare questi valori, e il thread genitore li restituirà una volta che i figli avranno terminato. (Si potrebbe ovviamente espandere questo programma con la creazione di thread aggiuntivi per determinare altri valori statistici, come la mediana e la deviazione standard).

4.23 Scrivete un programma multithread che generi numeri primi. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce un numero alla riga di comando; il programma crea un thread distinto che riporta tutti i numeri primi minori o uguali al numero inserito dall'utente.

4.24 Un modo interessante per calcolare π consiste nell'utilizzare una tecnica nota come metodo di Monte Carlo, che utilizza numeri generati casualmente. La tecnica funziona nel modo seguente.

Supponete di avere un cerchio inscritto in un quadrato, come mostrato nella Figura 4.25 (assumiamo che il raggio sia unitario).



- Per prima cosa, generate una serie di punti casuali sotto forma di coordinate (x, y) . Questi punti devono cadere all'interno delle coordinate che limitano il quadrato. Fra tutti i punti generati, una parte ricadrà all'interno del cerchio.
- Quindi stimate π effettuando il seguente calcolo:

$$\pi = 4 \times (\text{numero di punti interni al cerchio}) / (\text{numero totale di punti})$$

Scrivete un versione multithreaded di questo algoritmo che crea un thread separato per generare un certo numero di punti casuali. Il thread conta il numero di punti all'interno del cerchio e memorizza il risultato in una variabile globale. Terminata l'esecuzione del thread, il thread padre calcolerà e scriverà in output il valore stimato di π . È utile fare esperimenti con diversi numeri di punti generati. Come regola generale, maggiore è il numero di punti, migliore è l'approssimazione di π .

Nel codice sorgente scaricabile per questo testo troverete un programma di esempio che fornisce una tecnica per generare numeri casuali e un metodo per determinare se il punto casuale (x, y) ricade all'interno del cerchio.

Coloro che sono interessati ai dettagli del metodo Monte Carlo per stimare π consultino la bibliografia alla fine del capitolo. Nel Capitolo 6 modificheremo questo esercizio utilizzando nozioni presentate in quel capitolo.

4.25 Ripetete l'Esercizio 4.24 ma, invece di usare un thread separato per generare punti casuali, utilizzate OpenMP per parallelizzare la generazione dei punti. Abbiate cura di non inserire il calcolo di π nella regione parallela, perché volete calcolarne il valore una volta sola.

4.26 Modificate il server basato sulle socket della Figura 3.27 nel Capitolo 3 in modo che esso dedichi un thread separato a ciascuna richiesta del client.

4.27 La successione di Fibonacci inizia con 0, 1, 1, 2, 3, 5, 8,

Essa è definita da:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

Scrivete un programma multithread che generi la successione di Fibonacci. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce alla riga di comando il numero di termini della successione di Fibonacci che il programma deve generare. Il programma crea un thread separato per la generazione dei numeri di Fibonacci, ma colloca i termini della successione in dati condivisi dai thread (un vettore è probabilmente la struttura dati più adatta). Quando il thread figlio conclude l'esecuzione, il thread genitore emette la sequenza generata dal figlio. Poiché il thread genitore non può produrre in uscita la sequenza prima che il thread figlio abbia terminato, sarà necessario applicare la tecnica illustrata nel Paragrafo 4.4, per sincronizzare il thread genitore con il thread figlio.

4.28 Modificate il Problema di programmazione 3.20 del Capitolo 3, che richiede di progettare un gestore di pid, scrivendo un programma multithread per testare la vostra soluzione all'Esercizio 3.20. Dovrete creare un certo numero di thread, per esempio 100, e ogni thread richiederà un pid, resterà sospeso per un periodo di tempo casuale e quindi rilascerà il pid. (La sospensione per un periodo di tempo casuale approssima il tipico utilizzo dei pid, in cui un pid viene assegnato a un nuovo processo, il processo viene eseguito e quindi termina, e il pid, al termine del processo, viene rilasciato). Sui sistemi unix e Linux questa sospensione è realizzata attraverso la funzione `sleep()`, alla quale viene passato un intero che rappresenta il numero di secondi. Questo problema verrà modificato nel Capitolo 7.

4.29 L'Esercizio 3.25 nel Capitolo 3 richiede di progettare un server che effettua l'eco utilizzando la api Java per il threading. Quel server utilizza un singolo thread, ossia non può rispondere a richieste di eco concorrenti finché il client che viene servito non esce. Modificate la soluzione dell'Esercizio 3.25 in modo che il server gestisca separatamente più richieste di eco.

Progetto 1 – Validatore di soluzioni di Sudoku

Un *Sudoku* è una griglia 9×9 in cui ogni colonna, ogni riga e ciascuna delle nove sottogrille 3×3 devono contenere tutte le cifre da 1 a 9. La Figura 4.26 mostra un esempio di Sudoku valido. Questo esercizio consiste nel progettare un'applicazione multithread per determinare se la soluzione di un Sudoku è valida. Ci sono diversi modi per rendere multithread una tale applicazione. Una strategia che suggeriamo è quella di creare thread per controllare i seguenti criteri:

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

Figura 4.26 Soluzione di un Sudoku 9 × 9.

- Un thread per verificare che ogni colonna contenga le cifre da 1 a 9
- Un thread per verificare che ogni riga contenga le cifre da 1 a 9
- Nove thread per verificare che ciascuna sottogriglia 3 × 3 contenga le cifre da 1 a 9

Per la convalida di un Sudoku si utilizzerebbe in questo caso un totale di undici thread distinti. Siete comunque liberi di creare ancora più thread. Per esempio, invece di creare un thread che controlli tutte e nove le colonne, potete creare nove thread, ognuno dei quali controlla una singola colonna.

Passaggio di parametri a ogni thread

Il thread genitore crea i thread di lavoro, passando a ognuno la posizione che deve controllare nella griglia del Sudoku. Questo passo richiede il passaggio di diversi parametri a ogni thread. L'approccio più semplice è quello di creare una struttura dati utilizzando una `struct`. Per esempio, una struttura per passare riga e colonna da cui un thread deve iniziare la convalida potrebbe essere la seguente:

```
/* struttura per il passaggio di dati ai thread */

typedef struct

{
    int row;
    int column;
} parameters;
```

Sia i programmi Pthreads sia quelli Windows creeranno thread di lavoro utilizzando una strategia simile a quella mostrata di seguito:

```
parameters *data = (parameters *) malloc(sizeof(parameters));  
  
data->row = 1;  
  
data->column = 1;  
  
/* Ora create il thread passandogli data come parametro */
```

Il puntatore `data` sarà passato alla funzione `pthread_create()` (Pthreads) o alla funzione `createThread()` (Windows), che a loro volta lo passeranno come parametro alla funzione che deve essere eseguita in un thread separato.

Restituzione dei risultati al thread genitore

A ogni thread di lavoro viene assegnato il compito di determinare la validità di una particolare regione del Sudoku. Una volta che uno di questi thread ha eseguito il controllo, esso deve restituire i risultati al genitore. Un buon modo per gestire tale situazione è quello di creare un array di valori interi visibile a ogni thread. L'indice *i-esimo* di questo array corrisponde al thread di lavoro *i-esimo*. Un thread imposta il corrispondente valore a 1 per indicare che la regione del Sudoku da lui controllata sia valida; il valore 0 indica invece il contrario. Quando tutti i thread di lavoro hanno completato l'esecuzione, il thread genitore controlla ogni voce dell'array risultato per determinare se il Sudoku sia valido.

Progetto 2 – Programma di ordinamento multithread

Scrivete un programma di ordinamento multithread che funziona come segue. Un vettore di numeri interi viene suddiviso in due vettori più piccoli di dimensioni uguali. Due thread separati (che chiameremo thread di ordinamento) ordinano ogni sottovettore utilizzando un algoritmo di ordinamento di vostra scelta. I due sottovettori sono poi uniti da un terzo thread (il thread di fusione) che forma così un unico vettore ordinato.

Il modo più semplice per gestire i dati è forse quello di creare un array globale, poiché i dati globali sono condivisi tra tutti i thread. Ogni thread di ordinamento lavorerà su una metà di questo array. Verrà inoltre creato un secondo array globale di pari dimensione, utilizzato dal thread di fusione per inserire l'unione dei due sottovettori. Graficamente, questo programma è strutturato come mostrato nella Figura 4.27.

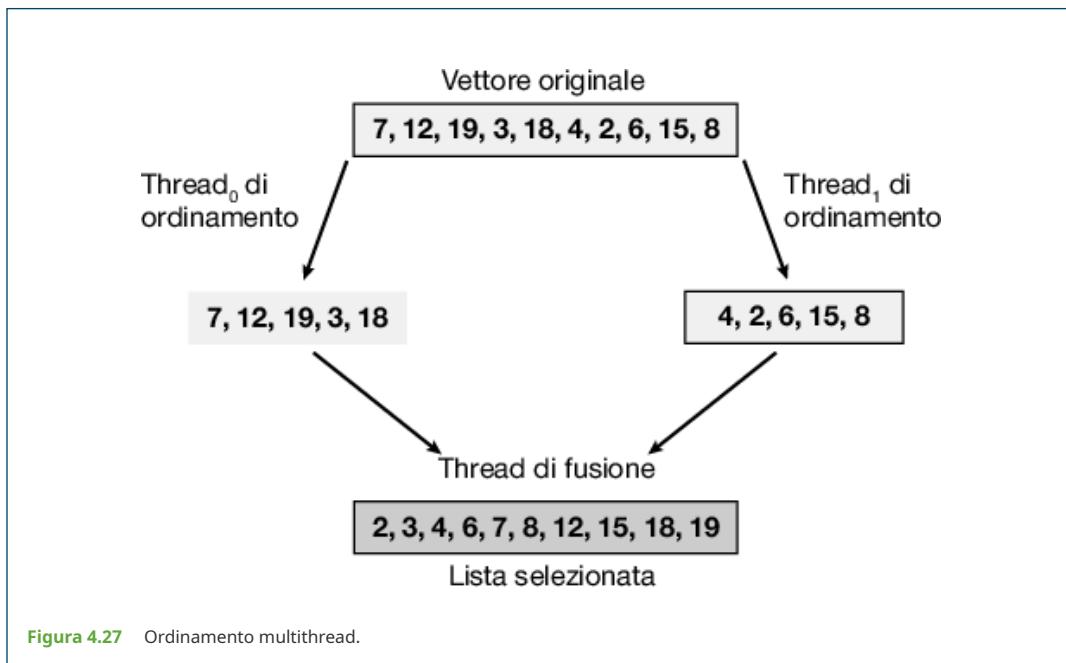


Figura 4.27 Ordinamento multithread.

Questo progetto di programmazione richiederà il passaggio di parametri a ciascuno dei thread di ordinamento. In particolare, sarà necessario individuare l'indice da cui ciascun thread deve iniziare l'ordinamento.

Fare riferimento alle istruzioni contenute nel Progetto 1 per i dettagli sul passaggio di parametri a un thread.

Il thread genitore darà in output l'array ordinato una volta che ogni thread di lavoro abbia terminato la sua esecuzione.

Progetto 3 – Applicazione di ordinamento fork-join

Implementate il progetto precedente (Ordinamento Multithreaded) utilizzando la API Java che supporta il parallelismo fork-join. Questo progetto deve essere sviluppato in due versioni differenti. Ciascuna versione implementerà un diverso algoritmo di tipo divide-and-conquer:

1. Quicksort
2. Mergesort

L'implementazione Quicksort utilizzerà l'algoritmo Quicksort per dividere la lista degli elementi da ordinare in una parte destra e una parte sinistra a seconda della posizione del valore pivot. L'algoritmo Mergesort invece divide la lista in due parti della stessa dimensione. Per entrambi gli algoritmi, quando la lista da ordinare diventa più piccola di un valore di soglia (per esempio contiene 100 elementi o meno), applicate un algoritmo semplice come Selection o Insertion sort.

La maggior parte dei testi sulle strutture dati descrive questi noti algoritmi di ordinamento di tipo divide-and-conquer.

La classe `SumTask` mostrata nel Paragrafo 4.5.2.1 estende `RecursiveTask`, che è un `ForkJoinTask` che restituisce un risultato. Dato che questo problema richiede di ordinare il vettore che viene passato al task, ma senza restituire alcun risultato, creerete invece una classe che estende `RecursiveAction`, un `ForkJoinTask` che non restituisce un risultato (si veda la Figura 4.19).

Gli oggetti che vengono passati agli algoritmi di ordinamento devono implementare l'interfaccia Java `Comparable` e questo requisito deve essere realizzato nella definizione della classe in ciascun algoritmo di ordinamento. Il codice sorgente scaricabile per questo testo include codice Java che fornisce le basi per intraprendere tale progetto.

CAPITOLO 5

Scheduling della CPU

Lo scheduling della cpu è alla base dei sistemi operativi multiprogrammati: attraverso la commutazione della cpu tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore. In questo capitolo s'introducono i concetti fondamentali dello scheduling e si descrivono vari algoritmi di scheduling della cpu. Si affronta inoltre il problema della scelta dell'algoritmo da impiegare per un dato sistema.

Nel Capitolo 4 abbiamo arricchito il concetto di processo introducendo i thread. Nei sistemi operativi che li supportano, in effetti sono i thread, e non i processi, l'oggetto dell'attività di scheduling. Ciononostante, le locuzioni scheduling dei processi e scheduling dei thread sono spesso considerate equivalenti. In questo capitolo useremo la prima nell'analizzare i principi generali dello scheduling e la seconda nel trattare idee specificamente inerenti ai thread.

Analogamente nel Capitolo 1 abbiamo visto che un core di elaborazione è l'unità di calcolo di base di una cpu e che un processo viene eseguito su un core. Tuttavia, utilizzeremo in questo capitolo la terminologia più diffusa: quando parliamo dello scheduling di un processo che deve essere "eseguito su una cpu" stiamo sottintendendo che il processo sarà in esecuzione su un core della cpu.

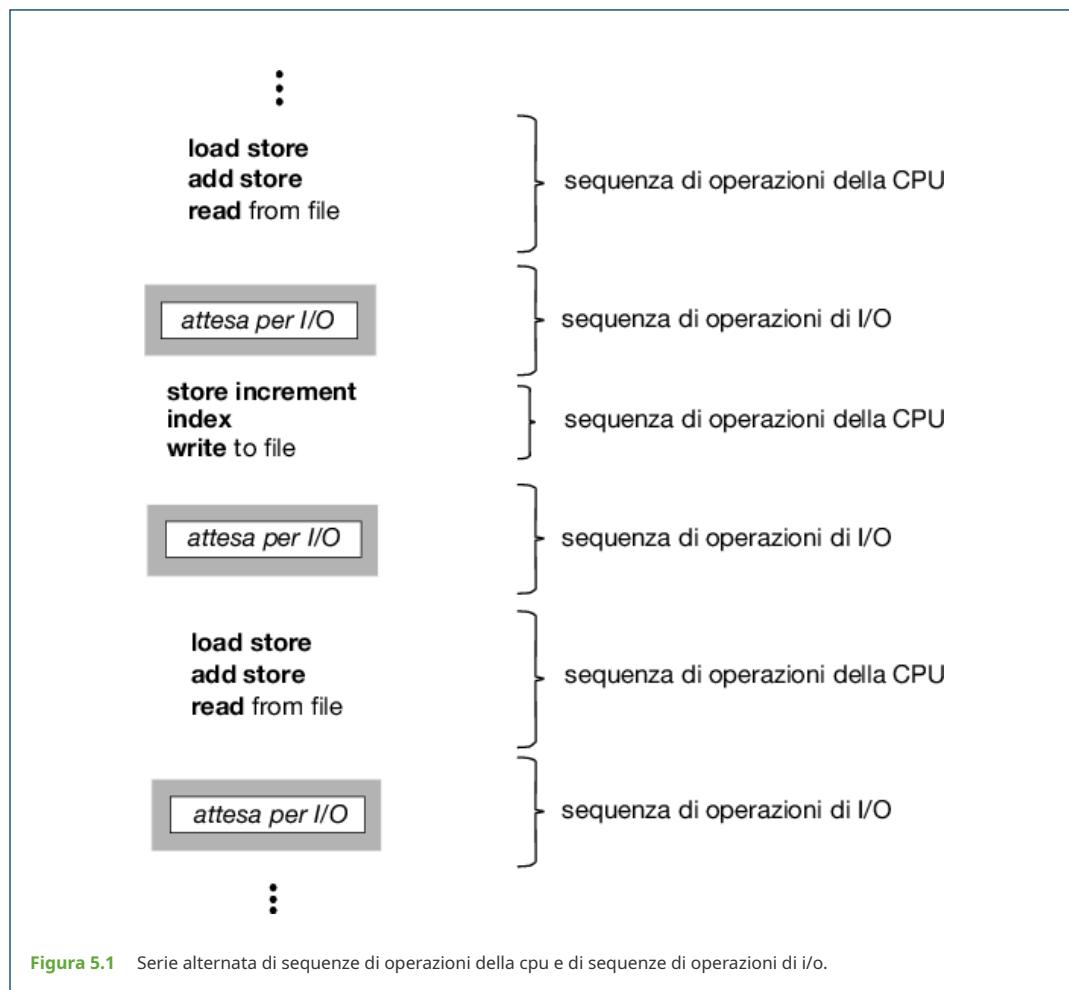
5.1 Concetti fondamentali

In un sistema dotato di un singolo core di elaborazione si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la cpu sia libera e possa essere nuovamente sottoposta a scheduling. L'obiettivo della multiprogrammazione è avere sempre un processo in esecuzione, in modo da massimizzare l'utilizzazione della cpu. L'idea è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di i/o; durante l'attesa, in un sistema di calcolo semplice, la cpu resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della cpu per cederlo a un altro processo. Su un sistema multicore questo concetto di mantenere occupata la cpu è esteso a tutti i core di elaborazione del sistema.

Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la cpu è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

5.1.1 Ciclicità delle fasi d'elaborazione e di i/o

Il successo dello scheduling della cpu dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un ciclo d'elaborazione (svolta dalla cpu) e d'attesa del completamento delle operazioni di i/o. I processi si alternano tra questi due stati. L'esecuzione di un processo comincia con una sequenza di operazioni d'elaborazione svolte dalla cpu (*cpu burst*), seguita da una sequenza di operazioni di i/o (*i/o burst*), quindi un'altra sequenza di operazioni della cpu, di nuovo una sequenza di operazioni di i/o, e così via. L'ultima sequenza di operazioni della cpu si conclude con una richiesta al sistema di terminare l'esecuzione (Figura 5.1).



Le durate delle sequenze di operazioni della cpu (dette anche “burst della cpu”) sono state misurate in molte sperimentazioni, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella Figura 5.2. La curva è generalmente di tipo esponenziale o iperesponenziale, con molte brevi sequenze di operazioni della cpu, e poche sequenze di operazioni della cpu molto lunghe. Un programma con prevalenza di i/o (*i/o bound*) produce generalmente molte sequenze di operazioni della cpu di breve durata. Un programma con prevalenza d’elaborazione (*cpu bound*), invece, può produrre varie sequenze di operazioni della cpu molto lunghe. Questa distribuzione può essere utile nella scelta di un appropriato algoritmo di scheduling della cpu.

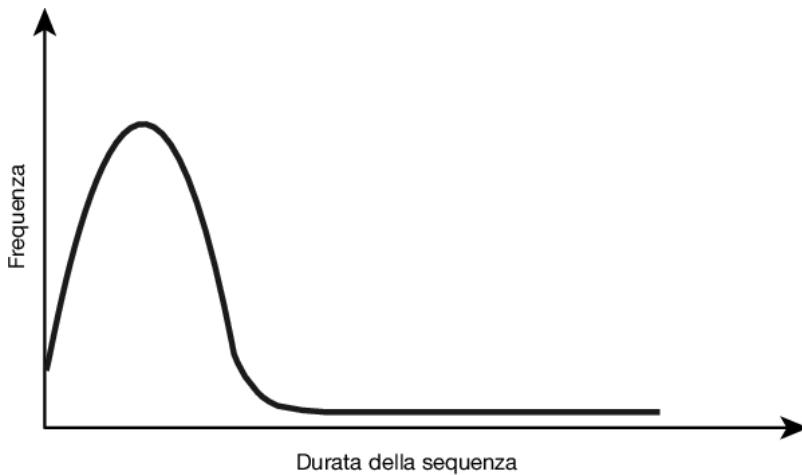


Figura 5.2 Diagramma delle durate delle sequenze di operazioni della cpu.

5.1.2 Scheduler della cpu

Ognqualvolta la cpu passa nello stato d’inattività, il sistema operativo sceglie per l’esecuzione uno dei processi presenti nella ready queue. In particolare, è lo scheduler a breve termine, o scheduler della cpu che, tra i processi in memoria pronti per l’esecuzione, sceglie quello cui assegnare la cpu.

La ready queue non è necessariamente una coda in ordine d’arrivo (*first-in, first-out* o fifo). Come vedremo analizzando i diversi algoritmi di scheduling, una ready queue si può realizzare come una coda fifo, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i processi della ready queue sono posti nella lista d’attesa per accedere alla cpu. Generalmente gli elementi delle code sono i *process control block* (pcb) dei processi.

5.1.3 Scheduling con e senza prelazione

Le decisioni riguardanti lo scheduling della cpu vengono prese nelle seguenti quattro circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di i/o o invocazione di `wait()` per la terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d’interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un’operazione di i/o);
4. un processo termina.

I casi 1 e 4 non danno alternative in termini di scheduling: si deve comunque scegliere un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella ready queue per l’esecuzione. Una scelta si può invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è senza prelazione (*nonpreemptive*) o cooperativo (*cooperative*); altrimenti, lo schema di scheduling è con prelazione (*preemptive*). Nel caso dello scheduling senza prelazione, quando si assegna la cpu a un processo, questo rimane in possesso della cpu fino al momento del suo rilascio, dovuto al termine dell’esecuzione o al passaggio nello stato di attesa. Praticamente tutti i moderni sistemi operativi, inclusi Windows, macos, Linux e unix, utilizzano algoritmi di scheduling con prelazione.

Sfortunatamente lo scheduling con prelazione può portare a *race condition* quando i dati sono condivisi tra diversi processi. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell’esecuzione dell’altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Tali questioni saranno trattate in dettaglio nel Capitolo 6.

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività per conto di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di i/o. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos. Come verrà discusso nel Paragrafo 6.2, i kernel dei sistemi operativi possono essere progettati con o senza prelazione. Un kernel senza prelazione attenderà che una chiamata di sistema venga completata o che un processo si blocchi in attesa di terminare l'i/o prima di eseguire un cambio di contesto. Questo schema garantisce che la struttura del kernel sia semplice, poiché il kernel non può esercitare la prelazione su un processo mentre le sue strutture dati si trovano in uno stato incoerente. Sfortunatamente, questo modello d'esecuzione non è adeguato alle elaborazioni in tempo reale, in cui i task devono essere conclusi entro un intervallo fissato di tempo. I requisiti di scheduling per sistemi real-time sono descritti nel Paragrafo 5.6. Un kernel con prelazione richiede meccanismi come i lock mutex per prevenire le race condition quando si accede a strutture dati condivise del kernel. La maggior parte dei sistemi operativi moderni è interamente con prelazione durante l'esecuzione in modalità kernel.

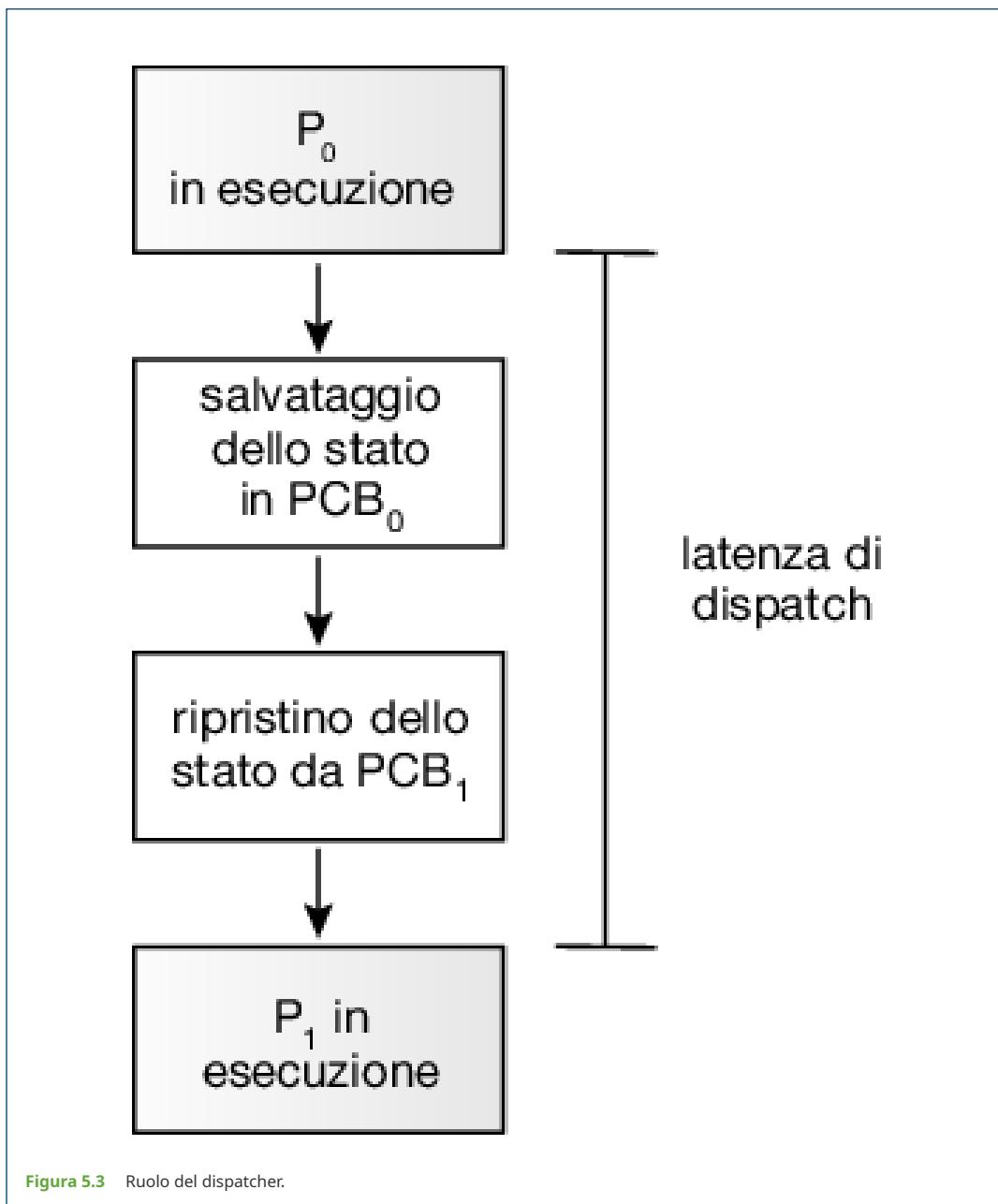
Poiché le interruzioni si possono, per definizione, verificare in ogni istante e il kernel non può sempre ignorarle, le sezioni di codice eseguite per effetto delle interruzioni devono essere protette da un uso simultaneo. Il sistema operativo deve ignorare raramente le interruzioni, altrimenti si potrebbero perdere dati in ingresso, o si potrebbero sovrascrivere dati in uscita. Per evitare che più processi accedano in modo concorrente a tali sezioni di codice, queste disattivano le interruzioni al loro inizio e le riattivano alla fine. Le sezioni di codice che disabilitano le interruzioni si verificano tuttavia raramente e, in genere, non contengono molte istruzioni.

5.1.4 Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della cpu è il dispatcher; si tratta del modulo che passa effettivamente il controllo della cpu al processo scelto dallo scheduler a breve termine. Questa funzione comprende:

- il cambio di contesto da un processo a un altro;
- il passaggio alla modalità utente;
- il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è noto come latenza di dispatch ed è mostrato nella Figura 5.3.



Una questione interessante da considerare è quanto spesso avvengono i cambi di contesto. A livello di sistema, il numero di cambi di contesto può essere ottenuto utilizzando il comando `vmstat` disponibile sui sistemi Linux. Di seguito è mostrata una parte dell'output del comando

```
vmstat 1 3
```

Questo comando fornisce 3 righe di output con un ritardo di un secondo tra gli aggiornamenti:

```
---cpu---
```

```
24
```

```
225
```

```
339
```

La prima riga fornisce il numero medio di cambi di contesto al secondo da quando il sistema è stato avviato e le due righe successive danno il numero di cambi di contesto nei due precedenti intervalli di 1 secondo. Da quando questa macchina è stata avviata c'è stata una media di 24 cambi di contesto al secondo; nel secondo appena trascorso sono avvenuti 225 cambi di contesto, mentre ne sono occorsi 339 nel secondo precedente.

È anche possibile usare il file system `/proc` per determinare il numero di cambi di contesto per un determinato processo. Il contenuto del file `/proc/2166/status`, per esempio, offre varie statistiche relative al processo con `pid = 2166`. Il comando

```
cat /proc/2166/status
```

fornisce il seguente output (ne mostriamo solo una parte):

```
voluntary_ctxt_switches      150

nonvoluntary_ctxt_switches    8
```

Queste linee mostrano il numero di cambi di contesto avvenuti durante l'esistenza del processo, distinguendo tra cambi di contesto volontari e involontari. Un cambio di contesto volontario si verifica quando un processo cede il controllo della cpu in seguito alla richiesta di una risorsa che al momento non è disponibile (per esempio, quando è bloccato in attesa di i/o). Un cambio di contesto involontario si verifica quando la cpu viene sottratta a un processo, per esempio quando il suo quanto di tempo è scaduto oppure quando è stata esercitata la prelazione da parte di un processo con priorità più alta.

5.2 Criteri di scheduling

Diversi algoritmi di scheduling della cpu hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi.

Per il confronto tra gli algoritmi di scheduling della cpu sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore. Di seguito si riportano alcuni criteri.

- Utilizzo della cpu. La cpu deve essere più attiva possibile. Teoricamente, l'utilizzo della cpu può variare dallo 0 al 100 per cento. In un sistema reale dovrebbe variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con carico elevato. (L'utilizzo della cpu può essere ottenuto utilizzando il comando `top` su sistemi Linux, macos e unix).
- Throughput. La cpu è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta produttività (*throughput*). Per processi di lunga durata il suo valore può essere di un processo all'ora, mentre per brevi transazioni è possibile avere un throughput di 10 processi al secondo.
- Tempo di completamento. Dal punto di vista di uno specifico processo, il criterio più importante è il tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato tempo di completamento (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella ready queue, durante l'esecuzione nella cpu e nelle operazioni di i/o.
- Tempo d'attesa. L'algoritmo di scheduling della cpu non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di i/o; influisce solo sul tempo d'attesa nella ready queue. Il tempo d'attesa è la somma degli intervalli d'attesa passati in questa coda.
- Tempo di risposta. In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase di output per l'utente. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la effettuazione di una richiesta e la prima risposta prodotta. Questa misura è chiamata tempo di risposta, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo necessario per completare l'output.

È auspicabile aumentare al massimo utilizzo e produttività della cpu, mentre il tempo di completamento, il tempo d'attesa e il tempo di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi; tuttavia in alcune circostanze è più opportuno ottimizzare i valori minimi o massimi, anziché i valori medi; per esempio, per garantire che tutti gli utenti ottengano un buon servizio, possiamo voler ridurre il massimo tempo di risposta.

Per i sistemi interattivi, come i sistemi desktop e laptop, alcuni analisti suggeriscono che sia più importante ridurre al minimo la varianza del tempo di risposta anziché il tempo medio di risposta. Un sistema il cui tempo di risposta sia ragionevole e prevedibile può essere considerato migliore di un sistema mediamente più rapido, ma molto variabile. Tuttavia, è stato fatto poco sugli algoritmi di scheduling della cpu al fine di ridurre al minimo la varianza.

Nell'analizzare i diversi algoritmi di scheduling della cpu dobbiamo esemplificare il funzionamento. Una descrizione approfondita richiederebbe il ricorso a molti processi, ognuno dei quali costituito da parecchie centinaia di sequenze di operazioni della cpu e di sequenze di operazioni di i/o. Per motivi di semplicità, negli esempi si considera una sola sequenza di operazioni della cpu (la cui durata è espressa in millisecondi) per ogni processo. La misura di confronto adottata è il tempo d'attesa medio. Meccanismi di valutazione più raffinati sono trattati nel Paragrafo 5.8.

5.3 Algoritmi di scheduling

Lo scheduling della cpu si occupa di decidere quale dei processi nella ready queue debba essere assegnato al core della cpu. Esistono molti algoritmi differenti di scheduling della cpu e in questo paragrafo ne descriveremo diversi. Sebbene la maggior parte delle architetture delle cpu moderne abbia più core di elaborazione, descriviamo questi algoritmi di scheduling nel caso di un solo core di elaborazione disponibile, ovvero di una singola cpu con un singolo core di elaborazione, per cui il sistema è in grado di eseguire solo un processo alla volta. Nel Paragrafo 5.5 discuteremo lo scheduling della cpu nel contesto dei sistemi multiprocessore.

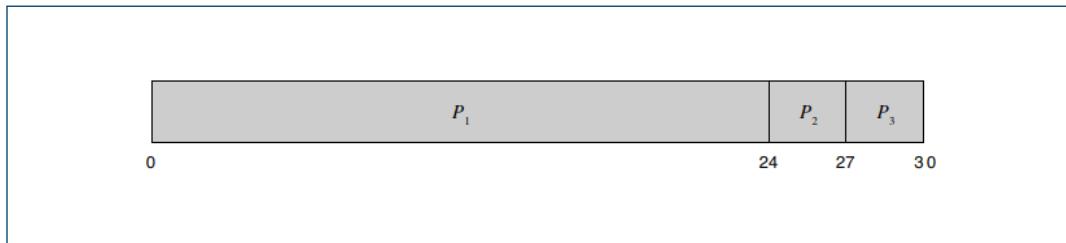
5.3.1 Scheduling in ordine d'arrivo

Il più semplice algoritmo di scheduling della cpu è l'algoritmo di scheduling in ordine d'arrivo (*scheduling first-come, first-served* o fcfs). Con questo schema la cpu si assegna al processo che la richiede per primo. La realizzazione del criterio fcfs si basa su una coda fifo. Quando un processo entra nella ready queue, si collega il suo pcb all'ultimo elemento della coda. Quando la cpu è libera, viene assegnata al processo che si trova alla testa della coda, rimuovendolo da essa. Il codice per lo scheduling fcfs è semplice sia da scrivere sia da capire.

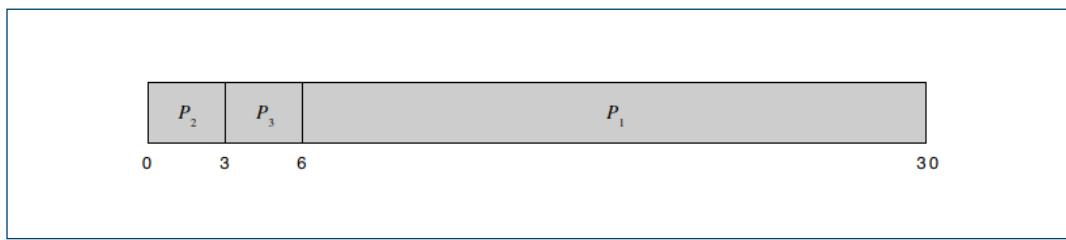
Un aspetto negativo è che il tempo medio d'attesa per l'algoritmo fcfs è spesso abbastanza lungo. Si consideri il seguente insieme di processi, che si presenta al momento 0, con la durata della sequenza di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell'ordine P_1, P_2, P_3 e sono serviti in ordine fcfs, si ottiene il risultato illustrato nel seguente diagramma di Gantt, un diagramma a barre che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.



Il tempo d'attesa è 0 millisecondi per il processo P_1 , 24 millisecondi per il processo P_2 e 27 millisecondi per il processo P_3 . Quindi, il tempo d'attesa medio è $(0 + 24 + 27)/3 = 17$ millisecondi. Se i processi arrivassero nell'ordine P_2, P_3, P_1 , i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:



Il tempo di attesa medio è ora di $(6 + 0 + 3)/3 = 3$ millisecondi. Si tratta di una notevole riduzione. Quindi, il tempo medio d'attesa in condizioni di fcfs non è in genere minimo, e può variare grandemente all'aumentare della variabilità dei cpu burst dei vari processi.

Si considerino inoltre le prestazioni dello scheduling fcfs in una situazione dinamica. Si supponga di avere un processo con prevalenza d'elaborazione e molti processi con prevalenza di i/o. Via via che i processi fluiscono nel sistema si può verificare la seguente

situazione. Il processo con prevalenza d'elaborazione occupa la cpu. Durante questo periodo tutti gli altri processi terminano le proprie operazioni di i/o e si spostano nella ready queue, nell'attesa della cpu. Mentre i processi si trovano nella ready queue, i dispositivi di i/o sono inattivi. Successivamente il processo con prevalenza d'elaborazione termina la propria sequenza di operazioni della cpu e passa a una fase di i/o. Tutti i processi con prevalenza di i/o, caratterizzati da sequenze di operazioni della cpu molto brevi, sono eseguiti rapidamente e tornano alle code di i/o, lasciando inattiva la cpu. Il processo con prevalenza d'elaborazione torna nella ready queue e riceve il controllo della cpu; così, finché non termina l'esecuzione del processo con prevalenza d'elaborazione, tutti i processi con prevalenza di i/o si trovano nuovamente ad attendere nella ready queue. Si ha un effetto convoglio, tutti i processi attendono che un lungo processo liberi la cpu, il che causa una riduzione dell'utilizzo della cpu e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling fcfs è senza prelazione; una volta che la cpu è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di i/o. L'algoritmo fcfs risulta particolarmente problematico nei sistemi in time-sharing, dove è importante che ogni utente disponga della cpu a intervalli regolari. Permettere a un solo processo di occupare la cpu per un lungo periodo condurrebbe a risultati disastrosi.

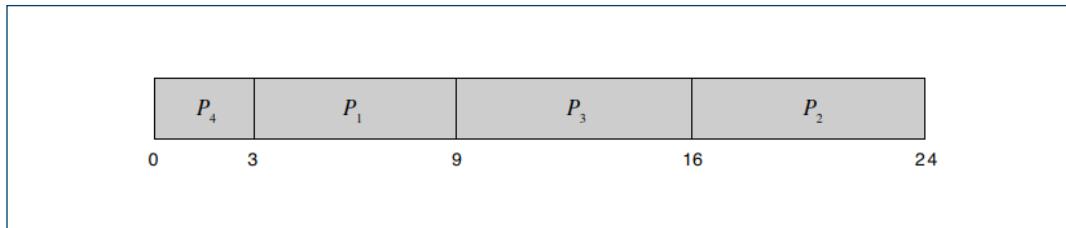
5.3.2 Scheduling shortest-job-first

Un criterio diverso di scheduling della cpu è l'algoritmo di scheduling per brevità (*shortest-job-first*, sjf). Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della cpu. Quando è disponibile, si assegna la cpu al processo che ha la più breve lunghezza della successiva sequenza di operazioni della cpu. Se due processi hanno le successive sequenze di operazioni della cpu della stessa lunghezza si applica lo scheduling fcfs. Si noti che sarebbe più appropriato il termine *shortest next cpu burst*, infatti lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della cpu del processo e non la sua lunghezza totale. Tuttavia, poiché è comunemente usato ed è presente nella maggior parte dei libri di testo, anche qui si fa uso del termine sjf.

Come esempio si consideri il seguente insieme di processi, con la durata della sequenza di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza
P_1	6
P_2	8
P_3	7
P_4	3

Con lo scheduling sjf questi processi si ordinerebbero secondo il seguente diagramma di Gantt.



Il tempo d'attesa è di 3 millisecondi per il processo P_1 , di 16 millisecondi per il processo P_2 , di 9 millisecondi per il processo P_3 e di 0 millisecondi per il processo P_4 . Quindi, il tempo d'attesa medio è di $(3 + 16 + 9 + 0)/4 = 7$ millisecondi. Usando lo scheduling fcfs, il tempo d'attesa medio sarebbe di 10,25 millisecondi.

Si può dimostrare che l'algoritmo di scheduling sjf è *ottimale*, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi. Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa *medio* diminuisce.

Sebbene sia ottimale, l'algoritmo sjf non si può realizzare a livello dello scheduling della cpu a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della cpu. Un possibile approccio a questo problema consiste nel tentare di approssimare lo scheduling sjf: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della cpu, si può cercare di *predire* il suo valore; è probabile, infatti, che sia simile alle precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

La lunghezza della successiva sequenza di operazioni della cpu generalmente si stima calcolando la media esponenziale delle lunghezze misurate delle precedenti sequenze di operazioni della cpu. La media esponenziale si definisce con la formula seguente. Siano t_n la lunghezza dell' n -esima sequenza di operazioni della cpu e t_{n+1} il valore previsto per la successiva sequenza. Allora, dato a tale che $0 \leq a \leq 1$, si definisce

$$t_{n+1} = at_n + (1 - a)t_n.$$

Il valore di t_n contiene le informazioni più recenti; t_n registra la storia passata. Il parametro a controlla il peso relativo sulla predizione della storia recente e di quella passata. Se $a = 0$, allora, $t_{n+1} = t_n$, e la storia recente non ha effetto (si suppone che le condizioni attuali siano transitorie); se $a = 1$, allora $t_{n+1} = t_n$, e ha significato solo la più recente sequenza di operazioni della cpu (si suppone che la storia sia irrilevante). Più comune è la condizione in cui $a = 1/2$, valore che indica che la storia recente e la storia passata hanno lo stesso peso. Il t_0 iniziale si può definire come una costante o come una media complessiva del sistema. Nella Figura 5.4 è illustrata una media esponenziale con $a = 1/2$ e $t_0 = 10$.

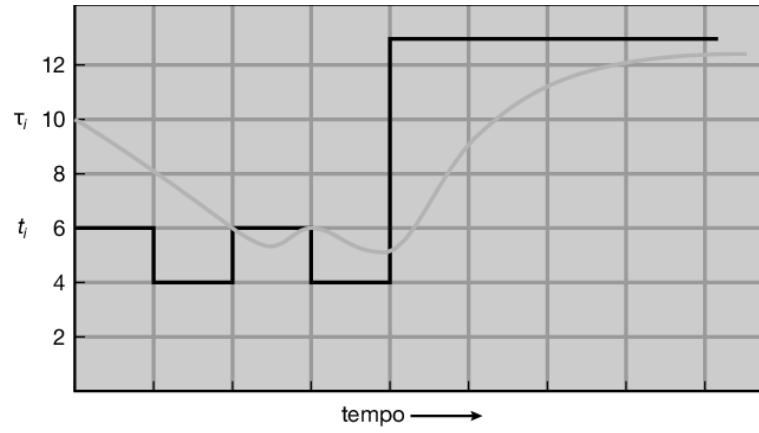


Figura 5.4 Predizione della lunghezza della successiva sequenza di operazioni della cpu (cpu burst).

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula per t_{n+1} sostituendo il valore di t_n , in modo da ottenere

$$t_{n+1} = at_n + (1 - a)at_{n-1} + \dots + (1 - a)^jat_{n-j} + \dots + (1 - a)^{n+1}t_0$$

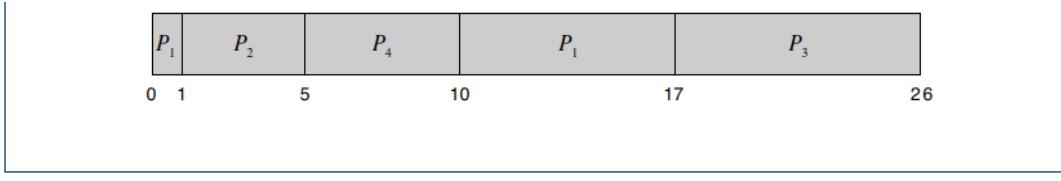
Di solito a è minore di 1. Quindi, anche $(1 - a)$ è minore di 1 e ogni termine ha peso inferiore a quello del suo predecessore.

L'algoritmo sjf può essere sia *con prelazione* sia *senza prelazione*. La scelta si presenta quando alla ready queue arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può richiedere una sequenza di operazioni della cpu più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo sjf con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo sjf senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della cpu. (Lo scheduling sjf con prelazione è talvolta chiamato scheduling *shortest-remaining-time-first*).

Come esempio, si considerino i quattro processi seguenti, dove la durata delle sequenze di operazioni della cpu è data in millisecondi:

Processo	Istante d'arrivo	Durata della sequenza
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Se i processi arrivano alla ready queue nei momenti indicati e richiedono i tempi di cpu illustrati, dallo scheduling sjf con prelazione risulta la sequenza indicata dal seguente diagramma di Gantt.



All'istante 0 si avvia il processo P_1 , poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo P_2 . Il tempo necessario per completare il processo P_1 (7 millisecondi) è maggiore del tempo richiesto dal processo P_2 (4 millisecondi), perciò si effettua la prelazione del processo P_1 sostituendolo col processo P_2 . Il tempo d'attesa medio per questo esempio è $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6,5$ millisecondi. Con uno scheduling sjf senza prelazione si otterebbe un tempo d'attesa medio di 7,75 millisecondi.

5.3.3 Scheduling circolare

L'algoritmo di scheduling circolare (*round-robin, rr*) è simile allo scheduling fcfs, ma aggiunge la capacità di prelazione in modo che il sistema possa commutare fra i vari processi. Ciascun processo riceve una piccola quantità fissata del tempo della cpu, chiamata quanto di tempo o porzione di tempo (*time slice*), che varia generalmente da 10 a 100 millisecondi; la ready queue è trattata come una coda circolare. Lo scheduler della cpu scorre la ready queue, assegnando la cpu a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

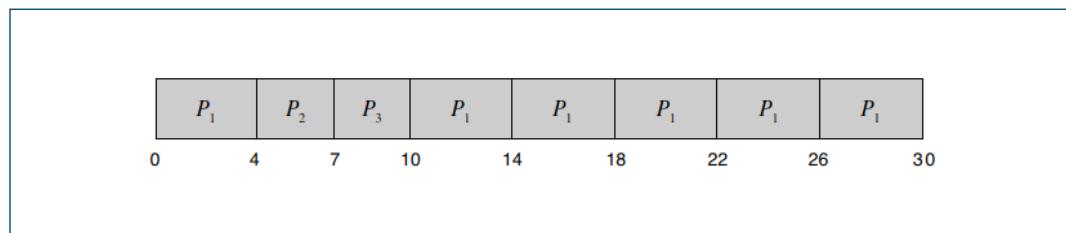
Per implementare lo scheduling rr si gestisce la ready queue come una coda fifo. I nuovi processi si aggiungono alla fine della ready queue. Lo scheduler della cpu prende il primo processo dalla ready queue, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per eseguire il processo.

A questo punto si può verificare una delle due seguenti situazioni: il processo ha una sequenza di operazioni della cpu di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la cpu e lo scheduler passa al processo successivo della ready queue; oppure la durata della sequenza di operazioni è più lunga di un quanto di tempo; in questo caso il timer scade e invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto e mette il processo alla fine della ready queue. Lo scheduler quindi seleziona il processo successivo nella ready queue.

Il tempo d'attesa medio per il criterio di scheduling rr è spesso abbastanza lungo. Si consideri il seguente insieme di processi che si presenta al tempo 0, con la durata delle sequenze di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se si usa un quanto di tempo di 4 millisecondi, il processo P_1 ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la cpu passa al processo successivo della coda, il processo P_2 . Poiché il processo P_2 non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la cpu al processo successivo, il processo P_3 . Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la cpu al processo P_1 per un ulteriore quanto di tempo. Dallo scheduling rr risulta quanto segue.

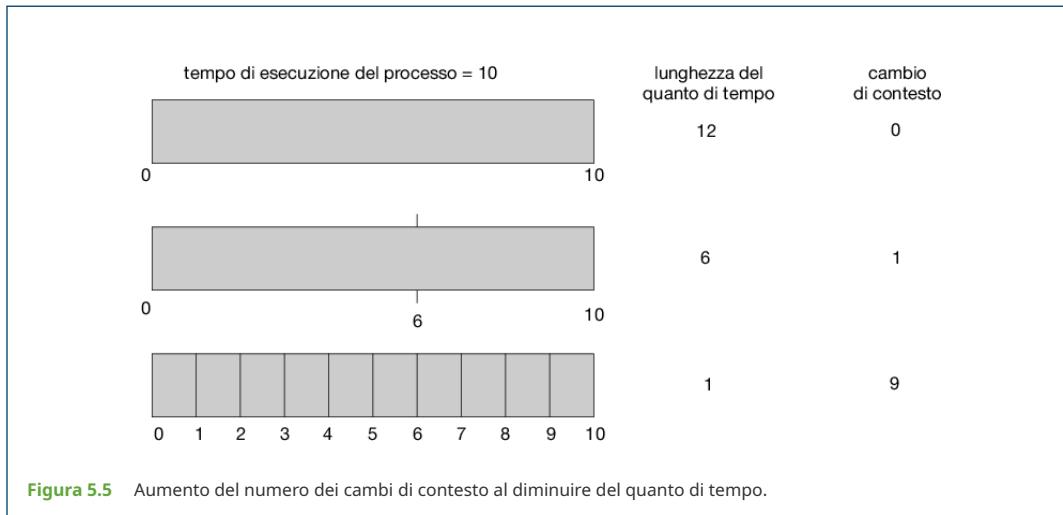


Calcoliamo ora il tempo di attesa medio per questa sequenza. P_1 resta in attesa per 6 millisecondi ($10 - 4$), P_2 per 4 millisecondi e P_3 per 7 millisecondi. Il tempo d'attesa medio è di $17/3 = 5,66$ millisecondi.

Nell'algoritmo di scheduling rr la cpu si assegna a un processo per non più di un quanto di tempo per volta (a meno che questo sia l'unico processo ready). Se la durata della sequenza di operazioni della cpu di un processo eccede il quanto di tempo, il processo viene sottoposto a prelazione e riportato nella ready queue. L'algoritmo di scheduling rr è pertanto con prelazione.

Se nella ready queue esistono n processi e il quanto di tempo è pari a q , ciascun processo ottiene $1/n$ -esimo del tempo di elaborazione della cpu in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1) \times q$ unità di tempo per il suo successivo quanto di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa fino a 20 millisecondi ogni 100 millisecondi.

Le prestazioni dell'algoritmo rr dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo, il criterio di scheduling rr si riduce al criterio di scheduling fcfs. Se il quanto di tempo è molto breve (per esempio, un millisecondo), il criterio rr può portare a un numero elevato di cambi di contesto. Assumiamo che, per esempio, ci sia un solo processo della durata di 10 unità di tempo, se il quanto di tempo è di 12 unità, il processo impiega meno di un quanto di tempo; se però il quanto di tempo è di 6 unità, il processo richiede 2 quanti di tempo e un cambio di contesto; e se il quanto di tempo è di un'unità di tempo, occorrono nove cambi di contesto, con proporzionale rallentamento dell'esecuzione del processo (Figura 5.5).



Quindi il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto; se, per esempio, questa è pari al 10 per cento del quanto di tempo, allora s'impiega in cambi di contesto circa il 10 per cento del tempo d'elaborazione della cpu. In pratica, nella maggior parte dei sistemi moderni un quanto di tempo va dai 10 ai 100 millisecondi. Il tempo richiesto per un cambio di contesto non eccede solitamente i 10 microsecondi, risultando quindi una modesta frazione del quanto di tempo.

Anche il tempo di completamento (*turnaround time*) dipende dalla dimensione del quanto di tempo: com'è evidenziato nella Figura 5.6, il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della dimensione del quanto di tempo. In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva sequenza di operazioni della cpu in un solo quanto di tempo. Per esempio, dati tre processi della durata di 10 unità di tempo ciascuno e un quanto di una unità di tempo, il tempo di completamento medio è di 29 unità. Se però il quanto di tempo è di 10 unità, il tempo di completamento medio scende a 20 unità. Aggiungendo il tempo del cambio di contesto, con un piccolo quanto di tempo, il tempo di completamento medio aumenta ancora poiché sono richiesti più cambi di contesto.

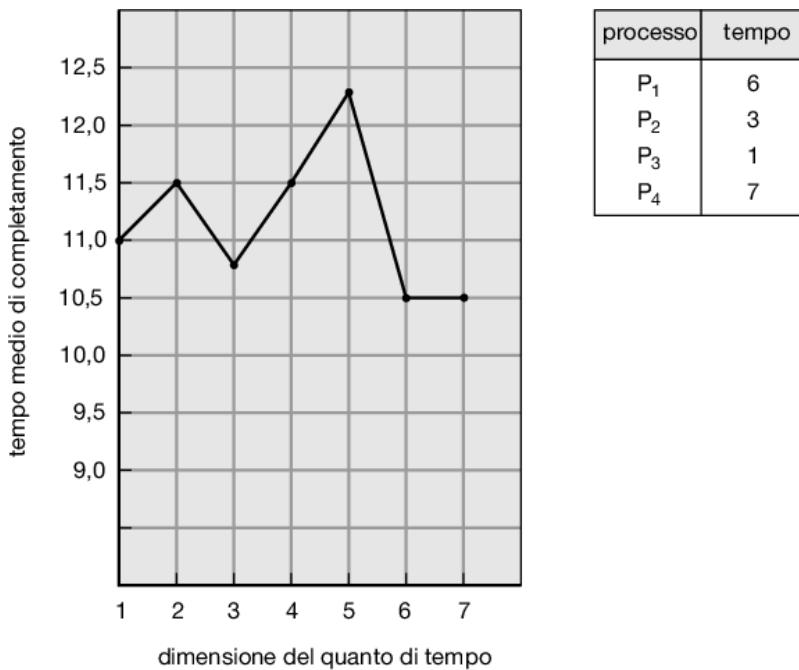


Figura 5.6 Variazione del tempo di completamento in funzione del quanto di tempo.

Benché il quanto di tempo debba essere grande in confronto al tempo necessario al cambio di contesto, non deve essere tuttavia troppo grande. Se il quanto di tempo è molto ampio, il criterio di scheduling rr, come puntualizzato in precedenza, tende al criterio fcfs. Empiricamente si può stabilire che l'80 per cento delle sequenze di operazioni della cpu debba essere più breve del quanto di tempo.

5.3.4 Scheduling con priorità

L'algoritmo sjf è un caso particolare del più generale algoritmo di scheduling con priorità: si associa una priorità a ogni processo e si assegna la cpu al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema fcfs. Un algoritmo sjf è semplicemente un algoritmo con priorità in cui la priorità (p) è l'inverso della lunghezza (prevista) della successiva sequenza di operazioni della cpu. A una maggiore lunghezza corrisponde una minore priorità, e viceversa.

Occorre notare che la discussione si svolge in termini di priorità *alta* e priorità *bassa*. Generalmente le priorità sono indicate da un intervallo fisso di numeri, come da 0 a 7, oppure da 0 a 4.095. Tuttavia, non c'è un orientamento comune sull'attribuire allo 0 la priorità più alta o quella più bassa; alcuni sistemi usano numeri bassi per rappresentare priorità basse, altri usano numeri bassi per priorità alte, il che può generare confusione. In questo testo i numeri bassi indicano priorità alte.

Come esempio, si consideri il seguente insieme di processi, che si suppone siano arrivati al tempo 0, nell'ordine $P_1, P_2 \dots P_5$, e dove la durata delle sequenze di operazioni della cpu è espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando lo scheduling con priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt.



Il tempo d'attesa medio è di 8,2 millisecondi.

Le priorità si possono definire sia *internamente* sia *esternamente*. Quelle definite internamente usano una o più quantità misurabili per calcolare la priorità del processo; per esempio sono stati utilizzati i limiti di tempo, i requisiti di memoria, il numero dei file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni di i/o e la lunghezza media delle sequenze di operazioni della cpu. Le priorità esterne si definiscono secondo criteri esterni al sistema operativo, come l'importanza del processo, il tipo e la quantità dei fondi pagati per l'uso del calcolatore, il dipartimento che promuove il lavoro e altri fattori, spesso di ordine politico.

Lo scheduling con priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla ready queue, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling con priorità con diritto di prelazione sottrae la cpu al processo attualmente in esecuzione se la priorità del processo appena arrivato è superiore. Un algoritmo senza prelazione si limita a porre l'ultimo processo arrivato alla testa della ready queue.

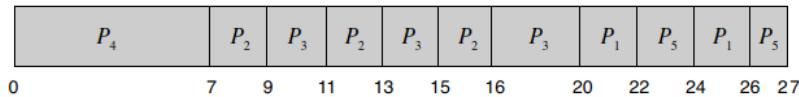
Un problema importante relativo agli algoritmi di scheduling con priorità è l'attesa indefinita (*starvation*). Un processo pronto per l'esecuzione, ma che non dispone della cpu, si può considerare bloccato nell'attesa della cpu. Un algoritmo di scheduling con priorità può lasciare processi a bassa priorità nell'attesa indefinita della cpu. In un sistema con carico elevato, un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla cpu. Generalmente accade che o il processo è eseguito, alle ore 2 del mattino della domenica, quando il sistema ha finalmente ridotto il proprio carico, oppure il calcolatore, prima o poi, va in crash e perde tutti i processi a bassa priorità non terminati. Corre voce che, quando fu fermato l'ibm 7094 al mit, nel 1973, si scoprì che un processo con bassa priorità sottoposto nel 1967 non era ancora stato eseguito.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'invecchiamento (*aging*); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo. Per esempio, se le priorità variano da 127 (bassa) a 0 (alta), si potrebbe incrementare di 1 ogni secondo il livello di priorità di un processo in attesa. Alla fine anche un processo con priorità iniziale 127 può ottenere la priorità massima nel sistema e quindi essere eseguito: un processo con priorità 127 impiegherebbe poco più di 2 minuti per raggiungere la priorità 0.

Un'altra opzione consiste nel combinare scheduling circolare e scheduling con priorità in modo tale che il sistema esegua il processo con la priorità più alta ed esegua processi con la stessa priorità utilizzando lo scheduling circolare. Si consideri per esempio il seguente insieme di processi, con la durata delle sequenze di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Utilizzando lo scheduling con priorità insieme allo scheduling circolare per processi con uguale priorità, con un quanto di tempo di 2 millisecondi, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt:



In questo esempio, il processo P_4 ha la priorità più alta e verrà quindi eseguito fino al completamento. Dopo P_4 , i processi P_2 e P_3 hanno la priorità maggiore e verranno eseguiti in modalità rr. Si noti che quando il processo P_2 termina, all'istante 16, il processo P_3 è il processo con la priorità più alta e verrà quindi eseguito fino al suo completamento. Restano poi i processi P_1 e P_5 che, avendo uguale priorità, verranno eseguiti in ordine circolare fino al completamento.

5.3.5 Scheduling a code multilivello

Nello scheduling con priorità e in quello circolare tutti i processi possono essere collocati in una singola coda da cui lo scheduler seleziona il processo con la priorità più alta da eseguire. A seconda di come vengono gestite le code può essere necessaria una ricerca $O(n)$ per determinare il processo con maggiore priorità. Nella pratica è spesso più semplice disporre di code separate per ciascuna priorità distinta e lasciare che lo scheduling con priorità si occupi semplicemente di selezionare il processo nella coda con priorità più alta, come illustrato nella Figura 5.7. Questo approccio, noto come scheduling a code multilivello, funziona bene anche quando lo scheduling con priorità è combinato con lo scheduling rr: se ci sono più processi nella coda con priorità più alta, questi vengono eseguiti in ordine circolare. Nella forma più generale di questo approccio viene assegnata una priorità statica a ciascun processo e un processo rimane nella stessa coda per tutto il suo tempo di esecuzione.

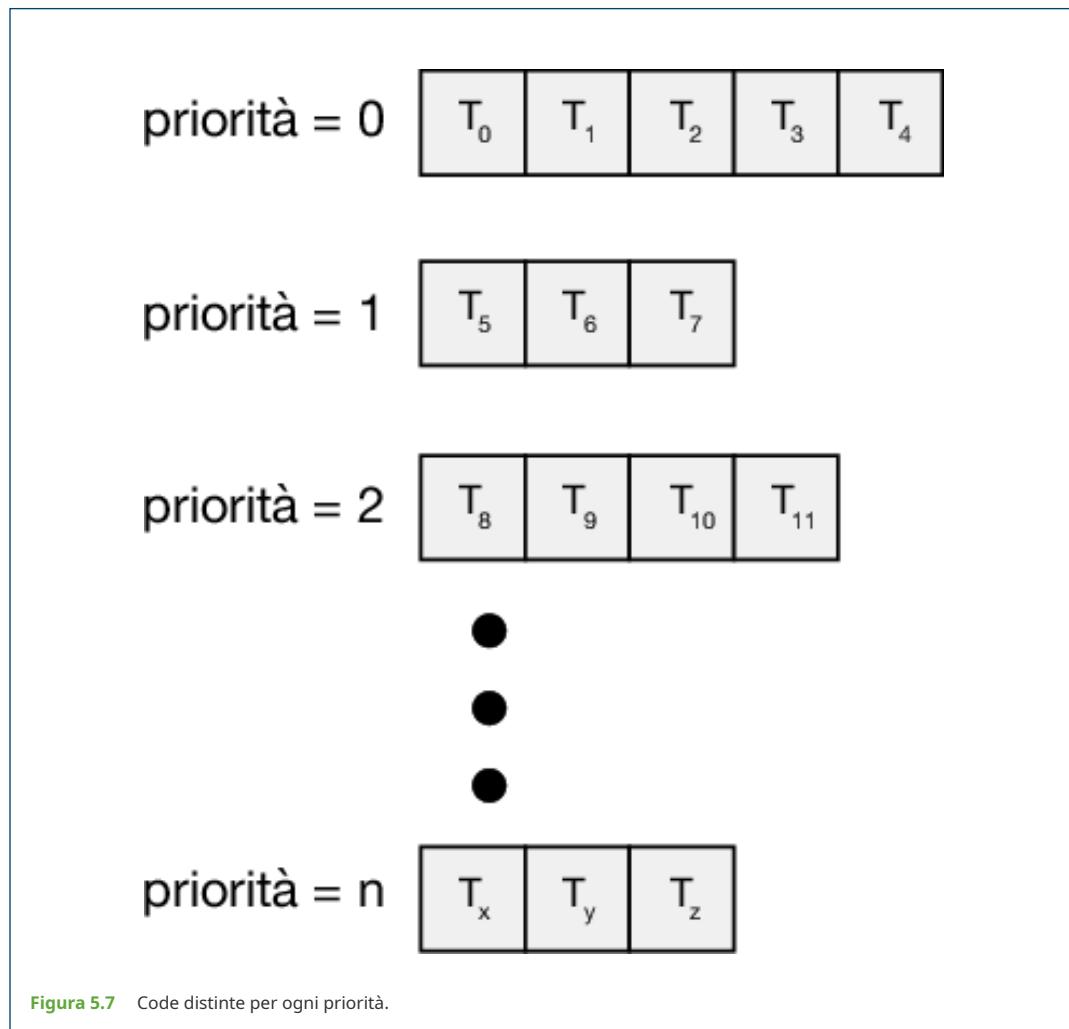
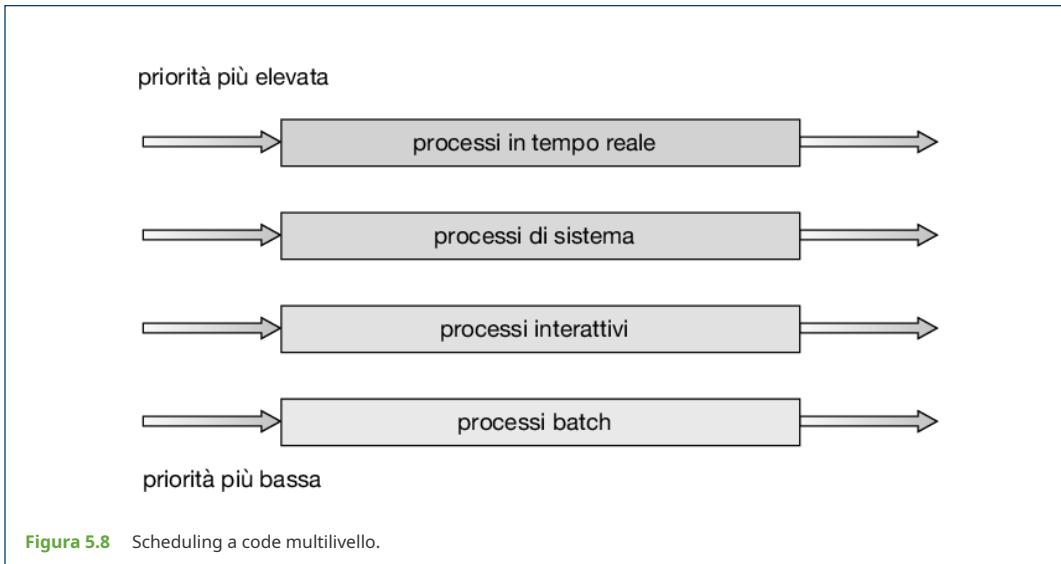


Figura 5.7 Code distinte per ogni priorità.

Un algoritmo di scheduling a code multilivello può anche essere utilizzato per suddividere i processi in diverse code in base al tipo di processo (Figura 5.8). Per esempio, di solito vengono divisi i processi in primo piano (interattivi) e i processi in background (batch). Questi due tipi di processi hanno requisiti diversi in termini di tempo di risposta e possono quindi avere esigenze di scheduling diverse. Inoltre, i processi in primo piano possono avere una priorità più alta (definita esternamente) rispetto ai processi in background. È possibile utilizzare code distinte per i processi in primo piano e in background e ciascuna coda potrebbe avere il proprio algoritmo di schedulazione. Per la coda in primo piano si potrebbe utilizzare, per esempio, un algoritmo rr, mentre lo scheduling della coda in background potrebbe utilizzare un algoritmo fcfs.



In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling a priorità fissa con prelazione. Per esempio, la coda dei processi in foreground (in primo piano) può avere la priorità assoluta sulla coda dei processi in background.

Si consideri come esempio il seguente algoritmo di scheduling a code multilivello, in ordine di priorità:

1. processi in tempo reale
2. processi di sistema
3. processi interattivi
4. processi batch

Ogni coda ha la priorità assoluta sulle code di priorità più bassa; per esempio nessun processo della coda dei processi batch può iniziare l'esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella ready queue durante l'esecuzione di un processo in background, si avrebbe la prelazione su quest'ultimo.

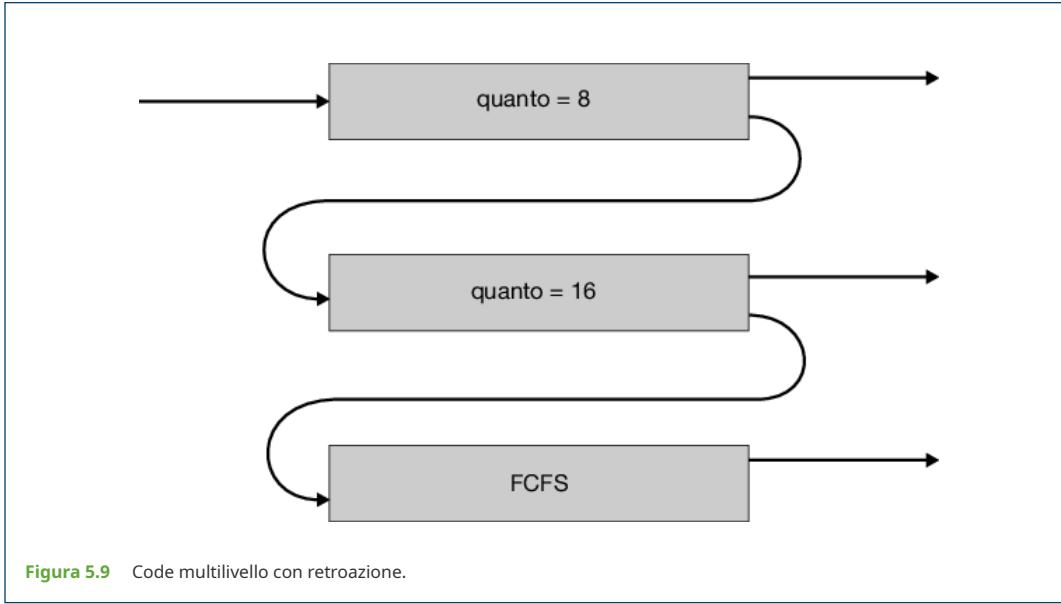
Un'altra possibilità è definire porzioni di tempo per le code. A ogni coda si assegna una parte del tempo d'elaborazione della cpu, suddivisibile a sua volta tra i processi che la costituiscono. Nel caso foreground/background, per esempio, si può assegnare l'80 per cento del tempo d'elaborazione della cpu alla coda dei processi in foreground (in primo piano), con scheduling rr tra i suoi processi; mentre per la coda dei processi in background si riserva il 20 per cento del tempo d'elaborazione della cpu, assegnato col criterio fcfs.

5.3.6 Scheduling a code multilivello con retroazione

Di solito in un algoritmo di scheduling a code multilivello i processi si assegnano in modo permanente a una coda all'entrata nel sistema, e non si possono spostare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in foreground e quelli che si eseguono in background, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in foreground o in background. Quest'impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

Lo scheduling a code multilivello con retroazione (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. L'idea consiste nel separare i processi che hanno caratteristiche diverse in termini di cpu burst. Se un processo usa troppo tempo di elaborazione della cpu, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi con prevalenza di i/o e i processi interattivi nelle code con priorità più elevata. Inoltre, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa. Questa forma di aging impedisce il verificarsi di un'attesa indefinita.

Si consideri, per esempio, uno scheduler a code multilivello con retroazione con tre code, numerate da 0 a 2, come nella Figura 5.9. Lo scheduler fa eseguire tutti i processi presenti nella coda 0; quando la coda 0 è vuota, si eseguono i processi nella coda 1; analogamente, i processi nella coda 2 si eseguono solo se le code 0 e 1 sono vuote. Un processo in ingresso nella coda 1 ha la prelazione sui processi della coda 2; un processo in ingresso nella coda 0, a sua volta, ha la prelazione sui processi della coda 1.



All'ingresso nella ready queue i processi vengono assegnati alla coda 0 e ottengono un quanto di tempo di 8 millisecondi; i processi che non terminano entro tale quanto di tempo, vengono spostati alla fine della coda 1. Se la coda 0 è vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio fcfs. Per evitare l'attesa indefinita, un processo in attesa da troppo tempo in una coda a priorità bassa viene gradualmente spostato in una coda con priorità più alta.

Questo algoritmo di scheduling dà la massima priorità ai processi con una sequenza di operazioni della cpu della durata di non più di 8 millisecondi. I processi di questo tipo ottengono rapidamente la cpu, terminano la propria sequenza di operazioni della cpu e passano alla successiva sequenza di operazioni di i/o; anche i processi che necessitano di più di 8 ma di non più di 24 millisecondi (coda 1) vengono serviti rapidamente, anche se con una priorità inferiore. I processi più lunghi finiscono automaticamente nella coda 2 e sono serviti secondo il criterio fcfs all'interno dei cicli di cpu lasciati liberi dai processi delle code 0 e 1.

Generalmente uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:

- numero di code;
- algoritmo di scheduling per ciascuna coda;
- metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

La definizione di uno scheduler a code multilivello con retroazione costituisce il più generale criterio di scheduling della cpu, che nella fase di progettazione si può adeguare a un sistema specifico. Sfortunatamente corrisponde anche all'algoritmo più complesso; la definizione dello scheduler migliore richiede infatti particolari metodi per la selezione dei valori dei diversi parametri.

5.4 Scheduling dei thread

Nel Capitolo 4 abbiamo arricchito il modello dei processi con i thread, distinguendo quelli *a livello utente* da quelli *a livello kernel*. Sui sistemi operativi che prevedono la loro presenza, il sistema non effettua lo scheduling dei processi, ma dei thread a livello kernel. I thread a livello utente sono gestiti da una libreria: il kernel non è consapevole della loro esistenza. Di conseguenza, per eseguire i thread a livello utente occorre associare loro dei thread a livello kernel. Tale associazione può essere indiretta, ossia realizzata con un processo leggero (lwp). Trattiamo adesso le questioni dello scheduling che riguardano i thread a livello utente e a livello kernel, offrendo esempi specifici dello scheduling per Pthreads.

5.4.1 Ambito della contesa

Una distinzione fra *thread a livello utente* e *a livello kernel* riguarda il modo in cui vengono schedulati. Nei sistemi che impiegano il modello da molti a uno (Paragrafo 4.3.1) e il modello da molti a molti (Paragrafo 4.3.3), la libreria dei thread pianifica l'esecuzione dei thread a livello utente su un lwp libero; si parla allora di ambito della contesa ristretto al processo (*process-contention scope*, pcs), perché la contesa per aggiudicarsi la cpu ha luogo fra thread dello stesso processo. In realtà, affermando che la libreria dei thread pianifica l'esecuzione dei thread a livello utente associandoli agli lwp liberi, non si intende che il thread sia in esecuzione su una cpu; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di thread del kernel su un processore fisico. Per determinare quale thread a livello kernel debba essere eseguito da una cpu, il kernel esamina i thread di tutto il sistema; si parla allora di ambito della contesa allargato al sistema (*system-contention scope*, scs). Quindi, nel caso di scs, tutti i thread del sistema competono per l'uso della cpu. I sistemi caratterizzati dal modello da uno a uno (Paragrafo 4.3.2) quali Windows e Linux, schedulano i thread unicamente sulla base di scs.

Nel caso del pcs, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta. Le priorità dei thread a livello utente sono stabilite dal programmatore, e la libreria dei thread non le modifica; alcune librerie danno facoltà al programmatore di cambiare la priorità di un thread. Si noti che quando l'ambito della contesa è ristretto al processo si è soliti applicare la prelazione al thread in esecuzione, a vantaggio di thread con priorità più alta; tuttavia, se i thread sono dotati della medesima priorità, non vi è garanzia di *time slicing* (porzione di tempo) (Paragrafo 5.3.3).

5.4.2 Scheduling di Pthread

La generazione dei thread con posix Pthreads è stata introdotta nel Paragrafo 4.4.1, insieme a un programma esemplificativo. Ci accingiamo ora a esaminare la api Pthread di posix, che consente di specificare pcs o scs nella fase di generazione dei thread. Per specificare l'ambito della contesa Pthreads usa i valori seguenti:

- `pthread_scope_process` pianifica i thread con lo scheduling pcs
- `pthread_scope_system` pianifica i thread tramite lo scheduling scs

Nei sistemi che si avvalgono del modello da molti a molti la politica `pthread_scope_process` pianifica i thread a livello utente sugli lwp disponibili. Il numero di lwp viene mantenuto dalla libreria dei thread, che può servirsi delle attivazioni dello scheduler (Paragrafo 4.6.5). La seconda politica, `pthread_scope_system`, crea, in corrispondenza di ciascun thread a livello utente, un lwp a esso vincolato, realizzando così, in effetti, una corrispondenza secondo il modello da molti a uno.

L'ipc di Pthread offre due funzioni per leggere (e impostare) l'ambito della contesa:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

Il primo parametro per entrambe le funzioni è un puntatore agli attributi del thread. Il secondo parametro della funzione `pthread_attr_setscope()` riceve uno dei valori `pthread_scope_system` o `pthread_scope_process`, che stabiliscono come deve essere impostato l'ambito della contesa. Nel caso di `pthread_attr_getscope()` il secondo parametro contiene un puntatore a un intero che contiene l'attuale valore dell'ambito della contesa. Qualora si verifichi un errore, ambedue le funzioni restituiscono valori non nulli.

Nella Figura 5.10 illustriamo l'api di scheduling Pthread mediante un programma che determina l'ambito della contesa in vigore e lo imposta a `pthread_scope_system`; quindi crea cinque thread distinti, che andranno in esecuzione secondo il modello di scheduling scs. Si noti che, nel caso di alcuni sistemi, sono possibili solo determinati valori per l'ambito della contesa. I sistemi Linux e macos, per esempio, consentono soltanto `pthread_scope_system`.

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
```

```

pthread_t tid[NUM_THREADS];

pthread_attr_t attr;

/* ottiene gli attributi di default */

pthread_attr_init(&attr);

/* per prima cosa appura l'ambito della contesa */

if (pthread_attr_getscope(&attr, &scope) != 0)

    fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");

else {

    if (scope == PTHREAD_SCOPE_PROCESS)

        printf("PTHREAD_SCOPE_PROCESS");

    else if (scope == PTHREAD_SCOPE_SYSTEM)

        printf("PTHREAD_SCOPE_SYSTEM");

    else

        fprintf(stderr, "Valore d'ambito della contesa non ammesso.\n");

}

/* imposta l'algoritmo di scheduling a PCS o SCS */

pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

/* genera i thread */

for (i = 0; i < NUM_THREADS; i++)

    pthread_create(&tid[i], &attr, runner, NULL);

/* adesso aspetta la terminazione di tutti i thread */

for (i = 0; i < NUM_THREADS; i++)

    pthread_join(tid[i], NULL);

}

/* ogni thread inizia l'esecuzione da questa funzione */

void *runner(void *param)

{

    /* fai qualcosa ... */

    pthread_exit(0);

}

```

Figura 5.10 api di scheduling Pthread.

5.5 Scheduling per sistemi multiprocessore

Fin qui la trattazione ha riguardato lo scheduling della cpu in un sistema dotato di un singolo core di elaborazione. Se sono disponibili più unità d'elaborazione è possibile la distribuzione del carico (*load sharing*), ma il problema dello scheduling diviene proporzionalmente più complesso. Si sono sperimentate diverse possibilità e, come s'è visto nella trattazione dello scheduling di una sola cpu, “la soluzione migliore” non esiste.

Tradizionalmente, il termine multiprocessore faceva riferimento a sistemi che fornivano più processori fisici, in cui ogni processore conteneva una cpu single-core. Tuttavia, la definizione di multiprocessore si è evoluta in modo significativo e sui moderni sistemi di elaborazione il termine multiprocessore si applica ora alle seguenti architetture di sistema:

- cpu multicore.
- Core multithread.
- Sistemi numa.
- Sistemi multiprocessore eterogenei.

Nel seguito si analizzano brevemente alcuni problemi attinenti lo scheduling per sistemi multiprocessore nel contesto di queste diverse architetture. Nei primi tre casi ci concentriamo su sistemi in cui i processori sono identici – omogenei – in termini di funzionalità: si può quindi utilizzare qualsiasi cpu disponibile per eseguire qualsiasi processo presente nella coda. Nell'ultimo caso analizziamo un sistema in cui i processori non hanno le stesse capacità.

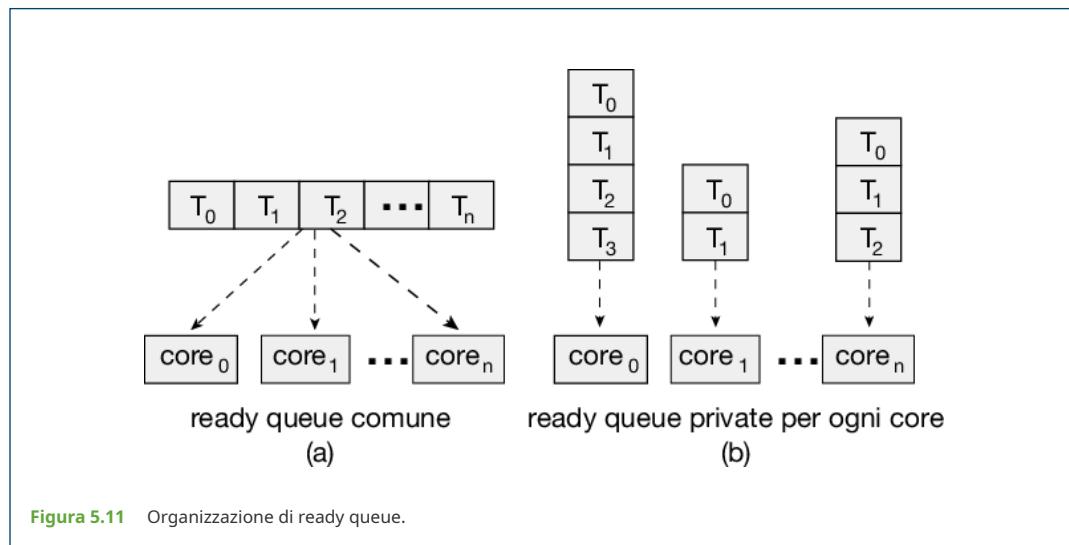
5.5.1 Approcci allo scheduling per multiprocessori

Una prima strategia di scheduling della cpu per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell' i/o e altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice utente. Si tratta della multielaborazione asimmetrica, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema. Lo svantaggio di questo approccio è che il master server costituisce un potenziale collo di bottiglia in grado di ridurre le prestazioni generali del sistema.

L'approccio standard per supportare i multiprocessori è la multielaborazione simmetrica (smp), in cui ciascun processore è in grado di autogestirsi. Lo scheduling viene realizzato facendo in modo che lo scheduler di ciascun processore esamina la ready queue e selezioni un thread da eseguire. Si noti che questo approccio offre due possibili strategie per organizzare i thread da selezionare per l'esecuzione:

1. tutti i thread possono trovarsi in una ready queue comune;
2. ogni processore può avere una propria coda privata di thread.

La Figura 5.11 mostra un confronto tra le due strategie. Se viene utilizzata la prima opzione è possibile avere race condition sulla coda condivisa e occorre quindi assicurarsi che due processori distinti non scelgano di eseguire lo stesso thread e che i thread nella coda non vengano persi. Come discusso nel Capitolo 6 è possibile usare una forma di lock per proteggere la ready queue comune da questa race condition. Il lock sarebbe tuttavia molto conteso, poiché tutti gli accessi alla coda richiederebbero il possesso del lock, e l'accesso alla coda condivisa costituirebbe probabilmente un collo di bottiglia per le prestazioni. La seconda opzione permette a ciascun processore di schedulare i thread da una coda di esecuzione privata e pertanto non risente dei possibili problemi di prestazioni associati a una coda condivisa. Per questa ragione si tratta dell'approccio più comune sui sistemi che supportano smp. Inoltre, come descritto nel Paragrafo 5.5.4, l'uso di code di esecuzione private, una per ogni processore, può portare a un uso più efficiente della memoria cache. Vi sono anche dei problemi nell'utilizzo di code distinte per ogni processore, in particolare relativi ai diversi carichi di lavoro. Tuttavia, come vedremo, possono essere utilizzati algoritmi di bilanciamento per distribuire equamente i carichi di lavoro tra tutti i processori.

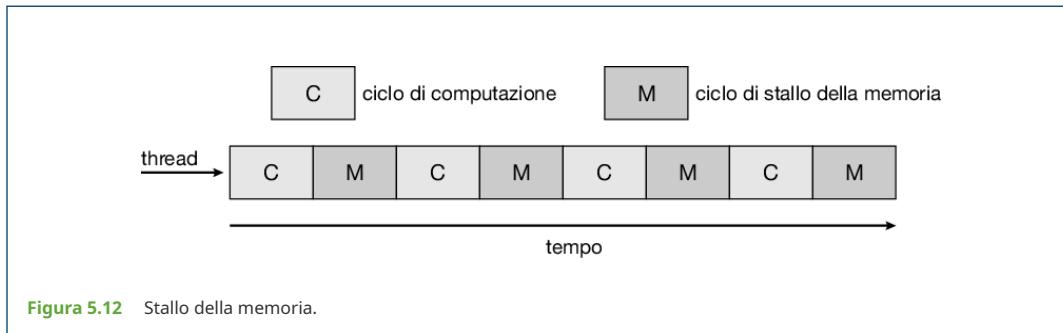


Praticamente tutti i sistemi operativi moderni supportano smp, inclusi Windows, Linux, macos e i sistemi mobili Android e ios. Nel resto di questo paragrafo discuteremo alcune questioni concernenti i sistemi smp, relativamente alla progettazione degli algoritmi di scheduling della cpu.

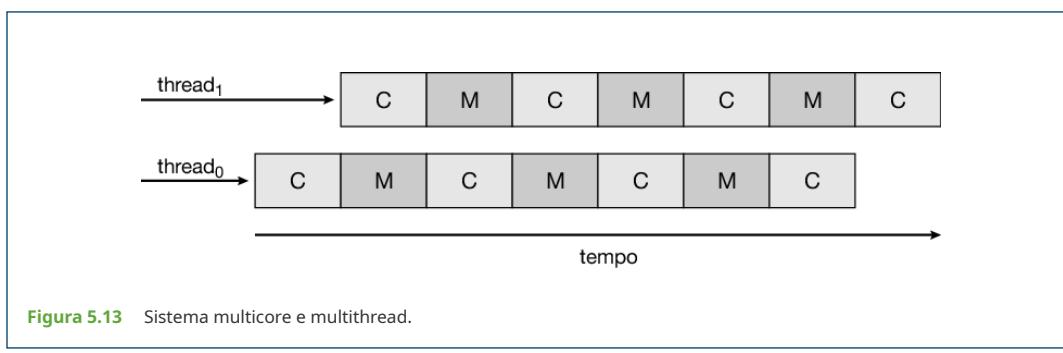
5.5.2 Processori multicore

Tradizionalmente i sistemi smp hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la pratica recente nel progetto hardware dei calcolatori è di inserire più core di elaborazione in un unico chip fisico, dando origine a un processore multicore. Ogni core mantiene il proprio stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi smp che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling. Proviamo a vedere che cosa può succedere. Le ricerche hanno permesso di scoprire che quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come stallo della memoria, può verificarsi per varie ragioni, per esempio la mancanza dei dati richiesti nella cache. La Figura 5.12 mostra uno stallo della memoria. In questo scenario, il processore può trascorrere fino al 50 per cento del suo tempo attendendo che i dati siano disponibili in memoria.



Per rimediare a questa situazione, molti dei progetti hardware recenti implementano delle unità di calcolo multithread in cui due o più thread hardware sono assegnati a un singolo core. In questo modo, se un thread è in situazione di stallo in attesa della memoria, il core può passare a eseguire un altro thread. La Figura 5.13 mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono avvicinate nel tempo. Dal punto di vista del sistema operativo, ogni thread hardware mantiene il suo stato architettonale, come il puntatore alle istruzioni e il set di registri, e appare quindi come una cpu logica che è disponibile per eseguire un thread software. Questa tecnica, nota come chip multithreading (cmt), è illustrata nella Figura 5.14, dove è mostrato un processore contenente quattro core di elaborazione, ognuno dei quali contiene due thread hardware: dal punto di vista del sistema operativo sono presenti otto cpu logiche.



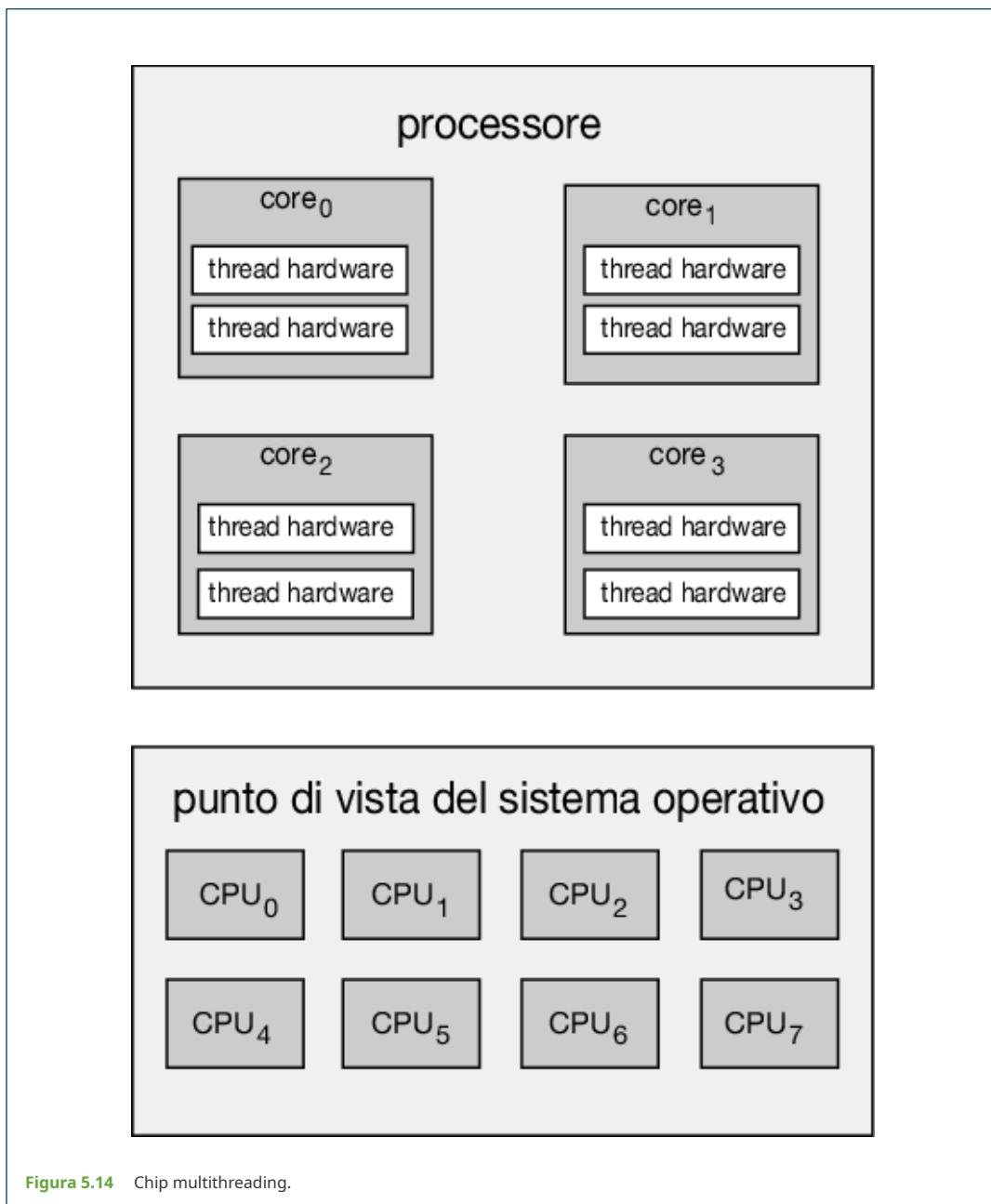


Figura 5.14 Chip multithreading.

I processori Intel utilizzano il termine hyper-threading (noto anche come *multithreading simultaneo* o smt) per descrivere l'assegnazione di più thread hardware a un singolo core di elaborazione. I processori Intel contemporanei, come l'i7, supportano due thread per core, mentre il processore Oracle Sparc M7 supporta otto thread per core, con otto core per processore, fornendo così al sistema operativo 64 cpu logiche.

In generale, ci sono due modi per rendere un core multithread: attraverso il multithreading a grana grossa (*coarse-grained*) o il multithreading a grana fine (*fine-grained*). Nel multithreading coarse-grained un thread resta in esecuzione su un processore fino al verificarsi di un evento a lunga latenza, per esempio uno stall di memoria. A causa dell'attesa introdotta dall'evento a lunga latenza, il processore deve passare a un altro thread e iniziare a eseguirlo. Tuttavia, il costo per cambiare il thread in esecuzione è alto, perché occorre ripulire la pipeline delle istruzioni prima che il nuovo thread possa iniziare a essere eseguito sull'unità di calcolo. Una volta che il nuovo thread è in esecuzione inizia a riempire la pipeline con le sue istruzioni. Il multithreading fine (anche detto *interleaved multithreading*) passa da un thread a un altro a un livello molto più fine di granularità (tipicamente al termine di un ciclo di istruzione). Tuttavia, il progetto di sistemi a multithreading fine include una logica dedicata al cambio di thread. Ne risulta così che il costo del passaggio da un thread a un altro è piuttosto basso.

E importante notare che le risorse del core fisico (come cache e pipeline) devono essere condivise tra i suoi thread hardware e dunque un core di elaborazione può eseguire solo un thread hardware alla volta. Di conseguenza, un processore multithreaded e multicore richiede due diversi livelli di scheduling, come mostrato nella Figura 5.15, che illustra un core di elaborazione dual-threaded.

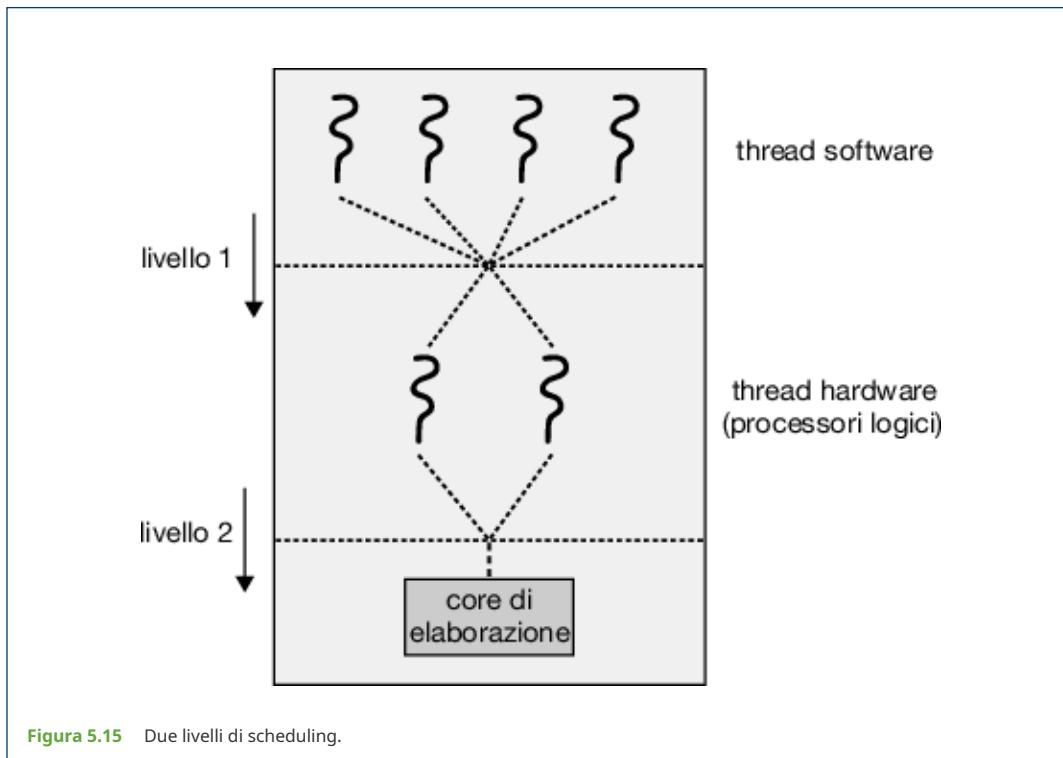


Figura 5.15 Due livelli di scheduling.

A un livello vi sono le decisioni di scheduling che devono essere prese dal sistema operativo per scegliere quale thread software eseguire su ogni thread hardware (cpu logica). Ai fini pratici, tali decisioni sono state l'obiettivo principale di questo capitolo e per questo livello di scheduling il sistema operativo può scegliere qualsiasi algoritmo, compresi quelli descritti nel Paragrafo 5.3.

Un secondo livello di scheduling specifica in che modo ciascun core decide quale thread hardware eseguire. In questa situazione è possibile adottare diverse strategie. Un approccio consiste nell'utilizzare un semplice algoritmo round-robin per assegnare un thread hardware al core di elaborazione: questo è l'approccio adottato da Ultrasparc T3. Intel Itanium, un processore dual-core con due thread hardware per core, adotta un approccio differente. In questo caso viene assegnato a ciascun thread hardware un indice dinamico di urgenza (*urgency*) compreso tra 0 e 7, dove 0 rappresenta l'urgenza più bassa e 7 la più alta. Itanium identifica cinque diversi eventi in grado di attivare un cambio di thread. Quando si verifica uno di questi eventi, la logica di cambio di thread confronta l'urgenza dei due thread e seleziona il thread con il valore di urgenza più alto da eseguire sul core del processore.

Si noti che i due diversi livelli di scheduling mostrati nella Figura 5.15 non sono necessariamente mutuamente esclusivi. Infatti, se lo scheduler del sistema operativo (il primo livello) viene reso consapevole della condivisione delle risorse del processore, può prendere decisioni di scheduling più efficaci. Per esempio, supponiamo che una cpu abbia due core di elaborazione e che ogni core abbia due thread hardware. Se due thread software sono in esecuzione su questo sistema, possono essere in esecuzione sullo stesso core o su core separati. Se entrambi vengono assegnati allo stesso core, devono condividere le risorse del processore e quindi è probabile che procedano più lentamente rispetto al caso in cui siano assegnati a core distinti. Se il sistema operativo è a conoscenza del livello di condivisione delle risorse del processore può programmare i thread software su processori logici che non condividono risorse.

5.5.3 Bilanciamento del carico

Sui sistemi smp è importante che il carico di lavoro sia distribuito equamente tra tutte le unità elaborate per sfruttare appieno i vantaggi di avere più processori. Se ciò non avvenisse, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati con una coda di processi in attesa. Il bilanciamento del carico tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema smp. Bisogna notare che il bilanciamento è necessario, di norma, solo nei sistemi in cui ciascun processore ha una coda privata di processi eseguibili. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è sovente superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune dei processi eseguibili.

Il bilanciamento del carico può seguire due approcci: la migrazione push e la migrazione pull. La prima prevede che un processo apposito controlli periodicamente il carico di ogni processore: se identifica uno sbilanciamento, riporta il carico in equilibrio,

spostando i processi dal processore saturo ad altri più liberi, o inattivi. La migrazione pull, invece, si ha quando un processore inattivo prende un processo in attesa a un processore sovraccarico. I due tipi di migrazione non sono mutuamente esclusivi, e trovano spesso applicazione contemporanea nei sistemi con bilanciamento del carico. Lo scheduler cfs di Linux, per esempio, che vedremo nel Paragrafo 5.7.1, e lo scheduler ule dei sistemi Freebsd, si avvalgono di entrambe le tecniche.

Il concetto di “carico bilanciato” può avere significati diversi. Una prima possibilità è richiedere semplicemente che tutte le code contengano approssimativamente lo stesso numero di thread. In alternativa, il bilanciamento potrebbe richiedere un’equa distribuzione delle priorità dei thread su tutte le code. Inoltre, in determinate situazioni, potrebbe non essere sufficiente alcuna di queste strategie, poiché esse possono essere in contrasto con gli obiettivi dell’algoritmo di scheduling. (Lasciamo ulteriori considerazioni su questo argomento come esercizio).

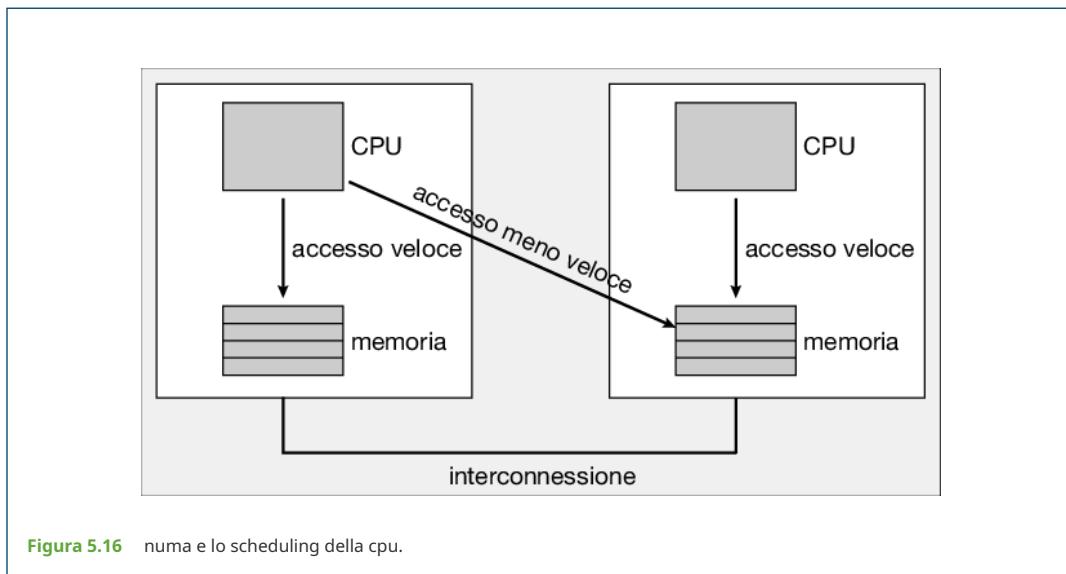
5.5.4 Predilezione per il processore

Si consideri che cosa accade alla memoria cache dopo che un processo è stato eseguito da uno specifico processore: i dati che il processore ha trattato più recentemente permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la memoria cache (chiamata warm cache). Si consideri ora che cosa succede se un processo si sposta su un altro processore: i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita. A causa degli alti costi di svuotamento e riempimento della cache, molti sistemi smp tentano di impedire il passaggio di thread da un processore all’altro, mirando a mantenere un thread sempre sullo stesso processore e a sfruttare i vantaggi della warm cache. Si parla di predilezione per il processore (*processor affinity*), intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione.

Le due strategie per l’organizzazione della coda dei thread disponibili per lo scheduling descritte nel Paragrafo 5.5.1 hanno implicazioni sulla predilezione per il processore. Infatti, se adottiamo l’approccio di una ready queue comune, un thread può essere selezionato per l’esecuzione da qualsiasi processore e ogni volta che un thread è schedulato su un nuovo processore la cache del processore deve essere ripopolata. Con le ready queue private, distinte per ogni processore, un thread viene sempre schedulato sullo stesso processore e può quindi trarre vantaggio dal contenuto di una warm cache. In sostanza, le ready queue private forniscono la predilezione del processore gratuitamente!

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di predilezione debole (*soft affinity*). In questo caso il sistema operativo tenta di mantenere il processo su un singolo processore, ma è possibile che un processo migri da un processore all’altro. Alcuni sistemi dispongono di chiamate di sistema per realizzare la predilezione forte (*hard affinity*), permettendo così a un processo di specificare un sottoinsieme di processori su cui può essere eseguito. Molti sistemi supportano sia la predilezione debole sia la predilezione forte. Linux, per esempio, implementa la predilezione debole, ma fornisce anche la chiamata `sched_setaffinity()` per il supporto della predilezione forte, consentendo così a un thread di specificare l’insieme di cpu su cui può essere eseguito.

Anche l’architettura della memoria principale di un sistema può influenzare le questioni relative alla predilezione. La Figura 5.16 mostra un’architettura con accesso non uniforme alla memoria (numa) in cui sono presenti due chip fisici di processore, ciascuno con la propria cpu e memoria locale. Sebbene un’interconnessione di sistema consenta a tutte le cpu in un sistema numa di condividere uno spazio di indirizzamento fisico, una cpu ha un accesso più rapido alla propria memoria locale rispetto alla memoria locale di un’altra cpu. Se lo scheduler della cpu del sistema operativo e gli algoritmi di gestione della memoria sono *consci del numa* e lavorano insieme, a un thread che è stato schedulato su una particolare cpu può essere allocata la memoria più vicina a dove risiede la cpu, fornendo così al thread un accesso alla memoria più veloce possibile.



E interessante notare che il bilanciamento del carico spesso contrasta i vantaggi della predilezione per il processore. Il vantaggio di mantenere un thread in esecuzione sullo stesso processore, infatti, è che il thread può sfruttare i suoi dati nella memoria cache di quel processore. Il bilanciamento del carico, spostando un thread da un processore a un altro, rimuove questo vantaggio. Allo stesso modo, la migrazione di un thread tra processori può comportare svantaggi su sistemi numa, in cui un thread può essere spostato su un processore che richiede tempi di accesso alla memoria più lunghi. In altre parole, esiste un conflitto naturale tra il bilanciamento del carico e la riduzione dei tempi di accesso alla memoria. Per questa ragione gli algoritmi di scheduling per i moderni sistemi numa multicore sono diventati piuttosto complessi. Nel Paragrafo 5.7.1 si esamina l'algoritmo di scheduling cfs di Linux, analizzando come questo algoritmo bilanci questi obiettivi in conflitto tra loro.

5.5.5 Multiprocessing eterogeneo

Negli esempi che abbiamo discusso finora tutti i processori sono identici in termini di funzionalità e consentono quindi a qualsiasi thread di essere eseguito su qualsiasi core di elaborazione. L'unica differenza è che i tempi di accesso alla memoria variano in base al bilanciamento del carico e alle politiche di predilezione per il processore, nonché sui sistemi numa.

Sebbene i sistemi mobili includano architetture multicore, alcuni sistemi attuali sono progettati utilizzando core che eseguono lo stesso set di istruzioni, ma differiscono in termini di velocità di clock e gestione energetica, includendo la possibilità di regolare il consumo energetico di un core fino a renderlo inattivo (idle). Tali sistemi sono chiamati sistemi a multielaborazione eterogenea o, più brevemente, sistemi hmp (*heterogeneous multiprocessing*). Si noti che questa non è una forma di multielaborazione asimmetrica come quella descritta nel Paragrafo 5.5.1, poiché sia le attività di sistema sia quelle dell'utente possono essere eseguite su qualsiasi core. Piuttosto, l'intenzione dietro hmp è quella di gestire meglio il consumo di energia assegnando un task a un determinato core in base alle sue specifiche richieste.

Nei processori arm che supportano hmp questa tipologia di architettura è nota come big.little ed è costituita da grandi core con prestazioni più elevate (big) combinati con piccoli core ad alta efficienza energetica (little). I core grandi consumano più energia e dovrebbero quindi essere usati solo per brevi periodi, mentre i core piccoli consumano meno energia e possono quindi essere utilizzati per periodi più lunghi.

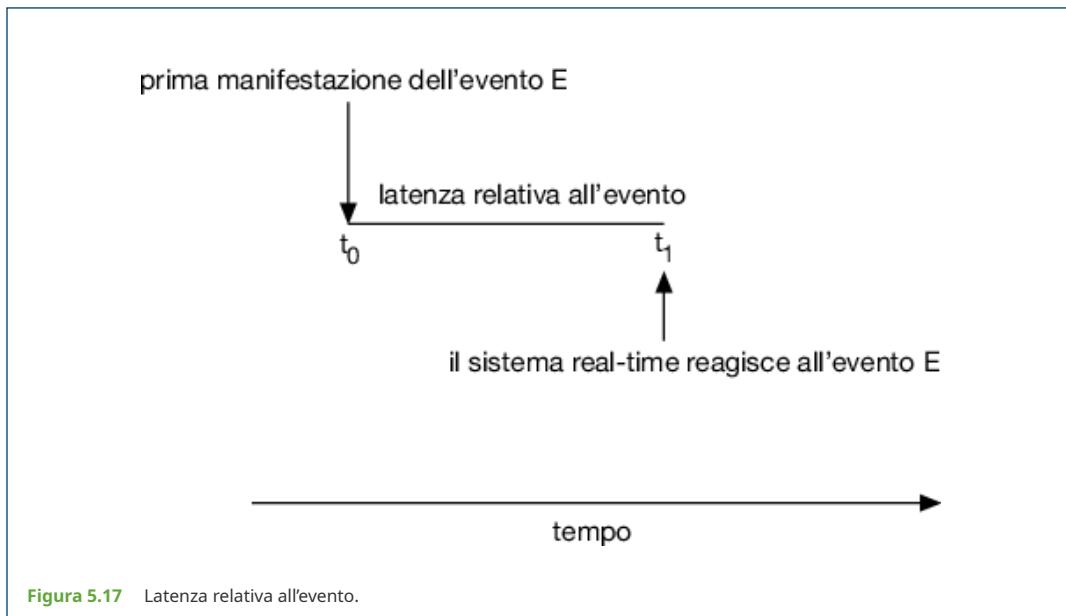
Questo approccio offre molti vantaggi. Grazie alla combinazione di un certo numero di core più lenti con core più veloci, uno scheduler della cpu può assegnare task che non richiedono prestazioni elevate, ma che potenzialmente saranno eseguiti per periodi più lunghi (come le attività in background) sui core piccoli, contribuendo così a preservare la carica della batteria. Allo stesso modo, le applicazioni interattive che richiedono più potenza di elaborazione, ma restano in esecuzione per periodi più brevi, possono essere assegnate ai core grandi. Inoltre, se il dispositivo mobile si trova in una modalità di risparmio energetico, è possibile disabilitare i core grandi ad alto consumo e forzare il sistema a utilizzare solo i core piccoli a basso consumo. Windows 10 supporta lo scheduling hmp, consentendo a un thread di selezionare una politica di scheduling ottimale secondo le sue esigenze di risparmio energetico.

5.6 Scheduling real-time della CPU

Lo scheduling della cpu nei sistemi real-time ha peculiarità proprie. In generale, possiamo distinguere tra sistemi real-time soft e sistemi real-time hard. I sistemi real-time soft non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano solamente che sarà data precedenza a quest'ultimo piuttosto che ad altri processi non critici. I sistemi real-time hard hanno vincoli più rigidi: i task vanno eseguiti entro una scadenza prefissata ed eseguirli dopo tale scadenza è del tutto inutile. In questo paragrafo analizziamo diversi aspetti relativi allo scheduling nei sistemi real-time soft e hard.

5.6.1 Minimizzazione della latenza

I sistemi real-time sono per loro natura guidati dagli eventi: generalmente, il sistema attende che si verifichi un evento in tempo reale. Quest'ultimo può avere luogo a livello software – come quando scatta un timer – o hardware – come quando un veicolo controllato a distanza si accorge di essere vicino a un ostacolo. In presenza di un evento, il sistema deve rispondere con la massima velocità possibile. Chiameremo latenza relativa all'evento (*event latency*) il periodo di tempo che intercorre tra l'occorrenza dell'evento e il momento in cui il sistema ne effettua la gestione (Figura 5.17).

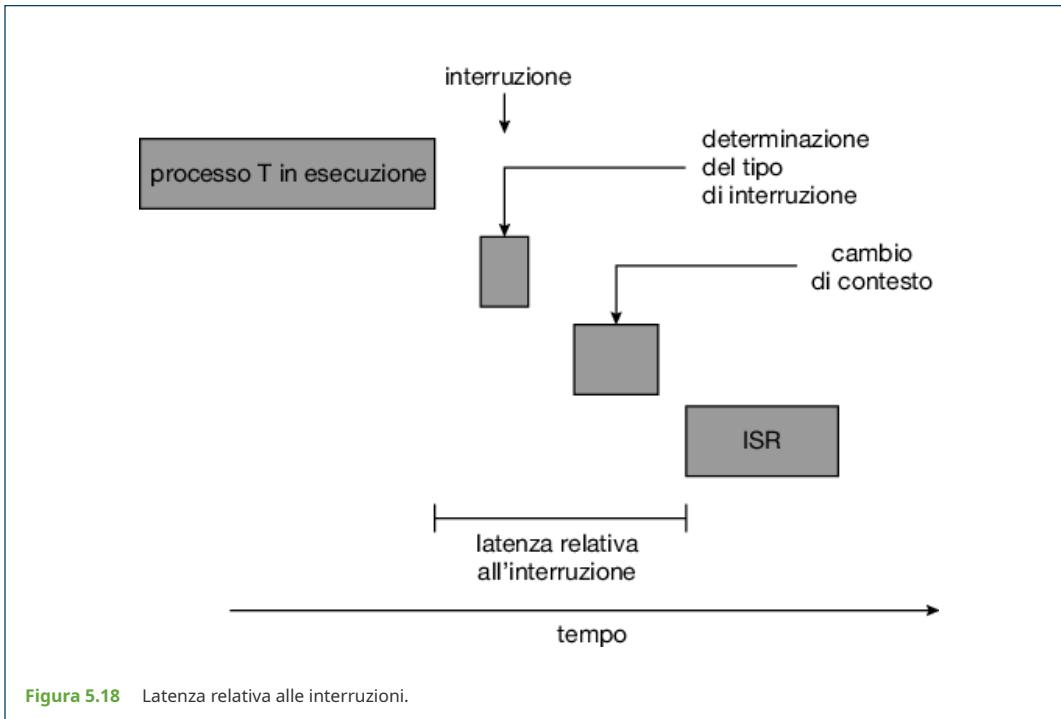


In genere, eventi diversi hanno diversi requisiti di latenza. Per esempio, la latenza per un sistema antiblocco dei freni potrebbe essere da tre a cinque millisecondi; ciò significa che dal momento in cui una ruota avverte che sta scivolando, il meccanismo che controlla il blocco dei freni ha da tre a cinque millesimi di secondo per reagire e controllare la situazione. Una risposta più lenta potrebbe causare lo sbandamento e la perdita di controllo del veicolo. Un sistema integrato di controllo del radar in un aeroplano passeggeri, invece, potrebbe tollerare una latenza di alcuni secondi.

Le categorie di latenza che influiscono sul funzionamento dei sistemi real-time sono due:

1. latenza relativa alle interruzioni;
2. latenza relativa al dispatch.

La latenza relativa all'interruzione si riferisce al periodo di tempo compreso tra la notifica di un'interruzione alla cpu e l'avvio della routine che gestisce l'interruzione. Quando sopraggiunge un'interruzione, il sistema operativo deve in primo luogo portare a compimento l'istruzione che sta eseguendo, e determinare di quale tipo di interruzione si tratti. Successivamente deve salvare lo stato del processo che è in atto, prima di occuparsi dell'interruzione tramite l'apposita procedura di gestione (ist). Il tempo complessivamente impiegato per svolgere questi task è definito come la latenza relativa all'interruzione (Figura 5.18).

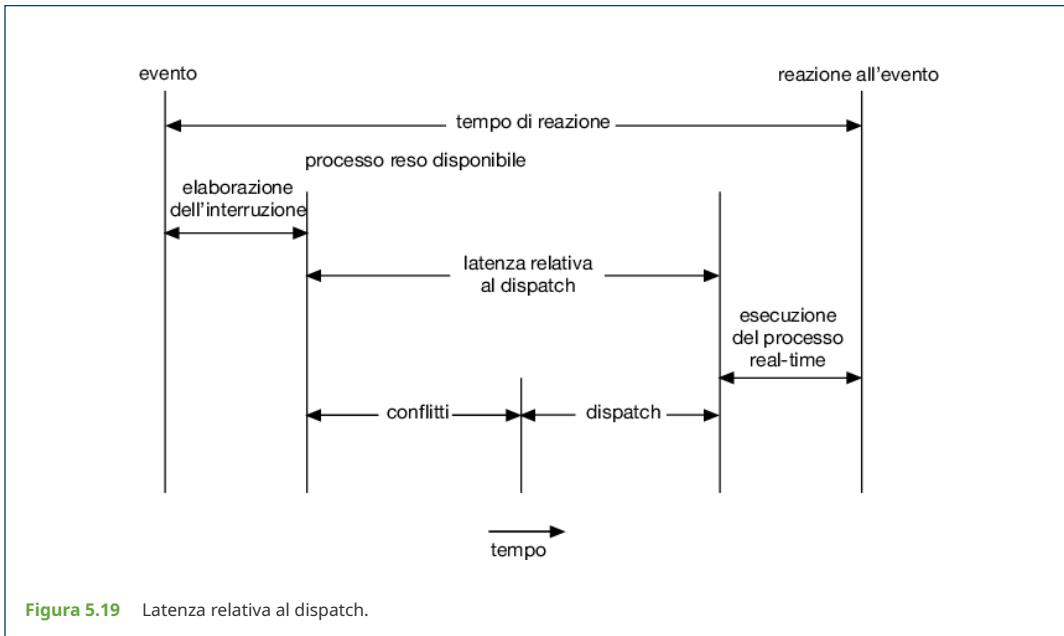


Evidentemente, per garantire l'immediata presa in consegna delle operazioni real-time è cruciale che un sistema real-time riduca al minimo la latenza relativa alle interruzioni. Nei sistemi real-time hard la latenza relativa alle interruzioni non deve essere semplicemente minimizzata, ma deve rispettare un limite superiore, per rispondere alle rigide esigenze di questo tipo di sistemi.

Un fattore importante che incide sulla latenza relativa alle interruzioni è l'intervallo di tempo in cui le interruzioni sono disattivate durante l'aggiornamento delle strutture dati del kernel. I sistemi operativi real-time esigono che le interruzioni siano inibite soltanto per intervalli di tempo molto piccoli.

Il periodo di tempo necessario al dispatcher per bloccare un processo e avviare un altro è noto come latenza di dispatch. Per garantire ai processi real-time l'accesso immediato alla cpu è necessario che i sistemi operativi minimizzino anche questo tipo di latenza. La tecnica più efficace per mantenere bassa la latenza relativa al dispatch consiste nell'implementare kernel con prelazione. Nei sistemi hard real-time, la latenza di dispatch si misura generalmente nell'ordine dei microsecondi.

Il grafico nella Figura 5.19 mostra la composizione della latenza relativa al dispatch. La *fase di conflitto* della latenza di dispatch consiste di due componenti:



1. prelazione di ogni processo in esecuzione nel kernel;
2. cessione, da parte dei processi a bassa priorità, delle risorse richieste dal processo ad alta priorità.

Dopo la fase di conflitto, la fase di dispatch assegna il processo ad alta priorità a una cpu disponibile.

5.6.2 Scheduling basato sulla priorità

La caratteristica più importante di un sistema operativo real-time è di rispondere immediatamente a un processo in tempo reale, non appena questo processo richiede la cpu. Lo scheduler di un sistema operativo real-time deve dunque utilizzare un algoritmo con prelazione basato su priorità. Ricordiamo che gli algoritmi di scheduling con priorità assegnano a ogni processo una priorità in base alla sua importanza; ai task più importanti vengono assegnate priorità più elevate rispetto a quelle assegnate ai processi ritenuti meno importanti. Se lo scheduler supporta anche la prelazione, un processo in esecuzione sulla cpu viene interrotto se diventa disponibile per l'esecuzione un processo con priorità più alta.

Gli algoritmi di scheduling con prelazione basati sulla priorità sono discussi in dettaglio nel Paragrafo 5.3.4 e il Paragrafo 5.7 presenta esempi delle funzionalità di scheduling real-time soft nei sistemi operativi Linux, Windows e Solaris. Tutti questi sistemi assegnano ai processi in tempo reale la massima priorità. Per esempio, Windows ha 32 livelli di priorità diversi. I livelli più alti, con valori di priorità da 16 a 31, sono riservati ai processi real-time. Solaris e Linux hanno schemi di priorità simili.

Si noti che uno scheduler con prelazione basata sulla priorità garantisce solo la funzionalità real-time soft. I sistemi real-time hard devono anche garantire che le attività in tempo reale saranno servite rispettando le scadenze. Per soddisfare a questo requisito lo scheduler deve avere funzionalità addizionali. Nel resto di questo paragrafo ci occupiamo di algoritmi di scheduling per sistemi real-time hard.

Prima di procedere con i dettagli dei singoli scheduler, dobbiamo tuttavia definire alcune caratteristiche dei processi che devono essere schedulati.

Innanzitutto, i processi sono considerati periodici, nel senso che richiedono la cpu a intervalli costanti di tempo (periodi). Ciascun processo periodico, una volta avuto accesso alla cpu, ha un tempo di elaborazione fisso t , una scadenza d entro cui la cpu deve completare la sua esecuzione e un periodo p . La relazione tra il tempo di elaborazione, la scadenza e il periodo si può riassumere nelle diseguaglianze $0 \leq t \leq d \leq p$. La frequenza di un processo periodico è data da $1/p$. La Figura 5.20 mostra l'esecuzione di un processo periodico nel corso del tempo. Lo scheduler può trarre vantaggio da queste caratteristiche, assegnando le priorità in base alla scadenza o ai requisiti di frequenza di un processo.

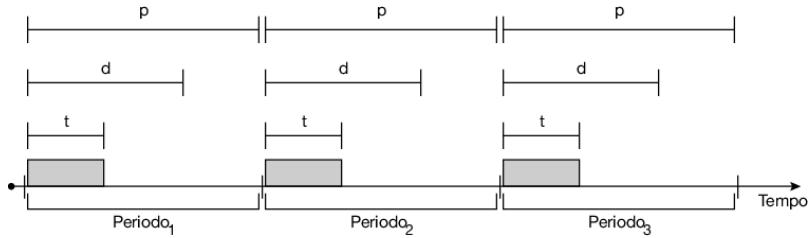


Figura 5.20 Processo periodico.

L'aspetto inusuale di questo tipo di scheduling è il fatto che un processo può dover dichiarare allo scheduler la propria scadenza d . Sulla base di questa informazione, grazie a una tecnica nota come algoritmo di controllo dell'ammissione, lo scheduler decide se accettare il processo – con la garanzia di eseguirne le richieste in tempo – o rifiutarlo, qualora non sia certo di poterne soddisfare le richieste entro la scadenza relativa.

5.6.3 Scheduling con priorità proporzionale alla frequenza

L'algoritmo di scheduling con priorità proporzionale alla frequenza programma i task periodici applicando un modello statico di attribuzione delle priorità con prelazione. Se un processo con priorità elevata diventa pronto per l'esecuzione mentre è in esecuzione un processo con priorità più bassa, il primo avrà diritto di prelazione. Entrando nel sistema, ciascun task periodico si vede assegnare una priorità inversamente proporzionale al proprio periodo: più breve è il periodo, più alta la priorità; più lungo il periodo, più bassa la priorità. La logica di questa regola consiste nell'assegnare priorità più elevate ai processi che fanno uso più frequente della cpu. Per più, lo scheduling con priorità proporzionale alla frequenza si fonda sul presupposto che il tempo di elaborazione di un processo periodico resti uguale per ogni cpu burst: ogni volta che il processo utilizza la cpu, dunque, la durata della sua esecuzione è la stessa.

Consideriamo, per esempio, due processi P_1 e P_2 , i cui periodi siano rispettivamente $p_1 = 50$ e $p_2 = 100$. Supponiamo che i tempi di elaborazione siano $t_1 = 20$ e $t_2 = 35$, rispettivamente. La scadenza di ogni processo impone loro di terminare l'esecuzione entro l'inizio del periodo seguente.

Per prima cosa dobbiamo chiederci se sia possibile schedulare l'esecuzione di ciascun task in modo da rispettare la sua scadenza. Se misuriamo l'utilizzo percentuale della cpu da parte di un processo P_i con il quoziente t_i/p_i , l'utilizzo della cpu da parte di P_1 è $20/50 = 0,40$, mentre quello di P_2 è $35/100 = 0,35$, per un utilizzo totale della cpu del 75 per cento. Sembra quindi che il problema di scheduling sia risolvibile rispettando le scadenze dei processi, e lasciando anche dei cicli di cpu disponibili.

Per cominciare, ipotizziamo di assegnare a P_2 una priorità più alta che a P_1 . L'esecuzione di P_1 e P_2 è illustrata dalla Figura 5.21. Come si può vedere, P_2 inizia l'esecuzione per primo e la termina dopo 35 millisecondi. A questo punto, è il turno di P_1 , che completa la sua esecuzione al millisecondo 55; ma la prima scadenza di P_1 scoccava al millisecondo 50: questa politica di scheduling non permette a P_1 di rispettare la propria scadenza.

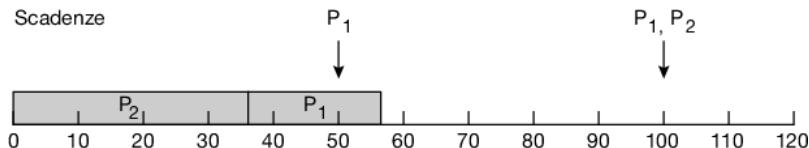


Figura 5.21 Scheduling dei task in caso P_2 abbia priorità maggiore di P_1 .

Proviamo adesso ad applicare lo scheduling con priorità proporzionale alla frequenza, in base al quale assegniamo a P_1 una priorità più elevata rispetto a P_2 , dato che il suo periodo è più breve. L'esecuzione di questi processi è mostrata nella Figura 5.22. Dapprima è eseguito P_1 , che termina al millisecondo 20, rispettando così la sua prima scadenza. Quindi, è il turno di P_2 , che prosegue fino al millisecondo 50. A questo punto, nonostante debba ancora usare la cpu per 5 millisecondi, P_2 lascia il posto per prelazione a P_1 . Quest'ultimo conclude la sua elaborazione al millisecondo 70, momento in cui lo scheduler riavvia P_2 , che termina al millisecondo 75, rispettando anch'esso la prima scadenza. Il sistema rimane inattivo fino al millisecondo 100, quando P_1 è nuovamente eseguito.

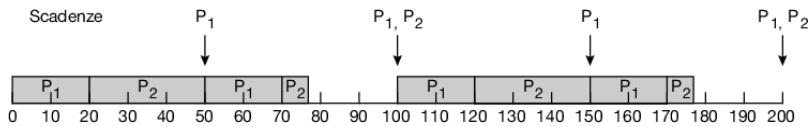


Figura 5.22 Scheduling con priorità proporzionale alla frequenza.

Lo scheduling con priorità proporzionale alla frequenza è un algoritmo ottimale, nel senso che se non è in grado di pianificare l'esecuzione di una serie di processi rispettandone i vincoli temporali, nessun altro algoritmo che assegna priorità statiche vi riuscirà. Consideriamo, ora, un insieme di processi la cui esecuzione non può essere schedulata dall'algoritmo con priorità proporzionale alla frequenza rispettando i vincoli temporali.

Supponiamo che il processo P_1 abbia periodo $p_1 = 50$ e tempo d'esecuzione $t_1 = 25$, e che il processo P_2 abbia invece parametri corrispondenti pari a $p_2 = 80$ e $t_2 = 35$. Lo scheduling con priorità proporzionale alla frequenza assegnerebbe a P_1 la priorità più elevata, visto che ha il periodo più breve. Nel complesso la cpu è utilizzata dai due processi per $(25/50)+(35/80) = 0,94$, quindi sembra ragionevole attendersi che il problema di scheduling abbia soluzione, lasciando alla cpu un 6 per cento di tempo libero. La Figura 5.23 illustra lo scheduling dei processi P_1 e P_2 . Nella fase iniziale, P_1 termina la sua prima esecuzione al millisecondo 25. Il processo P_2 , invece, va avanti fino al millisecondo 50, e qui si arresta perché P_1 fa valere il diritto di prelazione. A questo punto P_2 ha bisogno di altri 10 millisecondi per terminare la sua esecuzione. Il processo P_1 continua a girare fino al millisecondo 75, ma P_2 viola la sua scadenza, che scoccava al millisecondo 80.

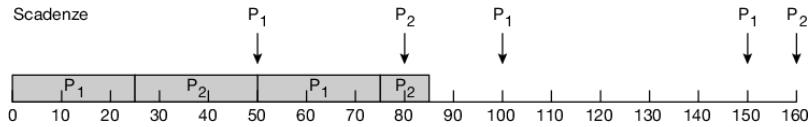


Figura 5.23 Scadenze non rispettate con lo scheduling con priorità proporzionale alla frequenza.

Lo scheduling con priorità proporzionale alla frequenza, dunque, benché ottimale nel senso descritto sopra, presenta una limitazione: l'utilizzo della cpu è limitato, per cui non sempre è possibile ricavarne un rendimento ottimale. Il caso peggiore di utilizzo della cpu per lo scheduling di N processi ammonta a

$$N(2^{1/N} - 1)$$

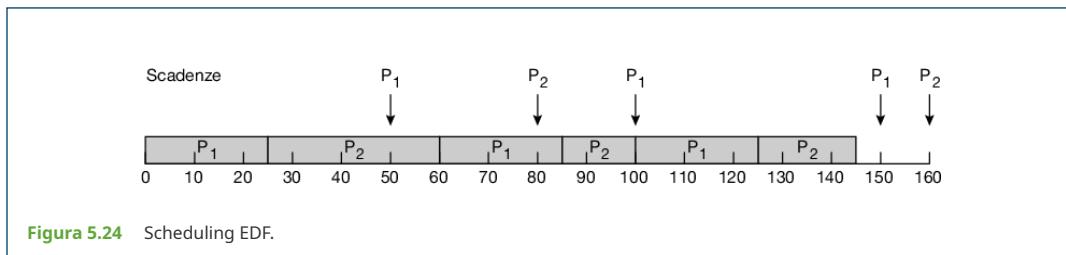
Con un solo processo nel sistema, la cpu è sfruttata al 100 per cento, ma la percentuale scende al 69 per cento circa quando il numero di processi tende all'infinito. Con due processi l'utilizzo della cpu è limitato a circa l'83 per cento. L'utilizzo congiunto della cpu per i due processi programmati nelle Figure 5.21 e 5.22 ammonta al 75 per cento; pertanto, lo scheduling con priorità proporzionale alla frequenza ne garantisce l'esecuzione entro le rispettive scadenze. Per i due processi nella Figura 5.23 l'utilizzo complessivo della cpu è del 94 per cento circa; di conseguenza, lo scheduling con priorità proporzionale alla frequenza non è in grado di pianificare l'esecuzione nel rispetto dei loro vincoli temporali.

5.6.4 Scheduling edf

Lo scheduling *edf* (earliest-deadline-first, ossia "per prima la scadenza più ravvicinata"), attribuisce le priorità dinamicamente, sulla base delle scadenze. Più vicina è la scadenza, maggiore è la priorità; una scadenza più lontana implica una priorità più bassa. Nel modello edf, un processo pronto per l'esecuzione deve notificare al sistema la propria scadenza. Potrà essere allora necessario modificare le priorità vigenti per tenere conto della scadenza del nuovo processo eseguibile. Si osservi come questo procedimento sia diverso dallo scheduling con priorità proporzionale alla frequenza, che ipotizza priorità fisse.

Per illustrare lo scheduling edf ci serviamo nuovamente dei processi raffigurati nella Figura 5.23, per i quali lo scheduling con priorità proporzionale alla frequenza risultava inadeguato. Ricordiamo che i parametri di P_1 e P_2 sono $p_1 = 50$, $t_1 = 25$ e $p_2 = 80$, $t_2 = 35$, rispettivamente. Lo scheduling edf di questi processi è mostrato nella Figura 5.24. Il processo P_1 ha la scadenza più vicina, quindi parte con una priorità più alta rispetto a P_2 . Non appena P_1 termina l'esecuzione, il processo P_2 inizia a girare. Laddove, però, lo scheduling con priorità proporzionale alla frequenza consente a P_1 di esercitare prelazione nei confronti di P_2 all'inizio del suo periodo successivo, cioè al millisecondo 50, lo scheduling edf lascia che P_2 continui a girare. Infatti, P_2 ha adesso priorità più alta rispetto a P_1 , perché la sua prossima scadenza (al millisecondo 80) precede quella di P_1 (al millisecondo 100). Pertanto, sia P_1 sia P_2 hanno onorato le loro prime scadenze. Il processo P_1 ritorna in esecuzione al millisecondo 60, e termina al millisecondo 85,

rispettando anche la seconda scadenza, che scocca al millisecondo 100. E a questo punto che P_2 comincia a girare, ma subisce prelazione da parte di P_1 , che è all'inizio del suo periodo successivo, al millisecondo 100. La prelazione si applica perché la scadenza di P_1 (al millisecondo 150) è più vicina di quella di P_2 (al millisecondo 160). Al millisecondo 125 P_1 termina l'esecuzione e P_2 riparte, completando al millisecondo 145, e ottemperando così alla sua scadenza. Il sistema rimane inattivo fino al millisecondo 150, allorché P_1 è ancora una volta pronto per l'esecuzione.



A differenza dell'algoritmo con priorità proporzionale alla frequenza, lo scheduling edf non postula la periodicità dei processi, e non prevede neanche di impiegare sempre lo stesso tempo della cpu per ogni burst. L'unico obbligo a carico dei processi è di notificare allo scheduler la propria prossima scadenza nel momento in cui divengano eseguibili. Il vantaggio dello scheduling edf è di essere teoricamente ottimo. Idealmente, esso garantisce l'esecuzione di tutti i processi entro le proprie scadenze, sfruttando inoltre la cpu al 100 per cento. Nella pratica, tuttavia, un rendimento così alto della cpu è impossibile, per il rallentamento dovuto ai cambi di contesto tra processi e alla gestione delle interruzioni.

5.6.5 Scheduling a quote proporzionali

Lo scheduler a quote proporzionali opera distribuendo un certo numero di quote, diciamo T , fra tutte le applicazioni. Un'applicazione può ricevere N quote di tempo, assicurandosi così l'uso di una frazione N/T del tempo totale del processore. Poniamo, per esempio, di avere un totale di quote $T = 100$, da ripartire fra tre processi, A , B e C . A può disporre di 50 quote, B di 15, mentre C ha 20 quote. Questa suddivisione assicura che A possa sfruttare il 50 percento del tempo totale del processore, B il 15 percento e C il 20 percento.

Lo scheduler a quote proporzionali deve lavorare in sinergia con un meccanismo di controllo dell'ammissione, per garantire che ogni applicazione possa effettivamente ricevere le quote di tempo che le sono state destinate. Il meccanismo di controllo dell'ammissione accetterà le richieste da parte dei processi solo a condizione che il numero di quote disponibili sia sufficiente. Nell'esempio sopra citato, abbiamo assegnato $50 + 15 + 20 = 85$ quote su un totale di 100; se un nuovo processo D ne richiedesse 30, il meccanismo di controllo dell'ammissione gli negherebbe l'accesso al sistema.

5.6.6 Scheduling real-time di posix

Lo standard posix fornisce un'estensione per l'elaborazione real-time, detta posix.1b. In questo paragrafo si descrivono alcune api di posix relative alla programmazione real-time dei thread. posix definisce due classi di scheduling per i thread real-time:

- sched_fifo;
- sched_rr.

sched_fifo pianifica l'esecuzione dei thread secondo la politica fcfs esposta al Paragrafo 5.3.1. Non è prevista alcuna forma di time slicing fra thread di pari priorità. Pertanto, il thread real-time con priorità più elevata in prima posizione nella coda fifo avrà accesso alla cpu fin quando termina o si arresta. sched_rr adotta una politica round-robin ed è simile a sched_fifo, eccetto che prevede il time slicing fra thread di uguale priorità. posix fornisce una terza classe di scheduling – sched_other – la cui implementazione è indefinita e dipendente dal sistema; il suo comportamento può variare da sistema a sistema.

La api posix specifica le due funzioni seguenti per leggere e impostare le politiche di scheduling:

- pthread_attr_getschedpolicy(pthread_attr_t *attr, int *policy)
- pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)

In entrambe le funzioni il primo parametro è un puntatore all'insieme degli attributi del thread. Il secondo parametro è, nel primo caso, un puntatore a un intero impostato alla politica di scheduling corrente; nel secondo caso, un valore intero che denota sched_fifo, sched_rr o sched_other. Entrambe le funzioni restituiscono valori non nulli in caso di errore.

La Figura 5.25 contiene un programma posix Pthread che usa questa api. Il programma appura in primo luogo la politica di scheduling vigente, per scegliere poi l'algoritmo di scheduling sched_fifo.

```
#include <pthread.h>
```

```

#include <stdio.h>

#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;

    pthread_t tid[NUM_THREADS];

    pthread_attr_t attr;

    /* appura gli attributi di default */

    pthread_attr_init(&attr);

    /* appura la politica di scheduling corrente */

    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)

        fprintf(stderr, "Unable to get policy.\n");

    else {

        if (policy == SCHED_OTHER)

            printf("SCHED_OTHER\n");

        else if (policy == SCHED_RR)

            printf("SCHED_RR\n");

        else if (policy == SCHED_FIFO)

            printf("SCHED_FIFO\n");

    }

    /* imposta la politica di scheduling - FIFO, RR o OTHER */

    if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)

        fprintf(stderr, "Unable to set policy.\n");

    /* genera i thread */

    for (i = 0; i < NUM_THREADS; i++)

        pthread_create(&tid[i], &attr, runner, NULL);

    /* ora attende la terminazione di ciascun thread */

    for (i = 0; i < NUM_THREADS; i++)

        pthread_join(tid[i], NULL);

    }

    /* ciascun thread comincia l'esecuzione da questa funzione */

    void *runner(void *param)

    {

        /* fai qualcosa ... */

```

```
    pthread_exit(0);  
}
```

Figura 5.25 Scheduling real-time con la api posix.

5.7 Esempi di sistemi operativi

Procediamo ora nella descrizione dei criteri di scheduling per i sistemi operativi Linux, Windows e Solaris.

È importante notare che viene qui utilizzato il termine *scheduling dei processi* in senso generale. In realtà nei sistemi Solaris e Windows stiamo descrivendo lo *scheduling dei thread del kernel* e in Linux lo *scheduling dei task*.

5.7.1 Un esempio: scheduling di Linux

Lo scheduling dei processi in Linux ha avuto una storia interessante. Prima della versione 2.5, il kernel Linux utilizzava una variante dell'algoritmo tradizionale di scheduling di unix.

Tuttavia questo algoritmo, progettato senza pensare ai sistemi smp, non supporta adeguatamente sistemi multiprocessore e fornisce prestazioni scarse nei sistemi in grado di eseguire un gran numero di processi. Con la versione 2.5 del kernel, lo scheduler è stato rivisitato per includere un algoritmo di scheduling noto come $O(1)$, che veniva eseguito in tempo costante indipendentemente dal numero di task nel sistema. Lo scheduler $O(1)$ forniva anche un maggiore supporto ai sistemi smp fra cui la gestione della predilezione e il bilanciamento del carico. Tuttavia, in pratica, anche se lo scheduler $O(1)$ forniva eccellenti prestazioni su sistemi smp, portava a tempi di risposta troppo scarsi sui processi interattivi, comuni in molti sistemi desktop. Durante lo sviluppo del kernel 2.6, lo scheduler è stato nuovamente rivisto e dalla versione 2.6.23 del kernel, lo scheduler cfs (*completely fair scheduler*) è diventato l'algoritmo predefinito di scheduling Linux.

Nei sistemi Linux lo scheduling si basa sulle classi di scheduling. A ogni classe è assegnata una specifica priorità. Utilizzando diverse classi di scheduling, il kernel può utilizzare algoritmi distinti in base alle esigenze del sistema e dei suoi processi. I criteri di scheduling per un server Linux, per esempio, possono essere diversi da quelli per un dispositivo mobile che utilizza Linux. Per decidere quale task eseguire, lo scheduler seleziona il task con priorità più alta appartenente alla classe di scheduling a priorità più elevata. Il kernel Linux standard implementa due classi di scheduling, che verranno discusse nel seguito: (1) una classe di scheduling predefinita che utilizza l'algoritmo cfs e (2) una classe di scheduling real-time. Possono naturalmente essere aggiunte ulteriori classi.

Invece di utilizzare regole rigide che associano un valore di priorità relativo alla lunghezza di un quanto di tempo, lo scheduler cfs assegna a ogni task una percentuale del tempo di elaborazione della cpu. Questa percentuale è calcolata sulla base dei valori nice value assegnati a ciascun task. I nice value vanno da -20 a +19, dove un valore numerico più basso indica una priorità relativa superiore. I task con nice value minori ricevono una maggiore percentuale di tempo di elaborazione della cpu rispetto ai task con nice value più alti. Il nice value di default è 0. Il termine nice value deriva dall'idea che se un task aumenta il suo nice value, per esempio da 0 a +10, si comporta in maniera gentile – in inglese “nice” – rispetto agli altri task del sistema, abbassando la sua priorità relativa. cfs non utilizza valori discreti per i quanti di tempo, ma piuttosto definisce una latenza obiettivo (*targeted latency*), cioè un intervallo di tempo entro il quale ogni task eseguibile dovrebbe andare in esecuzione almeno una volta. Le porzioni di tempo di cpu vengono assegnate a partire dal valore della latenza obiettivo. La latenza obiettivo, che ha valore di default e un valore minimo, può aumentare qualora il numero di task attivi nel sistema cresca oltre una certa soglia.

Lo scheduler cfs non assegna direttamente le priorità, ma registra per quanto tempo ogni task è stato eseguito mantenendo il tempo di esecuzione virtuale di ogni task nella variabile `vruntime`. Il tempo di esecuzione virtuale è associato a un fattore di decadimento che dipende dalla priorità di un task: task a bassa priorità hanno fattori di decadimento più alti di task ad alta priorità. Per i task con priorità normale (nice value pari a 0), il tempo di esecuzione virtuale coincide con il tempo effettivo di esecuzione. Quindi, se un task con priorità normale viene eseguito per 200 millisecondi, anche il suo `vruntime` sarà di 200 millisecondi, ma se un task con priorità inferiore viene eseguito per 200 millisecondi, il suo `vruntime` sarà superiore a 200 millisecondi. Analogamente, se un task con priorità più alta viene eseguito per 200 millisecondi, il suo `vruntime` sarà inferiore a 200 millisecondi. Per decidere il prossimo task da eseguire, lo scheduler seleziona semplicemente il task con il valore `vruntime` più piccolo. Inoltre, un task con priorità più alta che diventa disponibile per l'esecuzione può avere prelazione su un task con priorità inferiore.

Esaminiamo lo scheduler cfs in azione. Si supponga che due task abbiano lo stesso nice value. Un task è i/o-bound e l'altro è cpu-bound. In genere, un task i/o-bound verrà eseguito solo per brevi periodi di tempo prima di bloccarsi in attesa di i/o, mentre un task cpu-bound esaurirà il suo periodo di tempo ogni volta che avrà l'opportunità di essere in esecuzione su un processore. Pertanto, il valore di `vruntime` del task i/o-bound diventerà inferiore rispetto a quello del task cpu-bound, dando così al task i/o-bound una priorità superiore. A questo punto, se il task cpu-bound è in esecuzione quando il task i/o-bound diventa pronto per l'esecuzione (per esempio, quando l'i/o che il task sta attendendo diventa disponibile), il task i/o-bound effettuerà la prelazione del task cpu-bound.

Linux implementa anche lo scheduling in tempo reale utilizzando lo standard posix, come descritto nel Paragrafo 5.6.6. Qualsiasi task pianificato utilizzando le politiche real-time `sched_fifo` o `sched_rr` viene eseguito con una priorità più alta rispetto ai normali task (non real-time). Linux usa due intervalli di priorità distinti, uno per i task real-time e l'altro per i normali task. Ai task real-time vengono assegnate priorità statiche nell'intervallo 0-99, mentre ai task non real-time vengono assegnate priorità nell'intervallo 100-139. Questi due intervalli sono mappati in uno schema di priorità globale in cui i valori numericamente inferiori indicano priorità relative più alte. Ai task normali vengono assegnate priorità in base ai loro nice value: il valore -20 viene mappato nella priorità 100 e il valore +19 nella priorità 139. Questo schema è mostrato nella Figura 5.26.

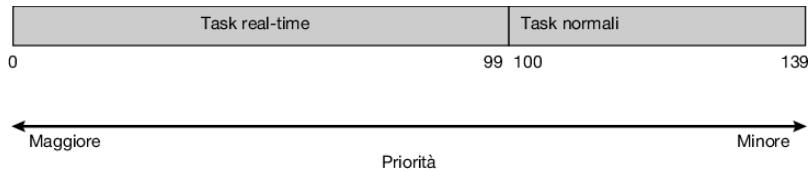


Figura 5.26 Priorità di scheduling in Linux.

Lo scheduler cfs supporta anche il bilanciamento del carico, utilizzando una sofisticata tecnica che equilibra il carico tra i core di elaborazione, ma è anche compatibile con numa (numa-aware) e riduce al minimo la migrazione dei thread. cfs definisce il carico di ogni thread come combinazione della sua priorità e del suo tasso medio di utilizzo della cpu. Pertanto, un thread con una priorità elevata, ma che si dedica per lo più all'i/o e richiede poco utilizzo della cpu, ha un carico generalmente basso, simile al carico di un thread con bassa priorità che ha un elevato utilizzo della cpu. Utilizzando questa metrica, il carico di una coda è la somma dei carichi di tutti i thread nella coda e il bilanciamento garantisce semplicemente che tutte le code abbiano approssimativamente lo stesso carico.

Come evidenziato nel Paragrafo 5.5.4, tuttavia, la migrazione di un thread può penalizzare l'accesso alla memoria a causa della necessità di invalidare il contenuto della cache o, nei sistemi numa, di incorrere in tempi di accesso alla memoria più lunghi. Per risolvere questo problema, Linux identifica un sistema gerarchico di domini di scheduling. Un dominio di scheduling è un insieme di core che può essere bilanciato l'uno con l'altro, come mostrato nella Figura 5.27. I core in ciascun dominio di scheduling sono raggruppati in base al modo in cui condividono le risorse del sistema. Per esempio, sebbene ciascun core mostrato nella Figura 5.27 possa avere una propria cache di livello 1 (L1), coppie di core condividono una cache di livello 2 (L2) e sono quindi organizzate in due domini separati dominio_0 e dominio_1 . Allo stesso modo, questi due domini possono condividere una cache di livello 3 (L3) e sono dunque organizzati in un dominio a livello di processore (noto anche come nodo numa). Facendo un ulteriore passo in avanti, su un sistema numa un dominio più grande a livello di sistema combinerebbe nodi numa distinti a livello di processore.

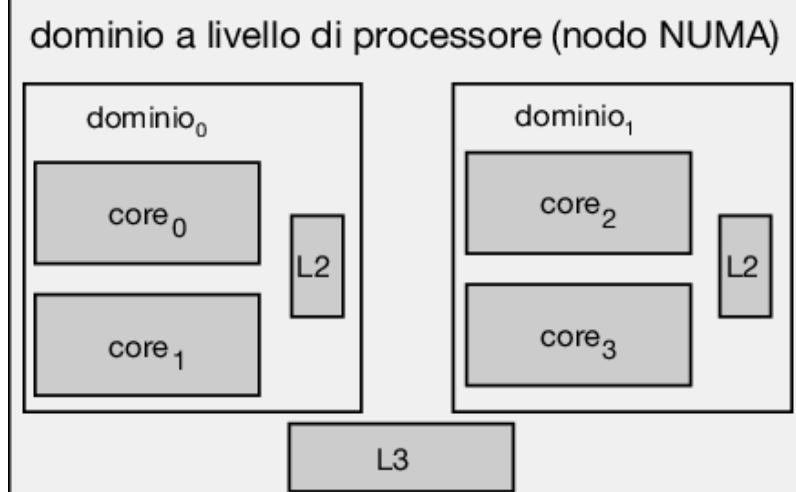


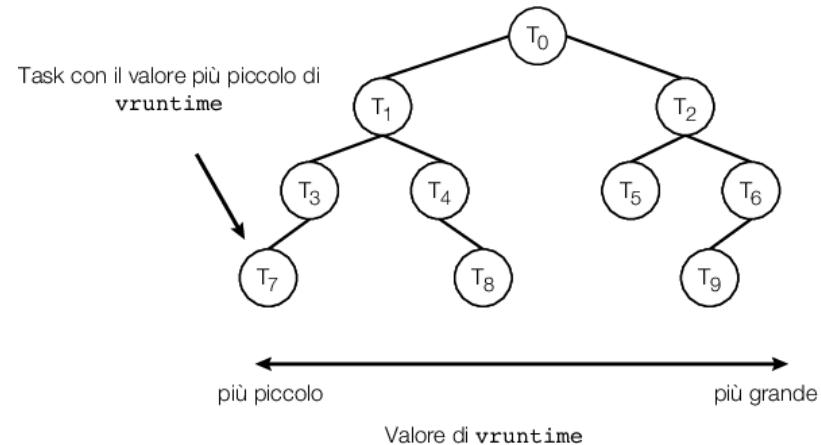
Figura 5.27 Bilanciamento del carico numa-aware nello scheduler cfs di Linux.

La strategia generale alla base di cfs è di bilanciare i carichi all'interno dei domini, iniziando dal livello più basso della gerarchia. Prendendo la Figura 5.27 come esempio, inizialmente un thread migrerebbe solo tra core sullo stesso dominio (cioè all'interno di dominio_0 o di dominio_1). Al livello successivo il bilanciamento del carico avverrebbe tra dominio_0 e dominio_1 . cfs tende a non migrare i thread tra nodi numa distinti quando un thread venisse spostato più lontano dalla sua memoria locale; una tale migrazione si potrebbe verificare solo in presenza di gravi squilibri di carico. Come regola generale, se il sistema nel suo complesso è occupato, cfs

non eseguirà il bilanciamento del carico oltre il dominio locale di ciascun core per evitare peggioramenti della latenza della memoria nei sistemi numa.

PRESTAZIONI DI CFS

Lo scheduler cfs di Linux fornisce un efficiente algoritmo per la selezione del prossimo task da eseguire. Ogni task eseguibile è posto in un R-B albero, un albero binario di ricerca bilanciato, la cui chiave si basa sul valore di `vruntime`. L'albero è il seguente:



Quando un task diventa eseguibile viene aggiunto all'albero. Se un task dell'albero non è eseguibile (per esempio, se è bloccato in attesa di i/o), viene rimosso. In generale, i task a cui è stato dato meno tempo di elaborazione (valori minori di `vruntime`) si trovano nel lato sinistro dell'albero e i task a cui è stato dato più tempo di elaborazione si trovano sul lato destro. Secondo le proprietà di un albero binario di ricerca il nodo più a sinistra ha il valore della chiave più piccolo, il che significa, per lo scheduler cfs, che è il task con la massima priorità. Poiché l'albero R-B è bilanciato, la ricerca del nodo più a sinistra richiede $O(\log N)$ operazioni (dove N è il numero di nodi dell'albero). Tuttavia, per ragioni di efficienza, lo scheduler Linux memorizza questo valore nella variabile `rb_leftmost` in modo da poter determinare qual è il prossimo task da eseguire recuperando semplicemente il valore memorizzato.

5.7.2 Un esempio: scheduling di Windows

Il sistema operativo Windows compie lo scheduling dei thread servendosi di un algoritmo basato su priorità e prelazione. Lo scheduler di Windows assicura che si eseguano sempre i thread a più alta priorità. La porzione del kernel che si occupa dello scheduling si chiama *dispatcher*. Una volta selezionato dal *dispatcher*, un thread viene eseguito finché non sia sottoposto a prelazione da un altro thread a priorità più alta oppure termini, esaurisce il suo quanto di tempo o esegua una chiamata di sistema bloccante, per esempio un'operazione di i/o. Se un thread d'elaborazione in tempo reale, ad alta priorità, diventa pronto per l'esecuzione mentre è in esecuzione un thread a bassa priorità, quest'ultimo viene sottoposto a prelazione. Ciò realizza un accesso preferenziale alla cpu per i thread d'elaborazione in tempo reale che ne hanno necessità.

Per determinare l'ordine d'esecuzione dei thread il *dispatcher* impiega uno schema di priorità a 32 livelli. Le priorità sono suddivise in due classi: la classe variabile raccoglie i thread con priorità da 1 a 15, mentre la classe real-time raccoglie i thread con priorità tra 16 e 31 (esiste anche un thread, per la gestione della memoria, che si esegue con priorità 0). Il *dispatcher* adopera una coda per ciascuna priorità di scheduling e percorre l'insieme delle code da quella a priorità più alta a quella a priorità più bassa, finché trova un thread pronto per l'esecuzione. In assenza di tali thread, il *dispatcher* manda in esecuzione un thread speciale detto idle thread.

C'è una relazione tra le priorità numeriche del kernel del sistema operativo Windows e quelle dell'api Windows. Secondo l'api Windows un processo può appartenere a una delle seguenti classi di priorità:

- idle_priority_class
- below_normal_priority_class
- normal_priority_class
- above_normal_priority_class
- high_priority_class
- realtime_priority_class

Di solito, i processi appartengono alla classe normal_priority_class, sempre che il processo genitore non appartenga alla classe idle_priority_class, o sia stata specificata un'altra classe alla creazione del processo. La classe di priorità di un processo può essere modificata mediante la funzione `SetPriorityClass()` dell'api Windows. Le priorità di ciascuna classe eccetto realtime_priority_class sono priorità di classe variabile, quindi la priorità di un thread appartenente a queste classi può cambiare.

Un thread, nell'ambito di una classe di priorità, ha a sua volta una priorità relativa, i cui valori comprendono i seguenti:

- idle
- lowest
- below_normal
- normal
- above_normal
- highest
- time_critical

La priorità di ciascun thread dipende dalla priorità della classe cui appartiene e dalla priorità relativa che il thread ha all'interno della stessa classe. Questa relazione è rappresentata nella Figura 5.28. I valori di ciascuna classe di priorità sono riportati nella prima riga in alto. La prima colonna a sinistra contiene i valori delle diverse priorità relative. Per esempio, se la priorità relativa di un thread nella classe above_normal_priority_class è normal, la priorità numerica di quel thread è 10.

	real-time	high	above_normal	normal	below_normal	idle_priority
time_critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above_normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below_normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figura 5.28 Priorità dei thread in Windows.

Inoltre, ogni thread ha una priorità di base che rappresenta un valore nell'intervallo di priorità della classe di appartenenza. Il valore predefinito per la priorità di base in una classe è quello della priorità relativa normal per quella classe. Le priorità di base per ciascuna classe di priorità sono le seguenti:

- realtime_priority_class-24
- high_priority_class-13
- above_normal_priority_class-10
- normal_priority_class-8
- below_normal_priority_class-6
- idle_priority_class-4.

Di solito la priorità iniziale di un thread è la priorità di base del processo a cui il thread appartiene, anche se può essere utilizzata la funzione `SetThreadPriority()` nella api di Windows per modificare la priorità di base di un thread.

Quando il quanto di tempo di un thread si esaurisce, il thread viene interrotto e se il thread fa parte della classe a priorità variabile, la sua priorità viene ridotta. Tuttavia, la priorità non si abbassa mai sotto la priorità di base. L'abbassamento della priorità tende a limitare l'uso della cpu da parte dei thread con prevalenza d'elaborazione. Se un thread a priorità variabile è rilasciato da un'operazione d'attesa, il *dispatcher* aumenta la sua priorità. L'entità di questo aumento dipende dal tipo d'evento che il thread attendeva: un thread che attendeva dati dalla tastiera riceve un forte aumento di priorità, uno che attendeva operazioni relative a un disco riceve un aumento più moderato. Questa strategia mira a fornire buoni tempi di risposta per i thread interattivi, con interfacce basate su mouse e finestre; permette inoltre ai thread con prevalenza di i/o di tenere occupati i dispositivi di i/o, e rende nel contempo possibile l'utilizzo in background dei cicli di cpu inutilizzati da parte dei thread con prevalenza d'elaborazione. Questa strategia si segue in molti sistemi operativi in time-sharing, compreso unix. Inoltre, per migliorare il tempo di risposta, la finestra attraverso cui l'utente sta interagendo ottiene un incremento di priorità.

Quando un utente richiede l'esecuzione di un programma interattivo, il sistema deve fornire al relativo processo prestazioni particolarmente elevate. Per questa ragione, il sistema Windows segue una regola specifica di scheduling per i processi della classe

`normal_priority_class`. Il sistema operativo Windows distingue tra il *processo in foreground*, correntemente selezionato sullo schermo e i *processi in background*, che non sono attualmente selezionati. Quando un processo passa in foreground, Windows aumenta il suo quanto di tempo di un certo fattore, tipicamente pari a 3; ciò fa sì che il processo in foreground possa continuare la propria esecuzione per un tempo tre volte più lungo, prima che si abbia una prelazione dovuta al time-sharing.

Windows 7 ha introdotto lo scheduling in modalità utente (ums), che consente alle applicazioni di creare e gestire i thread in maniera indipendente dal kernel. Un'applicazione può quindi creare e schedulare più thread senza coinvolgere l'utilità di scheduling del kernel Windows. Per le applicazioni che creano un gran numero di thread lo scheduling in modalità utente è molto più efficiente rispetto a quello in modalità kernel, proprio perché non è necessario alcun intervento del kernel.

Le versioni precedenti di Windows fornivano una caratteristica simile che permetteva a diversi thread in modalità utente (detti fibre) di essere associati a un singolo thread del kernel. Tuttavia, le fibre erano di utilità pratica limitata. Una fibra non era in grado di effettuare chiamate alle api di Windows, perché tutte le fibre dovevano condividere il blocco di ambiente (teb, *thread environment block*) del thread su cui erano in esecuzione. Ciò presentava un problema quando una funzione dell'api di Windows inseriva informazioni di stato per una fibra nel teb, perché queste informazioni potevano essere sovrascritte da altre fibre. ums ha superato questi problemi, fornendo a ogni thread in modalità utente il proprio contesto privato.

Inoltre, a differenza delle fibre, ums non è destinato a essere utilizzato direttamente dal programmatore. La programmazione di uno scheduler in modalità utente può essere molto impegnativa. ums non include un tale scheduler: gli scheduler provengono invece da librerie costruite su ums. Per esempio, Microsoft fornisce la libreria ConcRT (*concurrency runtime*), un framework per la programmazione concorrente C++ progettato per il parallelismo basato sui task (Paragrafo 4.2) su processori multicore. ConcRT fornisce uno scheduler in modalità utente e alcune funzionalità per scomporre i programmi in task da pianificare successivamente sui core disponibili.

Windows supporta anche lo scheduling su sistemi multiprocessore, come descritto nel Paragrafo 5.5, tentando di schedulare un thread sul core di elaborazione ottimale per quel thread, il che comprende il principio di mantenere il thread sul suo processore preferito o più recente. Una tecnica utilizzata da Windows consiste nel creare insiemi di processori logici (noti come set smt). Su un sistema smt hyper-threaded, i thread hardware appartenenti allo stesso core della cpu appartengono anche allo stesso set smt. I processori logici sono numerati a partire da 0, per esempio un sistema dual-threaded/quad-core conterebbe otto processori logici, raggruppati in quattro set smt: {0, 1}, {2, 3}, {4, 5} e {6, 7}. Per evitare le penalizzazioni nell'accesso alla memoria cache evidenziate nel Paragrafo 5.5.4, lo scheduler tenta di mantenere un thread in esecuzione su processori logici all'interno dello stesso set smt.

Per distribuire i carichi tra diversi processori logici, a ogni thread viene assegnato un processore ideale, ovvero un numero che identifica il processore preferito dal thread. Ogni processo ha un valore iniziale che identifica la cpu ideale per un thread che appartiene a quel processo. Questo valore viene incrementato per ogni nuovo thread creato da quel processo, permettendo così la distribuzione del carico tra diversi processori logici. Sui sistemi smt, l'incremento per assegnare il successivo processore ideale porta nel set smt successivo. Per esempio, su un sistema dual-threaded/quad-core, i processori ideali per i thread in un processo specifico verrebbero assegnati nell'ordine 0, 2, 4, 6, 0, 2, ... Per evitare la situazione in cui al primo thread di ogni processo venga assegnato il processore 0, ai processi vengono assegnati valori iniziali distinti, distribuendo in tal modo i thread su tutti i core di elaborazione fisici nel sistema. Continuando il nostro esempio, se il valore iniziale per un secondo processo fosse 1, i processori ideali sarebbero assegnati nell'ordine 1, 3, 5, 7, 1, 3 e così via.

5.7.3 Un esempio: scheduling di Solaris

Solaris utilizza uno scheduling dei thread basato sulle priorità in cui ogni thread appartiene a una delle sei seguenti classi.

1. Time-sharing (ts).
2. Interattivo (ia).
3. Real-time (rt).
4. Sistema (sys).
5. Ripartizione equa (fss, per *fair share*).
6. Priorità fissa (fp).

All'interno di ciascuna classe vi sono priorità e algoritmi di scheduling differenti.

La classe di scheduling predefinita per i processi è quella time-sharing. È basata su un criterio di scheduling che modifica dinamicamente le priorità, assegnando porzioni di tempo variabili, grazie a una coda multilivello con retroazione. Per default, tra le priorità e le frazioni di tempo sussiste una relazione inversa: più alta è la priorità, minore la frazione di tempo associata; più bassa è la priorità, maggiore sarà la frazione di tempo. Di solito, i processi interattivi hanno priorità alta, mentre i processi con prevalenza d'elaborazione hanno priorità bassa. Questo criterio di scheduling offre un buon tempo di risposta per i processi interattivi e una buona produttività per i processi con prevalenza d'elaborazione. La classe interattiva utilizza gli stessi criteri di scheduling di quella time-sharing, ma privilegia le applicazioni dotate di interfacce a finestre (come quelle create dagli ambienti kde e gnome), a cui attribuisce priorità più elevate per ottenere prestazioni migliori.

La Figura 5.29 mostra la tabella semplificata di dispatch per lo scheduling dei thread interattivi e a tempo ripartito. Queste due classi contemplano 60 livelli di priorità; ne elenchiemo solo alcuni. (Per vedere la tabella di dispatch completa su un sistema o una vm Solaris occorre eseguire il comando `dispadmin -c TS -g`). La tabella di dispatch della Figura 5.29 contiene i seguenti campi.

priorità	quanto di tempo	quanto di tempo esaurito	ripresa dell'attività
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51

priorità	quanto di tempo	quanto di tempo esaurito	ripresa dell'attività
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figura 5.29 Tabella di dispatch di Solaris per i thread interattivi e a tempo ripartito.

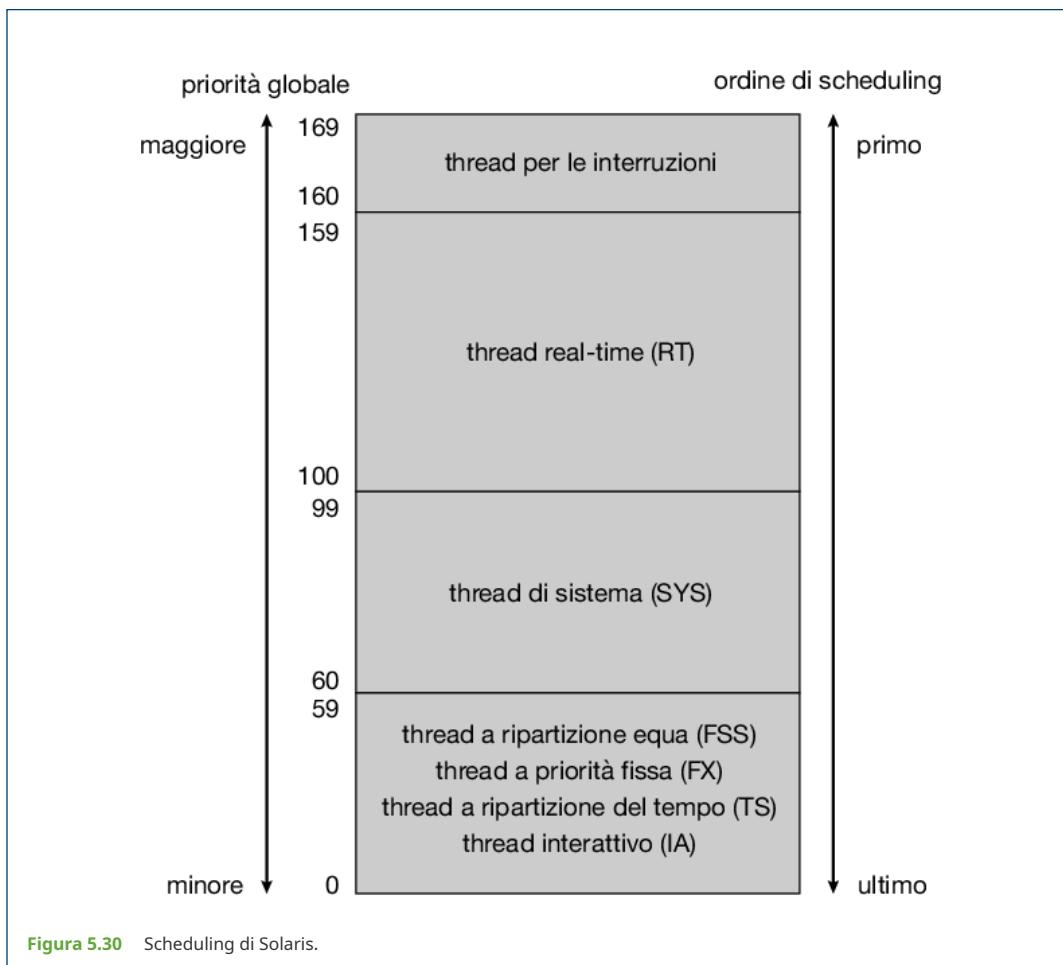
- Priorità. È la priorità dipendente dalle classi, nel caso di quelle interattiva e a tempo ripartito. Il valore cresce al crescere della priorità.
- Quanto di tempo. È il quanto di tempo della relativa priorità. Come si può notare, priorità e frazioni di tempo sono inversamente correlate: alla priorità 0, infatti, spetta il quanto di tempo più lungo (200 millisecondi), mentre alla priorità 59, cioè la più alta, corrisponde il quanto minima (20 millisecondi).
- Quanto di tempo esaurito. È la nuova priorità dei thread che abbiano consumato l'intero quanto di tempo loro assegnato senza sospendersi. Tali thread sono considerati a prevalenza d'elaborazione; le loro priorità, come evidenziato dalla tabella, subiscono una diminuzione.
- Ripresa dell'attività. È la priorità di un thread che ritorni in attività dopo un periodo di attesa (dell'i/o, per esempio). Come si può vedere nella tabella, quando è disponibile l'i/o per un thread in attesa, la priorità del thread viene incrementata fino a un valore compreso tra 50 e 59. Si implementa così il criterio di scheduling che consiste nel fornire risposte sollecite ai processi interattivi.

La classe dei thread real-time ha la priorità maggiore. Un processo real-time sarà eseguito prima di un processo appartenente a qualsiasi altra classe. Ciò fa sì che i processi real-time abbiano la garanzia di ottenere una risposta dal sistema entro limiti di tempo prefissati. In generale, tuttavia, pochi processi appartengono alla classe real-time.

Solaris sfrutta la classe sistema per eseguire i thread del kernel, quali lo scheduler e il demone per la paginazione. La priorità di un thread di sistema, una volta fissata, non cambia. La classe sistema è riservata all'uso da parte del kernel (i processi utenti eseguiti in modalità di sistema non appartengono alla classe sistema).

Le classi a priorità fissa e a ripartizione equa sono state introdotte in Solaris 9. I thread appartenenti alla classe a priorità fissa hanno lo stesso livello di priorità di quelli della classe time-sharing, ma le loro priorità non vengono modificate dinamicamente. Per la classe a ripartizione equa le decisioni di scheduling vengono prese sulla base delle quote (*shares*) di cpu, e non sulla base delle priorità. Le quote di cpu sono assegnate a un insieme di processi, chiamato progetto, e indicano in che misura il progetto ha diritto all'uso delle risorse disponibili.

Ogni classe di scheduling include una scala di priorità. Tuttavia, lo scheduler converte le priorità specifiche della classe in priorità globali e sceglie per l'esecuzione il thread con la priorità globale più elevata. La cpu esegue il thread prescelto finché (1) si blocca, (2) esaurisce la propria frazione di tempo, o (3) è soggetto a prelazione da un thread con priorità più alta. Se vi sono più thread con la stessa priorità, lo scheduler utilizza una coda circolare rr. La Figura 5.30 mostra le relazioni fra le sei classi di scheduling, e quali sono le priorità globali a loro assegnate. Va notato che il kernel utilizza 10 thread per servire le interruzioni. Questi thread non appartengono ad alcuna classe e sono eseguiti con massima priorità (160-169). Come già ricordato, tradizionalmente Solaris utilizzava il modello da molti a molti (Paragrafo 4.3.3), ma a partire da Solaris 9 è passato al modello da uno a uno (Paragrafo 4.3.2).



5.8 Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della cpu per un sistema particolare. Come abbiamo visto nel Paragrafo 5.3, esistono molti algoritmi di scheduling, ciascuno dotato dei propri parametri; quindi, la scelta di un algoritmo può essere abbastanza difficile.

Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo. Nel Paragrafo 5.2 si spiega che i criteri si definiscono spesso in termini di utilizzo della cpu, tempo di risposta o produttività. Per scegliere un algoritmo occorre innanzitutto stabilire l'importanza relativa di queste misure. Tra i criteri suggeriti si possono inserire diverse misure, per esempio le seguenti:

- rendere massimo l'utilizzo della cpu con il vincolo che il massimo tempo di risposta sia 300 millisecondi;
- rendere massima la produttività in modo che il tempo di completamento sia (in media) linearmente proporzionale al tempo d'esecuzione totale.

Una volta definiti i criteri di selezione è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari possibili metodi di valutazione.

5.8.1 Modellazione deterministica

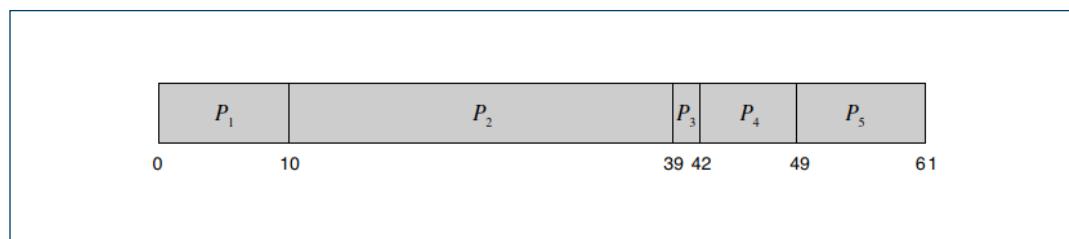
Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della valutazione analitica. La valutazione analitica, partendo dall'algoritmo dato e dal carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro.

La modellazione deterministica è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

Si supponga, per esempio, di avere il carico di lavoro illustrato di seguito; i cinque processi si presentano al tempo 0, nell'ordine dato, e la durata delle sequenze di operazioni della cpu è espressa in millisecondi:

Processo	Durata della sequenza
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Si può stabilire con quale fra gli algoritmi di scheduling fcfs, sjf e rr (con quanto di tempo = 10 millisecondi) per questo insieme di processi si ottenga il minimo tempo medio d'attesa. Con l'algoritmo fcfs i processi si eseguono secondo lo schema seguente.



Il tempo d'attesa è di 0 millisecondi per il processo P_1 , di 10 millisecondi per il processo P_2 , di 39 millisecondi per il processo P_3 , di 42 millisecondi per il processo P_4 e di 49 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(0 + 10 + 39 + 42 + 49)/5 = 28$ millisecondi.

Con l'algoritmo sjf senza prelazione i processi si eseguono come segue.

P_3	P_4	P_1	P_5	P_2
0	3	10	20	32

61

Il tempo d'attesa è di 10 millisecondi per il processo P_1 , di 32 millisecondi per il processo P_2 , di 0 millisecondi per il processo P_3 , di 3 millisecondi per il processo P_4 e di 20 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(10 + 32 + 0 + 3 + 20)/5 = 13$ millisecondi.

Con l'algoritmo rr i processi si eseguono come segue.

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2
0	10	20	23	30	40	50	52

61

Il tempo d'attesa è di 0 millisecondi per il processo P_1 , di 32 millisecondi per il processo P_2 , di 20 millisecondi per il processo P_3 , di 23 millisecondi per il processo P_4 e di 40 millisecondi per il processo P_5 . Quindi, il tempo d'attesa medio è di $(0 + 32 + 20 + 23 + 40)/5 = 23$ millisecondi.

È importante notare come, *in questo caso*, il criterio sjf fornisca come risultato un tempo medio d'attesa minore della metà del tempo corrispondente ottenuto con lo scheduling fcfs; l'algoritmo rr fornisce un risultato intermedio tra i precedenti.

La definizione e lo studio di un modello deterministico sono semplici e rapidi; i risultati sono numeri esatti che consentono il confronto tra gli algoritmi. Nondimeno, anche i parametri in ingresso devono essere numeri esatti e i risultati sono applicabili solo a quei casi. Il suo impiego principale consiste nella descrizione degli algoritmi di scheduling e nella presentazione d'esempi. Nei casi in cui vengano eseguiti ripetutamente gli stessi programmi e si possono misurare con precisione i requisiti d'elaborazione dei programmi, la modellazione deterministica è utilizzabile per scegliere un algoritmo di scheduling. Lo studio della modellazione deterministica su un insieme d'esempi può indicare tendenze che si possono poi analizzare e verificare separatamente. Si può per esempio mostrare che per l'ambiente descritto, vale a dire tutti i processi e i relativi tempi disponibili al tempo 0, con il criterio sjf si ottiene sempre il tempo d'attesa minimo.

5.8.2 Reti di code

In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi (o di tempi) da usare nella modellazione deterministica. Si possono però determinare le distribuzioni delle sequenze di operazioni della cpu e delle sequenze di operazioni di i/o, poiché si possono misurare e quindi approssimare, o più semplicemente stimare. Si ottiene una formula matematica che indica la probabilità di una determinata sequenza di operazioni della cpu. Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media. Analogamente, è possibile caratterizzare anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare la produttività media, l'utilizzo o il tempo d'attesa medi, e così via, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La cpu è un server con la propria ready queue, così come il sistema di i/o con le sue code di attesa dei dispositivi. Se sono noti le distribuzioni degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via. Questo tipo di studio si chiama analisi delle reti di code (*queueing-network analysis*).

Si consideri il seguente esempio: sia n la lunghezza media di una coda, escluso il processo correntemente servito, detti W il tempo medio d'attesa nella coda e l il tasso medio d'arrivo dei nuovi processi nella coda (per esempio, 3 processi al secondo); si prevede che, nel tempo W durante il quale un processo attende nella coda, raggiungano la coda $l \times W$ nuovi processi. Se il sistema è stabile, il numero dei processi che lasciano la coda deve essere uguale al numero dei processi che vi arrivano; quindi,

$$n = l \times W$$

Questa equazione è nota come formula di Little ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione degli arrivi. Per esempio n potrebbe essere il numero di clienti in un negozio.

La formula di Little è utilizzabile per il calcolo di una delle tre variabili, quando siano note le altre due. Per esempio, sapendo che ogni secondo arrivano 7 processi (in media), e che normalmente nella coda ne sono presenti 14, si può calcolare che il tempo medio d'attesa per ogni processo è di 2 secondi.

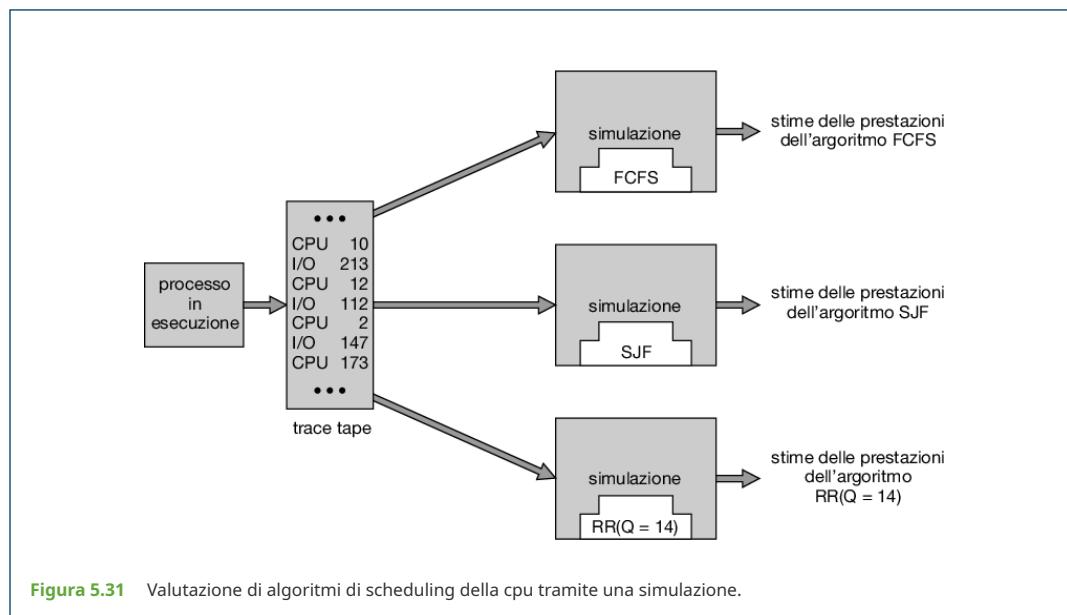
L'analisi delle reti di code può essere utile per il confronto degli algoritmi di scheduling, ma presenta alcuni limiti. Attualmente le classi di algoritmi e distribuzioni trattabili sono piuttosto limitate. Poiché può essere difficile lavorare matematicamente con distribuzioni e algoritmi complicati, spesso le distribuzioni d'arrivo e servizio vengono definite in maniera matematicamente trattabile, ma non realistica. Generalmente è necessario stabilire anche un numero di assunzioni indipendenti che possono non essere precise. Come risultato di queste difficoltà, le reti di code spesso si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

5.8.3 Simulazioni

Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di simulazioni. Le simulazioni implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema. Il simulatore dispone di una variabile che rappresenta un clock; all'incremento del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da tener conto delle attività dei dispositivi, dei processi e dello scheduler. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

I dati necessari per condurre la simulazione si possono ottenere in vari modi. Il metodo più diffuso impiega un generatore di numeri casuali, programmato per generare processi, durate dei burst della cpu, arrivi, partenze dal sistema e così via, in modo conforme alle rispettive distribuzioni di probabilità. Queste sono definibili matematicamente (esponenziali, uniformi, di Poisson) oppure in modo empirico. Se la distribuzione deve essere definita in modo empirico, si fanno misure sul sistema reale in esame, e si usano i risultati per definire la distribuzione effettiva degli eventi nel sistema reale; questa distribuzione è poi utilizzata per generare l'input della simulazione.

Tuttavia, una simulazione condotta sulla base delle distribuzioni degli eventi può non essere precisa, a causa delle relazioni esistenti tra eventi successivi nel sistema reale. La distribuzione delle frequenze, infatti, si limita a indicare quanti eventi di una data categoria si verificano, senza fornire informazioni sul loro ordine. Per rimediare a questo problema si può sottoporre il sistema reale a un monitoraggio, con la registrazione della sequenza degli eventi effettivi – in questo modo si ottiene un cosiddetto trace tape – (Figura 5.31), che poi si usa per condurre la simulazione. Si tratta di uno strumento eccellente che permette di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.



Poiché spesso richiedono diverse ore di tempo d'elaborazione, le simulazioni possono tuttavia essere molto onerose. Una simulazione più dettagliata dà risultati più precisi, ma richiede anche una maggiore quantità di tempo, e molto spazio di memoria per la registrazione degli eventi. Inoltre, la progettazione, la codifica e la messa a punto di un simulatore possono essere un compito assai impegnativo.

5.8.4 Realizzazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente preciso per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di

funzionamento del sistema.

Questo metodo non è privo di costi. Occorre tenere in conto la spesa sostenuta nella codifica dell'algoritmo e nella modifica del sistema operativo per supportarlo (comprese le strutture dati che richiede), ma vi è anche un costo nel testare le modifiche, di solito su macchine virtuali piuttosto che su hardware dedicato. I test di regressione sono in grado di confermare che le modifiche non abbiano peggiorato nulla e non abbiano causato nuovi bug o ricreato vecchi bug (per esempio perché l'algoritmo sostituito aveva risolto alcuni bug che la modifica ha riportato in vita).

Un'altra difficoltà da affrontare per la valutazione è il cambiamento dell'ambiente in cui si usa l'algoritmo. L'ambiente non cambia solo nel modo consueto, cioè per la scrittura di nuovi programmi e nuovi problemi che si possono riscontrare, ma si modifica anche in seguito alle prestazioni dello scheduler. Se si dà la priorità ai processi brevi, gli utenti possono suddividere i processi più lunghi in gruppi di processi brevi. Se si dà la priorità ai processi interattivi rispetto ai processi non interattivi, gli utenti possono passare all'uso interattivo. Questo problema viene solitamente affrontato utilizzando strumenti o script che incorporano serie complete di azioni, utilizzando tali strumenti ripetutamente e durante la misurazione dei risultati (e rilevando eventuali problemi causati nel nuovo ambiente).

Ovviamente il comportamento di un umano o di un programma può tentare di eludere gli algoritmi di scheduling. Per esempio, alcuni ricercatori progettaronno un sistema che classificava automaticamente i processi nelle categorie interattiva e non interattiva sulla base della quantità di i/o eseguita dal o verso il terminale. I processi che non leggevano o scrivevano sul terminale per un intero secondo erano classificati come non interattivi, e conseguentemente spostati in una coda a bassa priorità. Un programmatore reagì a questa strategia modificando i suoi programmi di modo che scrivessero sul terminale un carattere a intervalli regolari di meno di un secondo. Il risultato fu che le applicazioni del programmatore ricevettero alta priorità dal sistema, sebbene il loro output fosse del tutto privo di senso.

Gli algoritmi di scheduling più flessibili sono in generale quelli che possono essere tarati dagli amministratori del sistema o dagli utenti in modo da adattarsi a una specifica applicazione o insieme di applicazioni. Per esempio, le macchine impegnate in applicazioni grafiche sofisticate avranno necessità di scheduling diverse da quelle di un server web o di un file server. Alcuni sistemi operativi, e in particolare diverse versioni di unix, danno all'amministratore la possibilità di calibrare con precisione i parametri di scheduling a seconda delle particolari configurazioni del sistema. Solaris, per esempio, offre il comando `dispadmin` per la modifica dei parametri che regolano le classi di scheduling discusse nel Paragrafo 5.7.3.

Un altro approccio è di usare delle api appropriate per modificare la priorità di processi e thread. Le api Java, posix e Windows offrono tali funzionalità. Questa tecnica ha però lo svantaggio che tarare un sistema o un'applicazione per migliorarne le prestazioni in un caso specifico spesso non ha lo stesso risultato in situazioni più generali.

5.9 Sommario

- Lo scheduling della cpu consiste nella scelta di un processo dalla ready queue a cui assegnare la cpu. L'assegnazione della cpu al processo prescelto è eseguita dal dispatcher.
- Gli algoritmi di scheduling possono essere con prelazione (la cpu può essere sottratta a un processo) o senza prelazione (un processo deve rinunciare volontariamente al controllo della cpu). Quasi tutti i moderni sistemi operativi sono con prelazione.
- Gli algoritmi di scheduling possono essere valutati in base ai seguenti cinque criteri: (1) utilizzo della cpu, (2) throughput, (3) tempo di completamento, (4) tempo di attesa e (5) tempo di risposta.
- L'algoritmo di scheduling in ordine d'arrivo (fcfs) è il più semplice, ma può far sì che processi di breve durata attendano processi molto lunghi.
- Si dimostra che lo scheduling ottimale, che determina il minimo tempo medio d'attesa, è lo scheduling shortest-job-first (sjf). Realizzare lo scheduling sjf è complicato, poiché è difficile prevedere la lunghezza della successiva sequenza di operazioni della cpu.
- Lo scheduling circolare (rr) assegna la cpu a un processo per un quanto di tempo. Se entro quel tempo il processo non ha rilasciato la cpu avviene la prelazione e un altro processo viene mandato in esecuzione per un quanto di tempo.
- Lo scheduling con priorità assegna a ogni processo una priorità e la cpu viene allocata al processo con la priorità più alta. I processi con la stessa priorità possono essere schedulati nell'ordine fcfs o utilizzando lo scheduling rr.
- Lo scheduling a code multilivello partiziona i processi in code distinte in base alla loro priorità e lo scheduler esegue i processi nella coda con priorità più alta. All'interno di ogni coda possono essere utilizzati algoritmi di scheduling diversi.
- Le code multilivello con retroazione sono simili alle code multilivello, ma permettono a un processo di migrare tra diverse code.
- I processori multicore collocano una o più cpu sullo stesso chip fisico e ciascuna cpu può essere dotata di più di un thread hardware. Dal punto di vista del sistema operativo, ogni thread hardware appare come una cpu logica.
- Il bilanciamento del carico su sistemi multicore equilibra i carichi tra i core della cpu, sebbene la migrazione dei thread tra i core per bilanciare i carichi possa invalidare il contenuto della cache e quindi aumentare i tempi di accesso alla memoria.
- Lo scheduling real-time soft dà la priorità ai task real-time su quelli non real-time. Lo scheduling real-time hard fornisce garanzie temporali per i task real-time.
- L'algoritmo di scheduling con priorità proporzionale alla frequenza programma i task periodici applicando un modello statico di attribuzione delle priorità con prelazione.
- Lo scheduling edf (earliest-deadline-first) attribuisce le priorità sulla base delle scadenze. Più vicina è la scadenza, maggiore è la priorità; una scadenza più lontana implica una priorità più bassa.
- Lo scheduling a quote proporzionali opera distribuendo un certo numero di quote, diciamo T , fra tutte le applicazioni. Se un'applicazione riceve N quote di tempo, si assicura l'uso di una frazione N/T del tempo totale del processore.
- Linux utilizza lo scheduler cfs, che assegna una percentuale di tempo di elaborazione della cpu a ciascun task. La percentuale si basa sul tempo di esecuzione virtuale (`vruntime`) associato a ciascun task.
- Lo scheduling di Windows utilizza uno schema di priorità a 32 livelli con prelazione per determinare l'ordine di esecuzione dei thread.
- Solaris identifica sei classi di scheduling univoche che vengono associate a una priorità globale. I thread a uso intensivo di cpu sono generalmente assegnati a priorità più basse (e quanti di tempo più lunghi), mentre i thread i/o-bound sono solitamente assegnati a priorità più alte (con quanti di tempo più brevi).
- La modellazione e le simulazioni permettono di valutare un algoritmo di scheduling della cpu.

Esercizi di ripasso

5.1 Un algoritmo di scheduling della cpu stabilisce un ordine per l'esecuzione dei processi pianificati. Dati n processi da mandare in esecuzione su di un singolo processore, quanti differenti scheduling sono possibili? Date una formula in funzione di n .

5.2 Spiegate la differenza tra lo scheduling con e senza prelazione.

5.3 Supponete che i seguenti processi siano pronti per l'esecuzione agli istanti di tempo indicati nella tabella che segue. Nella tabella è anche indicata la durata della sequenza di operazioni per ogni processo. Nel rispondere alle domande seguenti utilizzate uno scheduling senza prelazione e basate le vostre decisioni sulle informazioni che avete a disposizione nel momento in cui la decisione deve essere presa.

Processo	Istante di arrivo	Durata della sequenza
P_1	0,0	8
P_2	0,4	4
P_3	1,0	1

- a. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling fcfs?
- b. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling sjf?
- c. L'algoritmo sjf dovrebbe migliorare le prestazioni, ma si noti che al tempo 0 viene scelto il processo P_1 , perché non si sa ancora che presto arriveranno due processi più brevi. Calcolate il tempo di completamento medio ipotizzando che la cpu venga lasciata inattiva per il primo istante di tempo e che successivamente venga utilizzato l'algoritmo sjf. Ricordate che i processi P_1 e P_2 restano in attesa durante il periodo di inattività, e quindi il loro tempo di attesa può aumentare. Questo algoritmo si potrebbe chiamare *scheduling con conoscenza del futuro*.

5.4 Considerate il seguente insieme di processi, con la durata della sequenza di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

- Presumiamo che i processi siano arrivati nell'ordine P_1, P_2, P_3, P_4, P_5 , e siano tutti presenti al tempo 0.
- a. Disegnate quattro diagrammi di Gantt che illustrino l'esecuzione di questi processi con gli algoritmi di scheduling fcfs, sjf, con priorità senza prelazione (un numero di priorità più alto indica una priorità maggiore) e rr (quanto = 2).
 - b. Calcolate il tempo di completamento di ciascun processo per ciascun algoritmo di scheduling di cui al punto a.
 - c. Calcolate il tempo d'attesa di ciascun processo per ciascun algoritmo di scheduling di cui al punto a.
 - d. Dite quale algoritmo ha il minimo tempo medio d'attesa (su tutti i processi).

5.5 I seguenti processi vengono pianificati utilizzando un algoritmo di scheduling round robin con prelazione.

Processo	Priorità	Durata della sequenza	Istante di arrivo
P_1	40	20	0
P_2	30	25	25
P_3	30	25	30
P_4	35	15	60
P_5	5	10	100
P_6	10	10	105

A ogni processo viene assegnato un valore numerico di priorità, dove un valore più grande indica una priorità relativa superiore. Oltre ai processi di seguito elencati, il sistema ha un task idle (che non consuma risorse di cpu e viene identificato come P_{idle}). Questo task ha priorità 0 e viene schedulato ogni volta che il sistema non ha altri processi disponibili per l'esecuzione. La lunghezza di un quanto di tempo è di 10 unità. Se un processo è prelazionato da un processo a priorità superiore viene posto alla fine della coda.

- a. Mostrate l'ordine di scheduling dei processi utilizzando un diagramma di Gantt.
- b. Calcolate il tempo di completamento di ciascun processo.
- c. Calcolate il tempo d'attesa di ciascun processo.
- d. Calcolate il tasso di utilizzo della cpu.

5.6 Quali vantaggi si hanno nell'avere quanti di tempo di dimensioni differenti a differenti livelli in un sistema di code multilivello?

5.7 Molti algoritmi di scheduling della cpu sono parametrici. L'algoritmo rr, per esempio, richiede un parametro che specifichi l'intervallo di tempo. Le code multilivello con retroazione richiedono parametri per specificare il numero di code, l'algoritmo di scheduling da utilizzare per ogni coda, il criterio usato per muovere i processi tra le code, e così via.

Questi algoritmi sono dunque classi di algoritmi (per esempio, la classe di algoritmi rr per tutti gli intervalli di tempo, e così via). Una classe di algoritmi ne può includere un'altra (per esempio, l'algoritmo fcfs è un algoritmo rr con intervallo di tempo infinito). Quale relazione intercorre (se esiste una relazione) tra le seguenti coppie di classi di algoritmi?

- a. Priorità e sjf.
- b. Code multilivello con retroazione e fcfs.
- c. Priorità ed fcfs.
- d. rr e sjf.

5.8 Supponete che un algoritmo di scheduling (a livello dello scheduling della cpu a breve termine) favorisca quei processi che hanno usato meno cpu in un passato recente. Spiegate perché questo algoritmo favorirà programmi con prevalenza di i/o senza bloccare permanentemente i programmi con prevalenza di elaborazione.

5.9 Individuate le differenze tra gli scheduling pcs e scs.

5.10 Lo scheduler tradizionale di unix stabilisce una relazione inversa tra numeri e priorità: più alto è il numero, minore è la priorità. Lo scheduler ricalcola le priorità dei processi una volta al secondo utilizzando la seguente funzione:

$$\text{Priorità} = (\text{utilizzo recente della cpu} / 2) + \text{base}$$

dove base = 60 utilizzo e *utilizzo recente della cpu* è un valore che indica in che misura un processo ha utilizzato la cpu dall'ultima volta in cui le priorità sono state ricalcolate.

Si supponga che l' utilizzo recente della cpu sia 40 per il processo P_1 , 18 per il processo P_2 e 10 per il processo P_3 . Quali saranno le nuove priorità di questi tre processi dopo il ricalcolo? Sulla base di questa informazione, dite se lo scheduler tradizionale di unix alza o abbassa la priorità relativa di un processo cpu-bound.

Esercizi

5.11 Tra questi due tipi di programmi

- a. i/o-bound
- b. cpu-bound

con quale è più probabilmente che avvenga un cambio di contesto volontario e con quale un cambio di contesto involontario?
Motivate la risposta.

5.12 Considerate come le seguenti coppie di criteri per lo scheduling entrino in conflitto in certe situazioni:

- a. utilizzo della cpu e tempo di risposta;
- b. tempo di completamento medio e tempo di attesa massimo;
- c. utilizzo dei dispositivi di i/o e utilizzo della cpu.

5.13 Una tecnica per l'implementazione dello scheduling a lotteria consiste nell'assegnare ai processi dei biglietti della lotteria che vengono utilizzati per l'attribuzione del tempo di cpu. Ogni volta che si deve prendere una decisione riguardo allo scheduling viene sorteggiato casualmente un biglietto della lotteria e il processo che ne è in possesso ottiene la cpu. Il sistema operativo btv implementa lo scheduling a lotteria effettuando 50 sorteggi ogni secondo e assegnando al vincitore 20 millisecondi di tempo di cpu (20×50 millisecondi = 1 secondo). Descrivete in che modo lo scheduler btv può assicurare che i thread con priorità più alta ricevano più attenzione dalla cpu rispetto a quelli a bassa priorità.

5.14 La maggior parte degli algoritmi di scheduling mantiene una coda di esecuzione, che elenca i processi ammissibili per l'esecuzione su un processore. Sui sistemi multicore vi sono due opzioni: (1) ogni core di elaborazione ha la sua coda di esecuzione, oppure (2) una sola coda di esecuzione è condivisa tra tutti i core. Descrivete vantaggi e svantaggi di ciascuno di questi approcci.

5.15 Considerate la formula della media esponenziale atta a predire la durata della sequenza successiva della cpu. Quali implicazioni scaturiscono dall'assegnazione dei seguenti valori ai parametri usati dall'algoritmo?

- a. $a = 0$ e $t_0 = 100$ millisecondi
- b. $a = 0,99$ e $t_0 = 10$ millisecondi

5.16 Una variante dello scheduler round-robin è lo scheduler round-robin regressivo. Questo scheduler assegna a ogni processo un quanto di tempo e una priorità. Il valore iniziale di un quanto di tempo è pari a 50 millisecondi. Ogni volta che un processo viene assegnato alla cpu e usa interamente il suo quanto di tempo (non bloccandosi per i/o), vengono aggiunti 10 millisecondi al suo quanto e il suo livello di priorità viene aumentato. Il quanto di tempo di un processo può essere aumentato fino a un massimo di 100 millisecondi. Quando un processo si blocca prima di utilizzare per intero il suo tempo, il suo quanto viene ridotto di 5 millisecondi e la sua priorità rimane la stessa. Quale tipo di processo (cpu-bound o i/o-bound) viene favorito dallo scheduler round-robin regressivo? Giustificate la vostra risposta.

Processo	Durata della sequenza	Priorità
P_1	5	2
P_2	3	1
P_3	1	2
P_4	7	2
P_5	4	3

5.17 Considerate il seguente set di processi, con la lunghezza della cpu burst espressa in millisecondi:

Si presume che i processi siano giunti nell'ordine P_1, P_2, P_3, P_4, P_5 , tutti a tempo 0.

Processo	Priorità	Burst	Arrivo
P_1	8	15	0
P_2	3	20	0
P_3	4	20	20
P_4	4	20	25
P_5	5	5	45
P_6	5	15	55

- a. Disegnate quattro diagrammi di Gantt che illustrino l'esecuzione di questi processi con gli algoritmi di scheduling fcfs, sjf, con priorità senza prelazione (un numero di priorità più alto indica una priorità maggiore) e rr (quanto = 2).
- b. Calcolate il tempo di completamento di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- c. Calcolate il tempo d'attesa di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- d. Dite quale algoritmo ha il minimo tempo medio d'attesa (su tutti i processi).

5.18 I seguenti processi sono stati programmati usando un algoritmo di scheduling senza prelazione, con prelazione, round-robin.

A ciascun processo viene assegnata una priorità numerica, dove un numero più alto indica una relativa priorità maggiore. Lo scheduler eseguirà il processo con la priorità più alta. Per i processi che hanno la stessa priorità sarà utilizzato uno scheduler round-robin con un quanto di tempo di 10 unità. Se un processo è soggetto a prelazione da un processo con priorità più alta il processo soggetto a prelazione è collocato alla fine della coda.

- a. Mostrate l'ordine di schedulino del processo basandovi su un diagramma di Gantt.
- b. Qual è il tempo di completamento (*turnaround*) di ciascun processo?
- c. Qual è il tempo di attesa di ciascun processo?

5.19 Il comando `nice` viene usato per impostare il nice value di un processo su Linux, oltre che su altri sistemi unix. Spiegate perché alcuni sistemi permettono a qualsiasi utente di assegnare a un processo un nice value ≥ 0 , mentre consentono solo all'utente root di assegnare un nice value < 0 .

5.20 Quale tra i seguenti algoritmi di scheduling potrebbe generare un'attesa indefinita?

- a. In ordine di arrivo (fcfs).
- b. Per brevità (sjf).
- c. Circolare (rr).
- d. Per priorità.

5.21 Data una variante dell'algoritmo di scheduling rr in cui gli elementi della ready queue sono puntatori ai pcb:

- a. descrivete l'effetto dell'inserimento di due puntatori allo stesso processo nella ready queue;
- b. descrivete due vantaggi e due svantaggi di questo schema;
- c. ipotizzate una modifica all'algoritmo rr ordinario che consenta di ottenere lo stesso effetto senza duplicare i puntatori.

5.22 Considerate un sistema su cui vengano eseguiti dieci processi con prevalenza di i/o e un processo con prevalenza di elaborazione. Supponiamo che i primi richiedano un'operazione di i/o ogni millisecondo di elaborazione della cpu, e che ciascuna di tali operazioni sia completata in 10 millisecondi. Ipotizzate, inoltre, che il tempo necessario per il cambio di contesto sia di 0,1 millisecondi e che tutti i processi siano attivi per un lungo periodo. Calcolate l'utilizzo della cpu in presenza di scheduler circolare rr se:

- a. il quanto di tempo è pari a 1 millisecondo;
- b. il quanto di tempo è pari a 10 millisecondi.

5.23 Considerate un sistema che applichi un algoritmo di scheduling a code multilivello. A quale strategia può ricorrere l'utente che voglia ottenere per un suo processo la massima quantità di tempo dalla cpu?

5.24 Considerate un algoritmo di scheduling a priorità con prelazione, basato su priorità variabili dinamicamente. I numeri di priorità maggiori indicano una priorità più alta. Quando un processo attende la cpu (nella ready queue), la sua priorità varia a un tasso a ; quando è in esecuzione, la sua priorità varia a un tasso b . All'ingresso nella ready queue, si attribuisce la priorità 0 a tutti i processi. I parametri a e b si possono impostare in modo da fornire algoritmi di scheduling diversi.

- a. Descrivete l'algoritmo risultante da $b > a > 0$.
- b. Descrivete l'algoritmo risultante da $a < b < 0$.

5.25 Spiegate quanto i seguenti algoritmi di scheduling gestiscono preferenzialmente i processi di breve durata:

- a. in ordine di arrivo (fcfs);
- b. circolare (rr);
- c. a code multilivello con retroazione.

5.26 Descrivete perché una ready queue condivisa potrebbe presentare problemi di prestazioni in un ambiente smp.

5.27 Considerate un algoritmo con bilanciamento del carico che assicuri che ciascuna coda abbia approssimativamente lo stesso numero di thread, indipendentemente dalla priorità. Come si comporterebbe in pratica un algoritmo di scheduling basato sulla priorità se una coda di esecuzione avesse tutti thread ad alta priorità e una seconda coda avesse tutti thread a bassa priorità?

5.28 Supponete che un sistema smp disponga di code private di esecuzione, una per ogni processore. Quando viene creato un nuovo processo può essere collocato o nella coda uguale a quella del task genitore oppure in una coda separata.

- a. Quai sono i vantaggi di collocare il nuovo processo nella coda uguale a quella del suo genitore?
- b. Quali sono i vantaggi di collocare il nuovo processo in una coda differente?

5.29 Supponiamo che un thread bloccato per i/o di rete torni idoneo per l'esecuzione. Spiegate perché un algoritmo di scheduling numa-aware dovrebbe ripianificare il thread sulla stessa cpu su cui è stato eseguito in precedenza.

5.30 Usando l'algoritmo di scheduling di Windows, quale priorità numerica è assegnata, nei casi che seguono a:

- a. Un thread appartenente alla `realtime_priority_class` con una priorità relativa normal.
- b. Un thread appartenente alla `above_normal_priority_class` con una priorità relativa highest.
- c. Un thread appartenente alla `below_normal_priority_class` con una priorità relativa `above_normal`.

5.31 Supponendo che nessun thread appartenga alla `realtime_priority_class` e che a nessuno possa essere assegnata una priorità `time_critical`, quale combinazione classe di priorità/priorità corrisponde al più alto livello possibile di priorità relativa nello scheduling Windows?

5.32 Considerate l'algoritmo di scheduling del sistema operativo Solaris per i thread time-sharing.

- a. Qual è il quanto di tempo (in millisecondi) per un thread con priorità 15? E per uno con priorità 40?
- b. Ipotizzate che un thread con priorità 50 abbia consumato l'intera porzione di tempo riservatagli senza bloccarsi. Quale sarà la nuova priorità assegnata dallo scheduler a questo thread?
- c. Poniamo che un thread con priorità 20 si blocchi per l'i/o prima che la propria porzione di tempo sia finita. Qual è la nuova priorità che lo scheduler assegnerà a questo thread?

5.33 Supponiamo che due task A e B siano in esecuzione su un sistema Linux. I nice value di A e B sono rispettivamente -5 e 5. Utilizzando lo scheduler cfs come guida, descrivete come variano i valori di `vruntime` dei due processi in ciascuno dei seguenti scenari:

- A e B sono cpu-bound.
- A è i/o-bound e B è cpu-bound.
- A è cpu-bound e B è i/o-bound.

5.34 In quali circostanze lo scheduling con priorità proporzionale alla frequenza si comporta peggio dello scheduling edf nel venire incontro alle scadenze associate ai processi?

5.35 Si considerino due processi P_1 e P_2 dove $P_1 = 50$, $t_1 = 25$, $p_2 = 75$ e $t_2 = 30$.

- a. È possibile schedularle questi due processi utilizzando lo scheduling a con priorità proporzionale alla frequenza? Illustrate la vostra risposta utilizzando un diagramma di Gantt come quelli mostrati nelle Figure 5.21-5.24.
- b. Illustrate lo scheduling di questi due processi utilizzando lo scheduling edf.

5.36 Spiegate perché nei sistemi real-time hard i tempi di latenza di interrupt e dispatch devono essere limitati.

5.37 Descrivete i vantaggi di utilizzare il multiprocessing eterogeneo in un sistema mobile.

Questo progetto prevede l'implementazione di diversi algoritmi di scheduling dei processi. Allo scheduler verrà assegnato un insieme predefinito di task che verranno pianificati in base all'algoritmo di scheduling selezionato. A ogni task è assegnata una priorità e una durata della sequenza di operazioni della cpu (*burst*). Saranno realizzati i seguenti algoritmi di scheduling.

- First-come, first-served (fcfs), che pianifica i task nell'ordine in cui richiedono la cpu.
- Shortest-job-first (sjf), che pianifica i task in base alla lunghezza del prossimo burst della cpu.
- Scheduling a priorità, che pianifica i task in base alla priorità.
- Scheduling round-robin (rr), in cui ogni task viene eseguito per un quanto di tempo (o per il resto del suo burst della cpu).
- Priorità round-robin, che pianifica i task in ordine di priorità e usa lo scheduling round-robin per task con uguale priorità.

Le priorità vanno da 1 a 10, dove un valore numerico più alto indica un valore più alto della priorità relativa. Per la schedulazione round-robin, la lunghezza di un quanto di tempo è 10 millisecondi.

I. Implementazione

L'implementazione di questo progetto può essere effettuata in C o Java, e file di supporto in entrambi i linguaggi sono forniti nel codice scaricabile con questo testo. Questi file di supporto leggono nella lista dei task, inseriscono i task in una lista e richiamano lo scheduler.

La lista dei task schedulabili ha la forma `[nome task] [priorità] [burst CPU]`, con il seguente formato di esempio:

- T1, 4, 20
- T2, 2, 25
- T3, 3, 25
- T4, 3, 15
- T5, 10, 10

Pertanto, il task T1 ha priorità 4 e un burst della cpu di 20 millisecondi, e così via. Si presume che tutti i task arrivino nello stesso momento, quindi gli algoritmi dello scheduler non devono supportare processi con priorità più elevata che effettuano la prelazione di processi con priorità inferiori. Inoltre, i task non devono essere inseriti in una coda o in una lista in un ordine particolare.

Esistono diverse strategie per organizzare la lista dei task, come presentato nel Paragrafo 5.1.2. Un approccio è quello di inserire tutti i task in un singolo elenco non ordinato, in cui la strategia per la selezione del task dipende dall'algoritmo di scheduling. Per esempio, lo scheduling sjf scandirà l'elenco per trovare l'attività con il più breve successivo burst della cpu. In alternativa, la lista potrebbe essere ordinata in base ai criteri di scheduling (cioè, per priorità). Un'altra strategia comporta avere una coda separata per ciascun livello di priorità, come mostrato nella Figura 5.7. Questi approcci sono brevemente discussi nel Paragrafo 5.3.6. Vale anche la pena evidenziare che usiamo i termini *lista* e *coda* in maniera intercambiabile. Tuttavia, una coda ha funzionalità fifo molto specifiche, mentre una lista non ha requisiti di inserimento e cancellazione rigorosi. Probabilmente troverete più adatte le funzionalità di una lista generica generale per realizzare questo progetto.

II. Dettagli di implementazione in C

Il file `driver.c` legge l'elenco dei task, inserisce ogni task in una lista linkata e richiama lo scheduler dei processi chiamando la funzione `schedule()`. La funzione `schedule()` esegue ciascun task in base all'algoritmo di scheduling specificato. I task selezionati per l'esecuzione sulla cpu sono determinati dalla funzione `pick-NextTask()` e vengono eseguiti richiamando la funzione `run()` definita nel file `CPU.c`. Viene utilizzato un `Makefile` per determinare l'algoritmo di scheduling specifico che verrà invocato da `driver`. Per esempio, per creare lo scheduler fcfs, scriveremmo

```
make fcfs
```

ed eseguiremmo lo scheduler (usando l'elenco dei task `schedule.txt`) come segue:

```
./fcfs schedule.txt
```

Fate riferimento al file `README` nel codice sorgente scaricabile per ulteriori dettagli. Prima di procedere, assicuratevi di familiarizzare con il codice sorgente fornito così come col `Makefile`.

III. Dettagli di implementazione in Java

Il file `Driver.java` legge nell'elenco dei task, inserisce ogni attività in una `ArrayList` Java e richiama lo scheduler dei processi invocando in metodo `schedule()`. La seguente interfaccia identifica un algoritmo di scheduling generico, che i cinque diversi algoritmi di scheduling implemetteranno:

```
public interface Algorithm
{
    // Implementazione dell'algoritmo di scheduling

    public void schedule();

    // Seleziona il prossimo task da pianificare

    public Task pickNextTask();
}
```

Il metodo `schedule()` ottiene l'attività successiva da eseguire sulla cpu invocando il metodo `pickNextTask()` e quindi esegue questa operazione chiamando il metodo `static run()` nella classe `CPU.java`. Il programma viene eseguito come segue:

```
java Driver fcfs schedule.txt
```

Fate riferimento al file `README` nel codice sorgente scaricabile per ulteriori dettagli. Prima di procedere, assicuratevi di familiarizzare con tutti i file sorgente Java forniti nel codice sorgente scaricabile.

Ulteriori sfide

Per questo progetto vengono presentate le seguenti due ulteriori sfide.

1. A ciascun task fornito allo scheduler viene assegnata un identificativo unico (`tid`). Se uno scheduler è in esecuzione in un ambiente smp in cui ogni cpu esegue separatamente il proprio programma di scheduling, esiste una possibile condizione di corsa critica sulla variabile che viene utilizzata per assegnare gli identificatori di task. Correggere questa condizione utilizzando un intero atomico.

Su sistemi Linux e macos la funzione `_sync_fetch_and_add()` può essere usata per incrementare atomicamente un valore intero. Per esempio, il seguente codice incrementa atomicamente il valore di 1:

```
int value = 0;
```

```
_sync_fetch_and_add (&value, 1);
```

Fate riferimento all'api Java per i dettagli su come utilizzare la classe `AtomicInteger` per i programmi Java.

2. Calcolate il tempo medio di completamento, il tempo di attesa e il tempo di risposta per ciascuno degli algoritmi di scheduling.

CAPITOLO 6

Strumenti di sincronizzazione

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso fili o messaggi. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. In questo capitolo si trattano vari meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti, che condividono uno spazio logico di indirizzi, così da mantenere la coerenza dei dati.

6.1 Introduzione

Abbiamo già visto che i processi possono essere eseguiti in modo concorrente o in parallelo. Il Paragrafo 3.2.2 ha introdotto il ruolo dello scheduling dei processi e ha descritto come lo scheduler della cpu passi rapidamente da un processo all'altro per offrire un'esecuzione concorrente. Questo significa che un processo può avere completato solo in parte la sua esecuzione quando viene schedulato un altro processo. In effetti, un processo può essere interrotto in qualsiasi punto del proprio flusso d'esecuzione, assegnando il core di elaborazione all'esecuzione di istruzioni di un altro processo. Il Paragrafo 4.2 ha inoltre introdotto l'esecuzione parallela, in cui due flussi di istruzioni (che rappresentano processi differenti) vengono eseguiti contemporaneamente su core distinti. In questo capitolo viene spiegato come l'esecuzione concorrente o parallela possa contribuire a problematiche che riguardano l'integrità dei dati condivisi da più processi.

Prendiamo in considerazione un esempio di come ciò può accadere. Nel Capitolo 3 è stato descritto un modello di sistema costituito da un certo numero di processi sequenziali cooperanti o thread, tutti in esecuzione asincrona e con la possibilità di condividere dati. Tale modello è stato illustrato attraverso l'esempio del produttore/consumatore, che ben rappresenta molte situazioni che riguardano i sistemi operativi. Nel Paragrafo 3.5, in particolare, si è descritto come un buffer limitato sia utilizzabile per permettere ai processi la condivisione della memoria.

Torniamo al concetto di buffer limitato. Com'è stato sottolineato, la nostra soluzione consentiva la presenza contemporanea di non più di `BUFFER_SIZE - 1` elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera, `contatore`, inizializzata a 0, che si incrementa ogniqualvolta s'inserisce un nuovo elemento nel buffer e si decrementa ogniqualvolta si preleva un elemento dal buffer. Il codice per il processo produttore si può modificare come segue:

```
while (true) {

    /* produce un elemento in next_produced */

    while (contatore == BUFFER_SIZE)

        ; /* non fa niente */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;

    contatore++;

}
```

Il codice per il processo consumatore si può modificare come segue:

```
while (true) {

    while (contatore == 0)

        ; /* non fa niente */

    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    contatore--;

    /* consuma un elemento in next_consumed */

}
```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile `contatore` sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni `contatore++` e `contatore--` in modo concorrente.

Terminata l'esecuzione delle due istruzioni, il valore della variabile `contatore` potrebbe essere 4, 5 o 6! Il solo risultato corretto è `contatore == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Si può dimostrare che il valore di contatore può essere scorretto: l'istruzione `contatore++` si può codificare in un tipico linguaggio macchina, come

```
registro1 := contatore
registro1 := registro1 + 1
contatore := registro1
```

dove `registro1` è un registro locale della cpu. Analogamente, l'istruzione `contatore--` si può codificare come

```
registro2 := contatore
registro2 := registro2 - 1
contatore := registro2
```

dove `registro2` è un registro locale della cpu. Anche se `registro1` e `registro2` possono essere lo stesso registro fisico, per esempio un accumulatore, occorre ricordare che il contenuto di questo registro viene salvato e recuperato dal gestore dei segnali d'interruzione (Paragrafo 1.2.3).

L'esecuzione concorrente delle istruzioni `contatore++` e `contatore--` equivale a un'esecuzione sequenziale delle istruzioni del linguaggio macchina introdotte precedentemente, intercalate (*interleaved*) in una qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione di alto livello. Una di queste sequenze è

$T_0: \text{produttore}$	esegue	<code>registro1 := contatore</code>	{ <code>registro1 = 5</code> }
$T_1: \text{produttore}$	esegue	<code>registro1 := registro1 + 1</code>	{ <code>registro1 = 6</code> }
$T_2: \text{consumatore}$	esegue	<code>registro2 := contatore</code>	{ <code>registro2 = 5</code> }
$T_3: \text{consumatore}$	esegue	<code>registro2 := registro2 - 1</code>	{ <code>registro2 = 4</code> }
$T_4: \text{produttore}$	esegue	<code>contatore := registro1</code>	{ <code>contatore = 6</code> }
$T_5: \text{consumatore}$	esegue	<code>contatore := registro2</code>	{ <code>contatore = 4</code> }

e conduce al risultato errato in cui `contatore == 4`; si registra la presenza di 4 elementi nel buffer, mentre in realtà gli elementi sono 5. Se si invertisse l'ordine delle istruzioni in T_4 e T_5 si giungerebbe allo stato errato in cui `contatore == 6`.

Si è arrivati a questo stato non corretto perché si è permesso a entrambi i processi di manipolare concorrentemente la variabile `contatore`. Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette race condition) occorre assicurare che un solo processo alla volta possa modificare la variabile `contatore`. Questa garanzia richiede una forma di sincronizzazione dei processi.

Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti del sistema compiono operazioni su risorse condivise. Inoltre, come evidenziato nei precedenti capitoli, la crescente importanza dei sistemi multicore ha dato maggior enfasi allo sviluppo di applicazioni multithread in cui diversi thread, che probabilmente possono condividere dei dati, sono in esecuzione in parallelo su core distinti. Ovviamente tali operazioni non devono interferire reciprocamente in modi indesiderati. Data l'importanza della questione, la maggior parte di questo capitolo è dedicata ai problemi della sincronizzazione e coordinamento dei processi.

6.2 Problema della sezione critica

Iniziamo la nostra discussione sui processi di sincronizzazione illustrando il problema della sezione critica. Si consideri un sistema composto di n processi $\{P_0, P_1, \dots, P_{n-1}\}$ ciascuno avendo un segmento di codice, chiamato sezione critica (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. La caratteristica fondamentale del sistema è che, quando un processo è in esecuzione nella propria sezione critica, non si consente ad alcun altro processo di essere in esecuzione nella propria sezione critica. Il problema della *sezione critica* consiste nel progettare un protocollo che i processi possano usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la sezione d'ingresso. La sezione critica può essere seguita da una sezione d'uscita, e la restante parte del codice è detta sezione non critica. La Figura 6.1 mostra la struttura generale di un tipico processo. La sezione d'ingresso e quella d'uscita sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

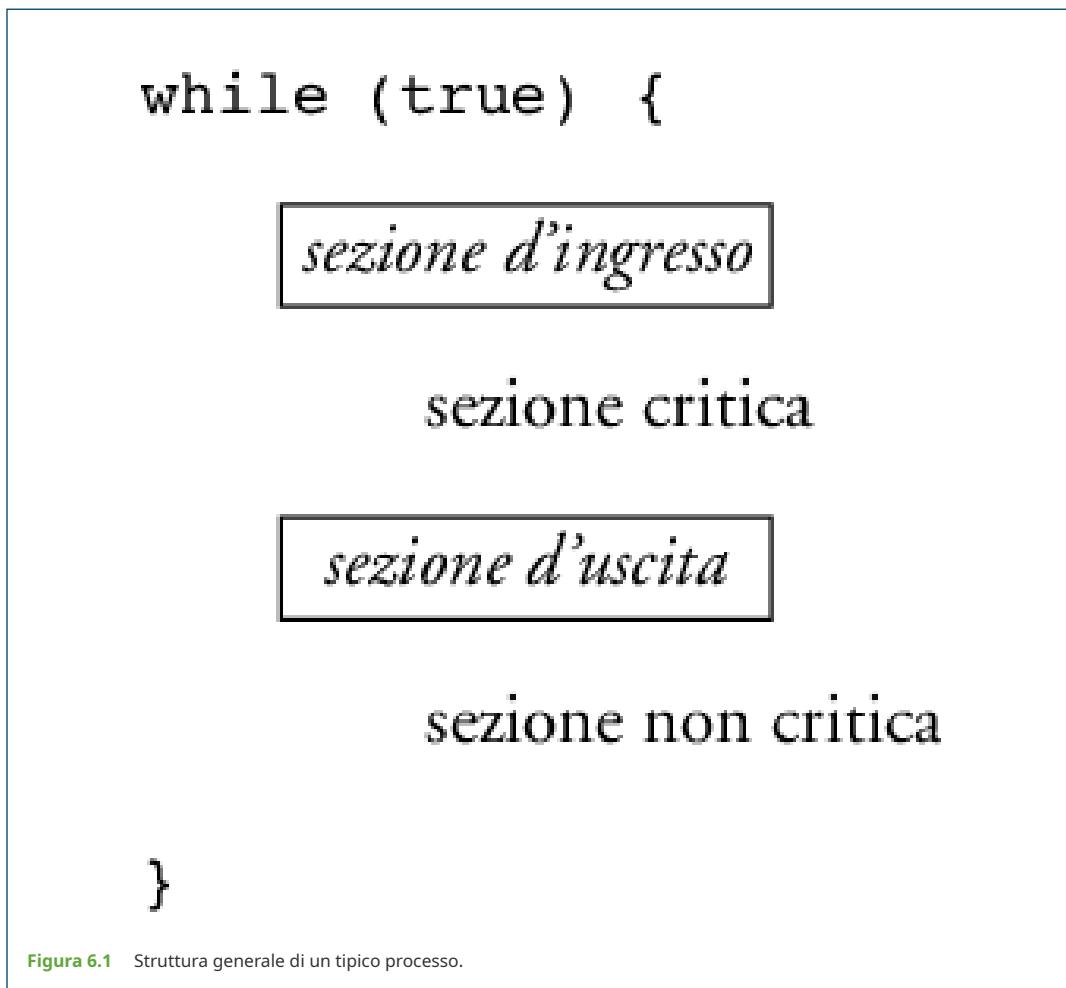


Figura 6.1 Struttura generale di un tipico processo.

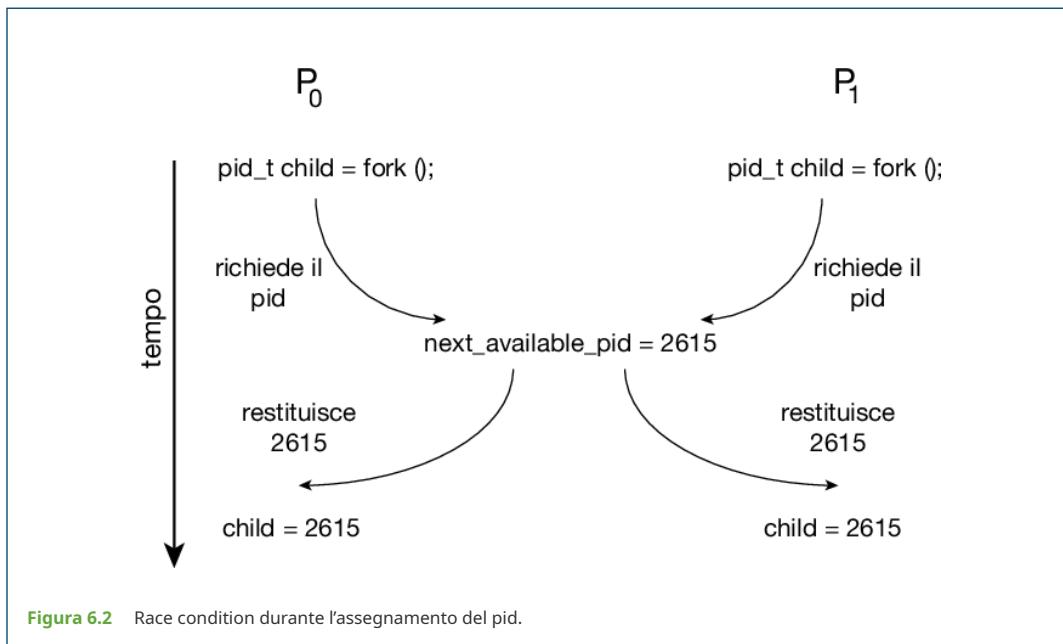
Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

1. Mutua esclusione. Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. Progresso. Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori dalle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
3. Attesa limitata. Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla velocità relativa degli n processi.

In un dato momento, numerosi processi in modalità kernel possono essere attivi nel sistema operativo. Se ciò si verifica, il codice del kernel, che implementa il sistema operativo, è soggetto a molte possibili race condition. Si consideri per esempio una struttura dati del kernel che mantenga una lista di tutti i file aperti nel sistema. Tale lista deve essere modificata quando un nuovo file è aperto, e quindi aggiunto all'elenco, oppure chiuso, e quindi tolto dall'elenco. Se due o più processi dovessero aprire dei file simultaneamente, potrebbero ingenerare nel sistema una race condition legata ai necessari aggiornamenti della lista.

Un altro esempio è illustrato nella Figura 6.2. In questa situazione, due processi P_0 e P_1 creano processi figlio (child) utilizzando la chiamata di sistema `fork()`. Ricordiamo dal Paragrafo 3.3.1 che `fork()` restituisce al processo genitore l'identificatore di processo (pid) del processo appena creato. In questo esempio, c'è una race condition sulla variabile del kernel `next_available_pid`, che contiene il valore del prossimo pid disponibile. In assenza di mutua esclusione è possibile che lo stesso pid venga assegnato a due processi distinti.



Altre strutture dati del kernel soggette a problemi analoghi sono quelle per l'allocazione della memoria, per la gestione delle interruzioni e le liste dei processi. La responsabilità di preservare il sistema operativo da simili problemi compete a chi sviluppa il kernel.

Il problema della sezione critica può essere risolto in maniera semplice in un ambiente single-core, impedendo che si verifichino interruzioni durante la modifica di una variabile condivisa. In questo modo, saremmo sicuri che l'attuale sequenza di istruzioni sia eseguita in ordine, senza prelazione. Visto che non verranno eseguite altre istruzioni sarà impossibile apportare modifiche inaspettate a una variabile condivisa.

Sfortunatamente questa soluzione non è praticabile in un ambiente multiprocessore. Disabilitare gli interrupt su un multiprocessore può richiedere infatti molto tempo, poiché il messaggio viene passato a tutti i processori. Questo invio di messaggi ritarda l'ingresso in ciascuna sezione critica e l'efficienza del sistema diminuisce. Va inoltre tenuto in considerazione l'effetto sul clock di sistema nel caso in cui il clock venga aggiornato dalle interruzioni.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi sono: kernel con diritto di prelazione e kernel senza diritto di prelazione. Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della cpu. Ovviamente, i kernel senza diritto di prelazione sono immuni da race condition sulle strutture dati del kernel, visto che un solo processo per volta impegnava il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi nell'accesso alle strutture dati del kernel. I kernel con diritto di prelazione presentano particolari difficoltà di progettazione quando sono destinati ad architetture smp, poiché in tali ambienti due processi nella modalità di sistema possono essere eseguiti in contemporanea su core differenti.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione? I kernel con diritto di prelazione possono vantare una maggior prontezza nelle risposte, grazie al basso rischio di eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la cpu per i processi in attesa (certamente il rischio può essere minimizzato anche progettando codice kernel che non si comporta in questo modo). Inoltre, i kernel con diritto di prelazione sono più adatti alla

programmazione real-time, dal momento che permettono ai processi in tempo reale di effettuare la prelazione di un processo attivo nel kernel. Vedremo più avanti come diversi sistemi operativi usino la prelazione all'interno del kernel.

6.3 Soluzione di Peterson

Illustriamo adesso una classica soluzione software al problema della sezione critica, nota come soluzione di Peterson. A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali `load` e `store`, non è affatto certo che la soluzione di Peterson funzioni correttamente su tali sistemi. Tuttavia si è scelto di presentarla ugualmente perché rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune complessità legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

La soluzione di Peterson è limitata a due processi, P_0 e P_1 , ognuno dei quali esegue alternativamente la propria sezione critica e la sezione non critica. Per il seguito, è utile convenire che se P_i denota uno dei due processi, P_j denoti l'altro; ossia, che $j = 1 - i$.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

```
int turn;

boolean flag[2];
```

La variabile `turn` segnala, per l'appunto, di chi sia il turno d'accesso alla sezione critica; quindi, se `turn == i`, il processo P_i è autorizzato a eseguire la propria sezione critica. L'array `flag`, invece, indica se un processo *sia pronto* a entrare nella sezione critica. Per esempio, se `flag[i]` è `true`, P_i è pronto a entrare nella propria sezione critica. Chiarito il ruolo delle strutture dati, possiamo ora analizzare l'algoritmo descritto nella Figura 6.3.

```
while (true) {

    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        /*sezione critica*/

    flag[i] = false;

        /*sezione non critica*/

}
```

Figura 6.3 Struttura del processo P_i nella soluzione di Peterson.

Per accedere alla sezione critica, il processo P_i assegna innanzitutto a `flag[i]` il valore `true`; quindi attribuisce a `turn` il valore j , conferendo così all'altro processo la facoltà di entrare nella sezione critica.

Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a turno sia il valore i sia il valore j . Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo di `turn` stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica.

Dimostriamo ora la correttezza di questa soluzione. Dobbiamo provare che:

1. la mutua esclusione sia preservata;
2. il requisito del progresso sia soddisfatto;
3. il requisito dell'attesa limitata sia rispettato.

Per dimostrare la proprietà 1, si osservi come ogni P_i acceda alla propria sezione critica solo se `flag[j] == false` oppure `turn == i`. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora `flag[0] == flag[1] == true`. Si desume da queste due osservazioni che P_0 e P_1 sono impossibilitati a eseguire con successo le rispettive istruzioni `while` approssimativamente nello stesso momento: `turn`, infatti, può valere 0 o 1, ma non entrambi. Pertanto, uno dei processi – poniamo P_j – deve aver eseguito con successo l'istruzione `while`, mentre P_i aveva da eseguire almeno un'istruzione aggiuntiva ("`turn == j`"). Tuttavia, poiché in quel momento, e fino al termine della permanenza di P_j nella propria sezione critica, restano valide le asserzioni `flag[j] == true` e `turn == j`, ne consegue che la mutua esclusione è preservata.

Per dimostrare le proprietà 2 e 3, osserviamo come l'ingresso di un processo P_i nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione `while`, con le condizioni `flag[j] == true` e `turn == j`; questa è l'unica possibilità. Qualora P_j non sia pronto a entrare nella sezione critica, `flag[j] == false`, e P_i può accedere alla propria sezione critica. Se P_j ha impostato `flag[j]` a `true` e sta eseguendo il proprio ciclo `while`, `turn == i`, oppure `turn == j`. Se `turn == i`, P_i entrerà nella propria sezione critica. Se `turn == j`, P_j entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica, P_j reimposta `flag[j]` a `false`, consentendo a P_i di entrarvi. Se P_j reimposta `flag[j]` a `true`, deve anche attribuire alla variabile `turn` il valore i . Poiché tuttavia P_i non modifica il valore della variabile `turn` durante l'esecuzione dell'istruzione `while`, P_i entrerà nella sezione critica (progresso) dopo che P_j abbia effettuato non più di un ingresso (attesa limitata).

Come accennato all'inizio di questo paragrafo, non è possibile garantire il funzionamento della soluzione di Peterson su architetture moderne, principalmente perché per migliorare le prestazioni dei sistemi i processori e/o i compilatori possono riordinare operazioni di lettura e scrittura che non hanno dipendenze. Per un'applicazione con un singolo thread questo riordino è irrilevante per quanto riguarda la correttezza del programma, in quanto i valori finali rimangono coerenti con quanto previsto (come avviene nel calcolo del saldo di un conto corrente: l'ordine effettivo in cui vengono eseguite le operazioni di credito e debito non è importante, perché il saldo finale sarà sempre lo stesso), ma per un'applicazione multithread con dati condivisi il riordino delle istruzioni può portare a risultati incoerenti o inattesi.

Si considerino, per esempio, i seguenti dati condivisi tra due thread:

```
boolean flag = false;

int x = 0;
```

Thread 1 esegue le istruzioni

```
while (!flag)

;

print x;
```

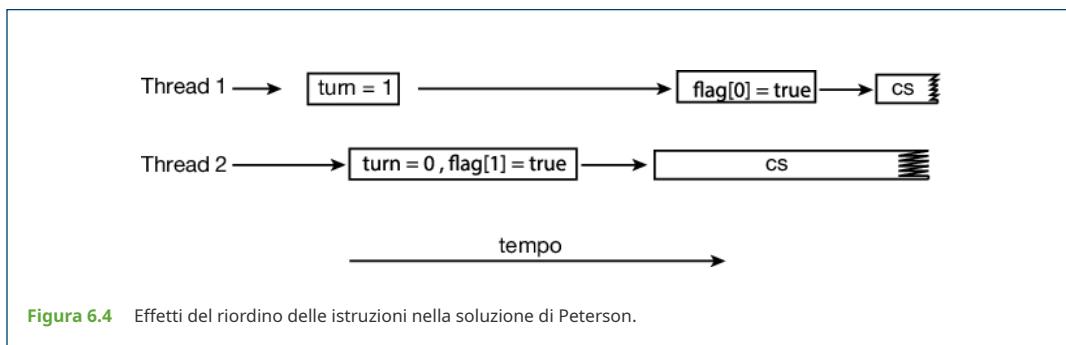
e Thread 2 esegue

```
x = 100;

flag = true;
```

Il comportamento previsto è, ovviamente, che Thread 1 restituisca il valore 100 per la variabile `x`. Tuttavia, poiché non ci sono dipendenze tra le variabili `flag` e `x`, è possibile che un processore riordini le istruzioni di Thread 2 in modo che il `flag` sia impostato a `true` prima dell'assegnamento `x = 100`. In questa situazione Thread 1 potrebbe restituire il valore 0 per la variabile `x`. Meno ovvio è che il processore possa anche riordinare le istruzioni eseguite da Thread 1 e caricare la variabile `x` prima del valore di `flag`. Se ciò accadesse, Thread 1 restituirebbe il valore 0 anche se le istruzioni eseguite da Thread 2 non fossero state riordinate.

In che modo questo influenza la soluzione di Peterson? Si consideri ciò che accadrebbe se gli assegnamenti che compaiono nella sezione d'ingresso della soluzione di Peterson della Figura 6.3 venissero riordinati. In questo caso sarebbe possibile avere entrambi i thread attivi nelle loro sezioni critiche contemporaneamente, come mostrato nella Figura 6.4.



Come si vedrà nei seguenti paragrafi, l'unico modo per preservare la mutua esclusione è utilizzare strumenti di sincronizzazione adeguati. La nostra analisi di questi strumenti inizierà dal supporto hardware e continuerà con le api astratte di alto livello che sono disponibili per gli sviluppatori di kernel e di applicazioni.

6.4 Supporto hardware per la sincronizzazione

Abbiamo appena descritto una soluzione software al problema della sezione critica. Parliamo in questo caso di *soluzione basata sul software*, perché l'algoritmo garantisce la mutua esclusione senza richiedere alcun supporto speciale dal sistema operativo né istruzioni hardware specifiche. Tuttavia, come già detto, soluzioni basate sul software come quella di Peterson non garantiscono il loro funzionamento su architetture elaborate moderne. In questo paragrafo presentiamo tre istruzioni hardware che forniscono supporto per risolvere il problema della sezione critica. Queste primitive possono essere utilizzate direttamente come strumenti di sincronizzazione, oppure come basi per costruire meccanismi di sincronizzazione più astratti.

6.4.1 Barriere di memoria

Nel Paragrafo 6.3 abbiamo visto che un sistema può riordinare le istruzioni e che questa politica può portare a uno stato dei dati inaffidabile. Il modello di memoria è il modo in cui l'architettura di un computer determina quali garanzie relative alla memoria vengono fornite a un programma applicativo. In generale, un modello di memoria rientra in una delle seguenti categorie.

1. Fortemente ordinato, in cui una modifica alla memoria su un processore è immediatamente visibile a tutti gli altri processori.
2. Debolmente ordinato, in cui le modifiche alla memoria su un processore potrebbero non essere immediatamente visibili agli altri processori.

I modelli di memoria variano in base al tipo di processore, quindi gli sviluppatori del kernel non possono formulare ipotesi sulla visibilità delle modifiche alla memoria su un multiprocessore a memoria condivisa. Per risolvere questo problema, le architetture dei computer forniscono istruzioni che possono forzare la propagazione di qualsiasi modifica in memoria a tutti i processori, garantendo in tal modo che le modifiche siano visibili ai thread in esecuzione su altri processori. Tali istruzioni sono note come barriere di memoria (*memory barrier*) o recinzioni di memoria (*memory fence*). Quando viene eseguita un'istruzione di barriera di memoria, il sistema garantisce che tutte le istruzioni di load e store siano completate prima che vengano eseguite le successive operazioni di load e store. Pertanto, anche se le istruzioni sono state riordinate, la barriera di memoria assicura che le operazioni di store siano completate e visibili ad altri processori prima che vengano eseguite le operazioni di load e store future.

Ritorniamo al nostro ultimo esempio, in cui il riordino delle istruzioni poteva produrre un risultato errato, e utilizziamo una barriera di memoria per garantire l'output atteso.

Se aggiungiamo un'operazione di barriera di memoria al Thread 1

```
while (!flag)  
  
    memory_barrier();  
  
    print_x;
```

garantiamo che il valore di flag sia caricato prima del valore di x.

Allo stesso modo, se poniamo una barriera di memoria tra gli assegnamenti presenti in Thread 2

```
x = 100;  
  
memory_barrier();  
  
flag = true;
```

ci assicuriamo che l'assegnamento a x avvenga prima dell'assegnamento a flag.

Per quanto riguarda la soluzione di Peterson, potremmo inserire una barriera di memoria tra le prime due istruzioni di assegnamento della sezione d'ingresso per evitare il riordino delle operazioni mostrato nella Figura 6.4. Si noti che le barriere di memoria sono considerate operazioni di livello molto basso e sono in genere utilizzate solo dagli sviluppatori del kernel quando scrivono codice specializzato per garantire la mutua esclusione.

6.4.2 Istruzioni hardware

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo atomico – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice. Anziché

discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali descrivendo le istruzioni `test_and_set()` e `compare_and_swap()`.

L'istruzione `test_and_set()` si può definire come nella Figura 6.5. La caratteristica fondamentale di questa istruzione è che viene eseguita atomicamente; quindi, se si eseguono contemporaneamente due istruzioni `test_and_set()`, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. Se si dispone dell'istruzione `test_and_set()`, si può realizzare la mutua esclusione dichiarando una variabile booleana globale `lock`, inizializzata a `false`. La struttura del processo P_i è illustrata nella Figura 6.6.

```
boolean test_and_set(boolean *obiettivo){  
    boolean valore = *obiettivo;  
    *obiettivo = true;  
  
    return valore;  
}
```

Figura 6.5 Definizione dell'istruzione atomica `test_and_set()`.

```
do {  
    while (test_and_set(&lock));  
    /*non fa niente*/  
  
    /*sezione critica*/  
  
    lock = false;  
  
    /*sezione non critica*/  
} while (true);
```

Figura 6.6 Realizzazione di mutua esclusione con `test_and_set()`.

L'istruzione `compare_and_swap()` (cas), proprio come l'istruzione `test_and_set()`, opera su due parole atomicamente, ma utilizza un meccanismo diverso che si basa sullo scambio del contenuto delle parole.

La cas, definita nella Figura 6.7, utilizza tre operandi. L'operando `value` viene impostato a `new_value` solo se l'espressione `(*value == expected)` è vera. A parte in questo caso, la `compare_and_swap()` restituisce sempre il valore originale della variabile `value`. L'importante caratteristica di questa istruzione è che viene eseguita atomicamente. Pertanto, se due istruzioni cas vengono eseguite simultaneamente (ciascuna su un core diverso), verranno eseguite sequenzialmente in un ordine arbitrario.

```

int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}

```

Figura 6.7 Definizione dell'istruzione atomica `compare_and_swap()`.

La mutua esclusione può essere realizzata usando cas come segue. Viene dichiarata e inizializzata a 0 una variabile globale (`lock`). Il primo processo che richiama `compare_and_swap()` imposterà `lock` a 1. Entrerà poi nella sua sezione critica, poiché il valore originale di `lock` era pari al valore atteso 0. Le chiamate successive di `compare_and_swap()` non avranno successo, perché ora `lock` non è uguale al valore atteso 0. Quando un processo esce dalla sezione critica, imposta di nuovo `lock` al valore 0, per permettere a un altro processo di entrare nella propria sezione critica. La struttura del processo P_i è illustrata nella Figura 6.8.

```

while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* non fa niente */

    /* sezione critica */

    lock = 0;

    /* sezione non critica */
}

```

Figura 6.8 Realizzazione di mutua esclusione con `compare_and_swap()`.

Questo algoritmo soddisfa il requisito della mutua esclusione, ma non quello dell'attesa limitata. La Figura 6.9 mostra un altro algoritmo che sfrutta l'istruzione `compare_and_swap()` che soddisfare tutti e tre i requisiti desiderati. Le strutture dati condivise sono

```

while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    /* sezione critica */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = 0;
    else
        waiting[j] = false;

    /* sezione non critica */
}

```

Figura 6.9 Mutua esclusione con attesa limitata con compare_and_swap().

```
boolean waiting[n];
```

```
boolean lock;
```

Gli elementi nell'array `waiting` vengono inizializzati a `false` e `lock` viene inizializzato a 0. Per dimostrare che l'algoritmo soddisfa il requisito di mutua esclusione, si noti che il processo P_i può entrare nella propria sezione critica solo se `waiting[i] == false` oppure `key == 0`. Il valore di `key` può diventare 0 solo se si esegue `compare_and_swap()`. Il primo processo che esegue `compare_and_swap()` trova `key == 0`; tutti gli altri devono attendere. La variabile `waiting[i]` può diventare `false` solo se un altro processo esce dalla propria sezione critica; solo una variabile `waiting[i]` vale `false`, il che consente di rispettare il requisito di mutua esclusione.

Per dimostrare che l'algoritmo soddisfa il requisito di progresso, basta osservare che le argomentazioni fatte per la mutua esclusione valgono anche in questo caso; infatti un processo che esce dalla sezione critica imposta `lock` al valore `false` oppure `waiting[j]` al valore `false`; entrambe consentono a un processo in attesa l'ingresso nella propria sezione critica.

Per dimostrare che l'algoritmo soddisfa il requisito di attesa limitata occorre osservare che un processo, quando lascia la propria sezione critica, scandisce il vettore `waiting` in ordine ciclico ($i+1, i+2, \dots, n-1, 0, \dots, i-1$) e designa il primo processo in questo ordinamento presente nella sezione d'ingresso (`waiting[j] == true`) come il primo processo che deve entrare nella propria sezione critica. Qualsiasi processo che attende l'ingresso nella propria sezione critica può farlo entro $n-1$ turni.

Una descrizione dettagliata dell'implementazione delle istruzioni atomiche `test_and_set()` e `compare_and_swap()` è reperibile nei testi di architettura dei calcolatori.

RENDERE ATOMICA LA COMPARE-AND-SWAP

Per implementare l'istruzione `compare_and_swap()` nelle architetture Intel x86 si utilizza l'istruzione assembly `cmpxchg`. Per forzare l'esecuzione atomica viene utilizzato il prefisso `lock`, che consente di bloccare il bus mentre viene aggiornato l'operando di destinazione. L'istruzione `cmpxchg` nella sua forma generale si presenta come segue:

```
lock cmpxchg <destination operand>, <source operand>
```

6.4.3 Variabili atomiche

In genere, l'istruzione `compare_and_swap()` non viene utilizzata direttamente per fornire la mutua esclusione, ma serve piuttosto da elemento base per la costruzione di altri strumenti che risolvono il problema della sezione critica. Uno di questi strumenti è la variabile atomica, che fornisce operazioni atomiche su tipi di dati di base come interi e booleani. Abbiamo visto nel Paragrafo 6.1 che l'incremento o il decremento di un valore intero può produrre una race condition. Le variabili atomiche possono essere utilizzate per garantire la mutua esclusione in situazioni in cui potrebbe esserci una race condition su una singola variabile durante il suo aggiornamento, come quando un contatore viene incrementato.

La maggior parte dei sistemi che supportano le variabili atomiche fornisce speciali tipi di dati atomici e funzioni per l'accesso e la manipolazione di variabili di tali tipi. Queste funzioni sono spesso implementate usando operazioni `compare_and_swap()`. Per esempio, il seguente codice incrementa l'intero atomico `sequence`:

```
increment(&sequence);
```

La funzione `increment()` è implementata usando l'istruzione `cas`:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != compare_and_swap(v, temp, temp+1));
}
```

È importante notare che, sebbene le variabili atomiche forniscono aggiornamenti atomici, non risolvono interamente le race condition in tutte le circostanze. Per esempio, nel problema del buffer limitato descritto nel Paragrafo 6.1 si potrebbe usare un numero intero atomico per il conteggio e ciò garantirebbe l'atomicità degli aggiornamenti del contatore. Tuttavia, i processi produttore e consumatore contengono anche cicli `while` la cui condizione dipende dal valore di `contatore`. Si consideri la situazione in cui il buffer è vuoto e due consumatori stanno effettuando il ciclo in attesa che si verifichi la condizione `contatore > 0`. Se un produttore inserisce un elemento nel buffer, entrambi i consumatori potrebbero uscire dai loro cicli `while` (in quanto il `contatore` non sarebbe più uguale a 0) e procedere al consumo, anche se il valore di `contatore` era impostato su 1.

Le variabili atomiche sono comunemente utilizzate nei sistemi operativi e nelle applicazioni concorrenti, sebbene il loro uso sia spesso limitato a singoli aggiornamenti di dati condivisi come contatori e generatori di sequenze. Nei seguenti paragrafi esploreremo

strumenti più robusti che risolvono le race condition in situazioni più generali.

6.5 Lock mutex

Le soluzioni hardware al problema della sezione critica presentate nel Paragrafo 6.4 sono complicate e generalmente inaccessibili ai programmati di applicazioni. In alternativa, i progettisti di sistemi operativi implementano strumenti software per risolvere lo stesso problema. Il più semplice di questi strumenti è il lock mutex (il termine *mutex* è in realtà l'abbreviazione di *mutual exclusion*, cioè mutua esclusione). Usiamo il lock mutex per proteggere le regioni critiche e quindi prevenire le race condition. In pratica un processo deve acquisire il lock prima di entrare in una sezione critica e rilasciarlo quando esce dalla sezione critica. La funzione `acquire()` acquisisce il lock e la funzione `release()` lo rilascia, come illustrato nella Figura 6.10.

```
while (true) {  
    acquisisci lock  
    sezione critica  
    rilascia lock  
    sezione non critica  
}
```

Figura 6.10 Soluzione al problema della sezione critica con lock mutex.

Un lock mutex ha una variabile booleana `available` il cui valore indica se il lock è disponibile o meno. Se il lock è disponibile la chiamata di `acquire()` ha successo e il lock viene da questo momento considerato non disponibile. Un processo che tenta di acquisire un lock indisponibile viene bloccato fino al rilascio del lock.

CONTESA DEI LOCK

I lock possono essere contesi oppure no. Un lock è considerato conteso se un thread si blocca mentre tenta di acquisire il lock. Se un lock è disponibile quando un thread tenta di acquisirlo, il lock è considerato non conteso. I lock contesi

possono essere soggetti a una contesa elevata (un numero relativamente grande di thread che tentano di acquisire il lock) o a una contesa moderata (un numero relativamente piccolo di thread che tenta di acquisire il lock). Come ci si può aspettare, i lock molto contesi tendono a ridurre le prestazioni complessive delle applicazioni concorrenti.

La definizione della funzione `acquire()` è la seguente:

```
acquire() {  
  
    while (!available)  
  
        ; /* attesa attiva */  
  
    available = false;  
  
}
```

La definizione di `release()` è la seguente:

```
release() {  
  
    available = true;  
  
}
```

Le chiamate alle funzioni `acquire()` e `release()` devono essere eseguite atomicamente. Per questa ragione i lock mutex sono spesso realizzati utilizzando uno dei meccanismi hardware descritti nel Paragrafo 6.4. Lasciamo la descrizione di questa tecnica come esercizio.

CHE COSA SI INTENDE PER "BREVE PERIODO"?

Gli spinlock vengono spesso identificati come il meccanismo di locking da preferire sui sistemi multiprocessore quando il lock deve essere mantenuto per un breve periodo. Ma che cosa si intende esattamente per un breve periodo? Dato che l'attesa su un lock richiede due cambi di contesto, uno per spostare il thread allo stato di attesa e un altro per ripristinare il thread in attesa una volta che il lock diventa disponibile, la regola generale è di usare uno spinlock quando il lock deve essere trattenuto per una durata inferiore a due cambi di contesto.

Il principale svantaggio dell'implementazione che abbiamo fornito è che richiede attesa attiva (*busy waiting*). Mentre un processo si trova nella sua sezione critica, ogni altro processo che cerchi di entrare nella sezione critica deve ciclare continuamente effettuando la chiamata `acquire()`. Questo continue ciclare è chiaramente un problema in un reale sistema multiprogrammato in cui un singolo core della cpu è condiviso tra molti processi. L'attesa attiva spreca inoltre cicli di cpu che altri processi potrebbero essere in grado di utilizzare in modo produttivo. (Nel Paragrafo 6.6 esamineremo una strategia che evita l'attesa attiva sospendendo temporaneamente il processo in attesa e quindi risvegliandolo una volta che il lock diventi disponibile).

Il tipo di lock mutex che abbiamo descritto è anche chiamato spinlock, perché il processo continua a "girare" (spin) in attesa che il lock diventi disponibile. (Osserviamo lo stesso comportamento negli esempi di codice che illustrano l'istruzione `compare_and_swap()`). Tuttavia, gli spinlock hanno il vantaggio di non rendere necessario alcun cambio di contesto (operazione che può richiedere molto tempo) quando un processo deve attendere un lock e tornano quindi utili quando si prevede che i lock verranno trattenuti per

tempi brevi. Gli spinlock sono spesso impiegati in sistemi multiprocessore in cui un thread può “girare” su un processore, mentre un altro thread esegue la sua sezione critica su un altro processore.

Nel Capitolo 7 esamineremo come i lock mutex possano essere utilizzati per risolvere classici problemi di sincronizzazione. Discuteremo anche di come questi lock siano usati in diversi sistemi operativi e in Pthreads.

6.6 Semafori

Come abbiamo accennato in precedenza, i lock mutex sono generalmente considerati il più semplice degli strumenti di sincronizzazione. In questo paragrafo esaminiamo uno strumento più robusto in grado di comportarsi in modo simile a un lock mutex, ma capace anche di fornire metodi più complessi per la sincronizzazione delle attività dei processi.

Un semaforo `s` è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`. Queste operazioni erano originariamente chiamate `P` (per `wait()`; dall'olandese *proberen*, verificare) e `V` (per `signal()`; da *verhogen*, incrementare). La definizione di `wait()` in pseudocodice è la seguente:

```
wait(s) {  
    while(s <= 0)  
  
        ;//attesa attiva  
  
    s--;  
}
```

La definizione di `signal()` in pseudocodice è la seguente:

```
signal(s) {  
    s++;  
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre, nel caso della `wait(s)` si devono eseguire senza interruzione anche la verifica del valore intero di `s` (`s ≤ 0`) e la sua possibile modifica (`s--`). Nel Paragrafo 6.6.2 si spiega come si possano realizzare queste operazioni. Ora vediamo come si possano utilizzare i semafori.

6.6.1 Uso dei semafori

Si usa distinguere tra semafori contatore, il cui valore è un numero intero, e i semafori binari, il cui valore è limitato a 0 o 1. I semafori binari sono dunque simili ai lock mutex e vengono utilizzati al loro posto per la mutua esclusione nei sistemi dove i lock mutex non sono disponibili.

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di risorse disponibili. I processi che desiderino utilizzare una risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono una risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, tutte le risorse sono occupate, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: P_1 con un'istruzione S_1 e P_2 con un'istruzione S_2 . Si supponga di voler eseguire S_2 solo dopo che S_1 è terminata. Questo schema si può prontamente realizzare facendo condividere a P_1 e P_2 un semaforo comune, `synch`, inizializzato a 0, e inserendo nel processo P_1 le istruzioni

```
S1;
```

```
signal(synch);
```

e nel processo P_2 le istruzioni

```
wait(synch);  
S2;
```

Poiché `synch` è inizializzato a 0, P_2 esegue S_2 solo dopo che P_1 ha eseguito `signal(synch)`, che si trova dopo S_1 .

6.6.2 Implementazione dei semafori

Ricordiamo che la realizzazione dei lock mutex trattati nel Paragrafo 6.5 presentava il problema dell'attesa attiva. Le definizioni delle operazioni sui semafori `wait()` e `signal()` appena descritte presentano lo stesso problema.

Per superare la necessità dell'attesa attiva si possono modificare le definizioni delle operazioni `wait()` e `signal()` come segue: quando un processo invoca l'operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può *bloccare* se stesso. L'operazione di *bloccaggio* pone il processo in una coda d'attesa associata al semaforo e pone lo stato del processo a *waiting*. Quindi, si trasferisce il controllo allo scheduler della cpu che sceglie un altro processo pronto per l'esecuzione.

Un processo sospeso, che attende a un semaforo s , sarà riavviato in seguito all'esecuzione di un'operazione `signal()` su s da parte di qualche altro processo. Il processo si riavvia tramite un'operazione `wakeup()`, che modifica lo stato del processo da *waiting* a *ready*. Il processo entra nella coda dei processi pronti. (L'uso della cpu può essere o non essere commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda del criterio di scheduling).

Per realizzare i semafori secondo quel che s'è detto si può definire il semaforo come segue:

```
typedef struct {  
  
    int value;  
  
    struct process *list;  
  
} semaphore;
```

A ogni semaforo sono associati un valore intero (`value`) e una lista di processi (`list`), contenente i processi in attesa a un semaforo; l'operazione `signal()` preleva un processo da tale lista e lo attiva.

L'operazione `wait()` del semaforo si può definire come segue:

```
wait(semaphore *S) {  
  
    S->value--;  
  
    if (S->value < 0) {  
  
        aggiungi questo processo a S->list;  
  
        sleep();  
    }  
}
```

L'operazione `signal()` del semaforo si può definire come segue:

```
signal(semaphore *S) {
    S->value++;

    if (S->value <= 0) {
        togli un processo P da S->list;

        wakeup(P);
    }
}
```

L'operazione `sleep()` sospende il processo che la invoca; l'operazione `wakeup(P)` pone in stato di pronto per l'esecuzione un processo `P` bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, il suo valore assoluto rappresenta il numero dei processi che attendono a quel semaforo. Ciò è dovuto all'inversione dell'ordine del decremento e della verifica nel codice dell'operazione `wait()`.

La lista dei processi che attendono a un semaforo si può facilmente realizzare con un campo puntatore in ciascun blocco di controllo del processo (pcb). Ogni semaforo contiene un valore intero e un puntatore a una lista di pcb. Un modo per aggiungere e togliere processi dalla lista assicurando un'attesa limitata è usare una coda fifo, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale tuttavia si può usare *qualsiasi* criterio d'accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

Come già abbiamo detto, le operazioni sui semafori devono essere eseguite in modo atomico. Si deve garantire che nessuna coppia di processi possa eseguire operazioni `wait()` e `signal()` contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocesso lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di `signal()` e `wait()`. Nei sistemi con una sola cpu, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interruzioni, dando la possibilità allo scheduler di riprendere il controllo della cpu, il processo corrente continua indisturbato la sua esecuzione.

Nei sistemi multicore sarebbe necessario disabilitare le interruzioni di tutti i core, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, disabilitare le interruzioni di tutti i processori può essere complesso, e causare un notevole calo delle prestazioni. È per questo che – per garantire l'esecuzione atomica di `wait()` e `signal()` – i sistemi smp devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, `compare_and_swap()` e gli spinlock).

È importante rilevare che questa definizione delle operazioni `wait()` e `signal()` non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi. Inoltre, l'attesa attiva si limita alle sezioni critiche delle operazioni `wait()` e `signal()`, che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l'attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l'attesa attiva è assai inefficiente.

6.7 Monitor

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre.

Un esempio di tali errori si è visto nell'utilizzo dei contatori della nostra soluzione al problema produttore/consumatore (Paragrafo 6.1). In quella circostanza, il problema di sincronizzazione appariva solo sporadicamente, e anche il valore del contatore si manteneva entro limiti ragionevoli, essendo sfasato tutt'al più di 1. Ciononostante, le soluzioni di questo tipo restano inaccettabili, ed è per ottenere soluzioni soddisfacenti che sono stati inventati i semafori.

Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo `mutex`, inizializzata a 1. Ogni processo deve eseguire `wait(mutex)` prima di entrare nella sezione critica e `signal(mutex)` al momento di uscirne. Se questa sequenza non è rispettata, può accadere che due processi occupino simultaneamente le rispettive sezioni critiche. Esaminiamo le difficoltà che possono insorgere. Si noti che tali difficoltà possono insorgere anche nel caso che *un solo* processo si comporti in maniera non corretta. L'inconveniente può nascere da un involontario errore di programmazione o essere causato dalla mancata collaborazione del programmatore.

- Supponiamo che un processo capovolga l'ordine in cui sono eseguite le istruzioni `wait()` e `signal()`, in questo modo:

```
signal(mutex);

...
sezione critica
...
wait(mutex);
```

In questa situazione, numerosi processi possono eseguire le proprie sezioni critiche allo stesso tempo, violando il requisito della mutua esclusione. Questo errore può essere scoperto solo qualora diversi processi siano attivi simultaneamente nelle rispettive sezioni critiche. Si osservi che tale situazione potrebbe non essere sempre riproducibile.

- Ipotizziamo che un processo sostituisca `signal(mutex)` con `wait(mutex)`, cioè che esegua

```
wait(mutex);

...
sezione critica
...
wait(mutex);
```

In questo caso, il processo si bloccherà in modo permanente alla seconda chiamata `wait()`, poiché il semaforo non è più disponibile.

- Si supponga che un processo ometta `wait(mutex)`, `signal(mutex)`, o entrambi. In questo caso si viola la mutua esclusione oppure il processo si blocca indefinitamente.

Questi esempi chiariscono come sia facile incorrere in errori allorché i programmatori utilizzino i semafori in maniera scorretta nel risolvere il problema delle sezioni critiche. Una strategia per gestire tali errori consiste nell'incorporare semplici strumenti di sincronizzazione in costrutti linguistici di alto livello. In questo paragrafo descriviamo un fondamentale costrutto di sincronizzazione di alto livello, il tipo monitor.

6.7.1 Uso del costrutto monitor

Un tipo di dato astratto (adt) incapsula i dati mettendo a disposizione un insieme di funzioni per operare su di essi; tali funzioni sono indipendenti dalla specifica implementazione del tipo di dato. Il monitor è un adt che comprende un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il *tipo monitor* contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre al corpo delle procedure o funzioni che operano su tali variabili. La sintassi di un monitor è mostrata nella Figura 6.11. La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una funzione definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

```
monitor monitor name

{
    /* dichiarazione di variabili condivise */

    function P1 ( . . . ) {

        . . .

    }

    function P2 ( . . . ) {

        . . .

    }

    . . .

    function Pn ( . . . ) {

        . . .

    }

    initialization_code ( . . . ) {

        . . .

    }
}
```

```
}
```

Figura 6.11 Sintassi di un monitor in pseudocodice.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione (Figura 6.12). Tuttavia la definizione di monitor presentata finora non è abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi forniti dal costrutto `condition`. Un programmatore che necessita di implementare un proprio particolare schema di sincronizzazione può definire una o più variabili di tipo `condition`:

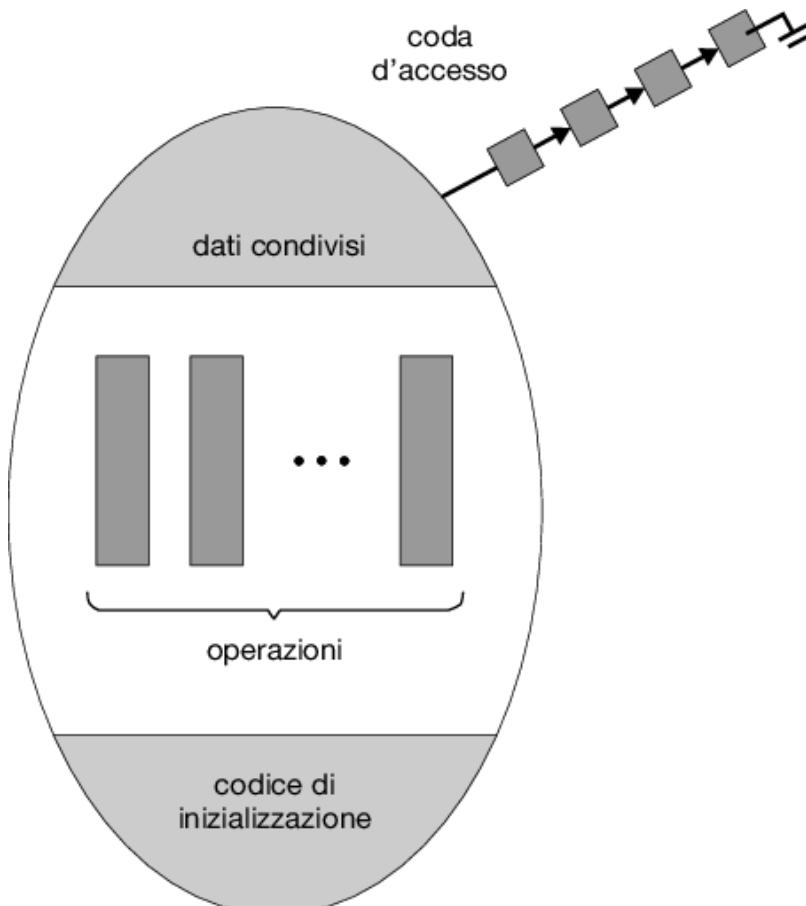


Figura 6.12 Schema di un monitor.

```
condition x, y;
```

Le uniche operazioni eseguibili su una variabile `condition` sono `wait()` e `signal()`. L'operazione

```
x.wait();
```

implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione

```
x.signal();
```

che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione `signal()` non ha alcun effetto, vale a dire che lo stato di `x` resta immutato, come se l'operazione non fosse stata eseguita; la situazione è descritta nella Figura 6.13. Tutto ciò contrasta con l'operazione `signal()` associata ai semafori, poiché questa influisce sempre sullo stato del semaforo.

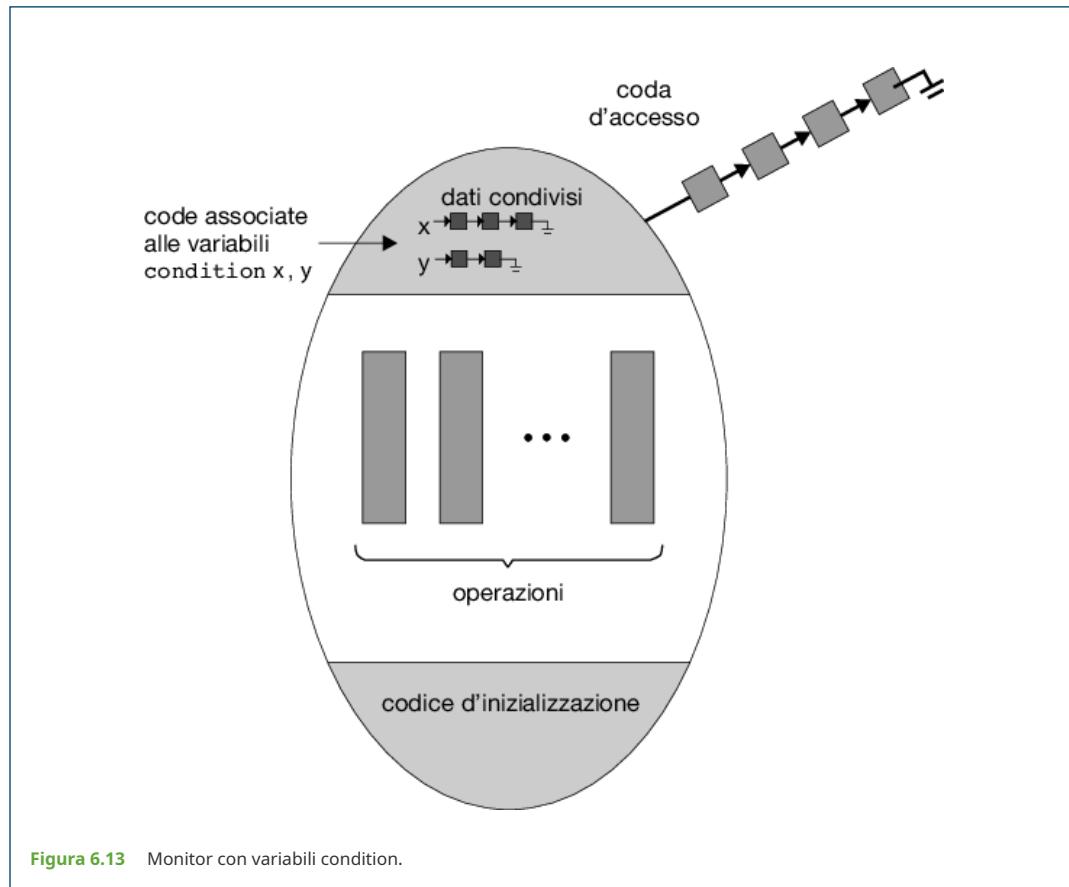


Figura 6.13 Monitor con variabili condition.

Si supponga, per esempio, che quando un processo P invoca l'operazione `x.signal()`, esista un processo sospeso Q associato alla variabile `x` di tipo `condition`. Chiaramente, se al processo sospeso Q si permette di riprendere l'esecuzione, il processo segnalante P è costretto ad attendere, altrimenti P e Q sarebbero contemporaneamente attivi all'interno del monitor. Occorre tuttavia notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono due possibilità:

1. segnalare e attendere. P attende che Q lasci il monitor o attenda su un'altra variabile `condition`;
2. segnalare e proseguire. Q attende che P lasci il monitor o attenda su un'altra variabile `condition`.

Si possono fornire argomenti ragionevoli a favore dell'uno o dell'altra opzione. Da un lato, visto che P era già in esecuzione all'interno del monitor, il secondo metodo appare più ragionevole. D'altro canto, se si lascia proseguire il thread P , la condizione attesa da Q potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione. Il linguaggio Concurrent Pascal ha scelto

un compromesso: quando il thread P esegue l'operazione `signal()`, lascia subito il monitor; pertanto, Q riprende immediatamente l'esecuzione.

Molti linguaggi di programmazione incorporano l'idea di monitor descritta in questo paragrafo. Tra questi vi sono Java e C# (da leggersi "C-sharp"). Altri linguaggi, come Erlang, forniscono alcune tipologie di supporto alla concorrenza usando un meccanismo simile.

6.7.2 Realizzazione di un monitor per mezzo di semafori

A questo punto si considera una possibile realizzazione del meccanismo del monitor usando i semafori. A ogni monitor si associa un semaforo `mutex`, inizializzato a 1, per garantire la mutua esclusione; un processo deve eseguire `wait(mutex)` prima di entrare nel monitor, e `signal(mutex)` dopo aver lasciato il monitor.

Useremo nella nostra implementazione lo schema signal-and-wait. Poiché un processo che esegue una `signal()` deve attendere finché il processo risvegliato si metta in attesa o lasci il monitor, si introduce un altro semaforo, `next`, inizializzato a 0, su cui i processi che eseguono una `signal()` possono autosospendersi. Per contare i processi sospesi al semaforo `next`, si usa una variabile intera `next_count`. Quindi, ogni procedura esterna `F` si sostituisce col seguente codice:

```
wait(mutex);
...
corpo di F
```

...

```
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

In questo modo si assicura la mutua esclusione all'interno del monitor.

A questo punto si può descrivere la realizzazione delle variabili `condition`. Per ogni variabile `x` di tipo `condition` si introducono un semaforo `x_sem` e una variabile intera `x_count`, entrambi inizializzati a 0. L'operazione `x.wait()` si può realizzare come segue:

```
x_count++;
if (next_count > 0)
    signal(next);

else
    signal(mutex);
    wait(x_sem);
    x_count--;
```

L'operazione `x.signal()` si può realizzare come segue:

```

if (x_count > 0) {

    next_count++;

    signal(x_sem);

    wait(next);

    next_count--;

}

```

Questa soluzione è applicabile alle definizioni di monitor date sia da Hoare sia da Brinch-Hansen (si vedano le note bibliografiche alla fine del capitolo). In alcuni casi, tuttavia, questo livello di generalità della codifica non è necessario, e si possono apportare notevoli miglioramenti all'efficienza. La soluzione a questo problema è lasciata al lettore nell'Esercizio 6.16.

6.7.3 Ripresa dei processi all'interno di un monitor

Ritorniamo ora al problema dell'ordine di ripresa dei processi all'interno di un monitor. Se più processi sono sospesi alla condizione `x`, e se qualche processo esegue l'operazione `x.signal()`, è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell'usare un ordinamento fcfs (first-come, first-served), secondo cui il processo che attende da più tempo viene ripreso per primo. Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di attesa condizionale della forma

```
x.wait(c);
```

dove con `c` si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione `wait()`. Il valore di `c`, chiamato numero di priorità, viene poi memorizzato col nome del processo sospeso. Quando si esegue `x.signal()`, si riprende il processo cui è associato il numero di priorità più basso.

Per comprendere questo nuovo meccanismo, si consideri il monitor `assegnazione_risorse` illustrato nella Figura 6.14; tale monitor ha il compito di assegnare una particolare risorsa a processi in competizione. Quando richiede l'assegnazione di una delle sue risorse, ogni processo specifica il tempo massimo per il quale prevede di usare la risorsa. Il monitor assegna la risorsa al processo con la richiesta di assegnazione più breve.

```

monitor assegnazione_risorse
{
    boolean occupato;
    condition x;

    acquisizione(int tempo) {
        if (occupato)
            x.wait(tempo);
        occupato = true;
    }

    rilascio() {
        occupato = false;
        x.signal();
    }

    codice di inizializzazione() {
        occupato = false;
    }
}

```

Figura 6.14 Un monitor per l'assegnazione di una singola risorsa.

Per accedere alla risorsa in questione il processo deve rispettare la sequenza:

```

R.acquire(t);

...
accesso alla risorsa;
...
R.release();

```

dove R è un'istanza di tipo `assegnazione_risorse`.

Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza d'accesso sia rispettata. In particolare, può accadere quanto segue:

- un processo può accedere alla risorsa senza prima ottenere il permesso d'accesso;
- una volta che ne ha ottenuto l'accesso, un processo può non rilasciare più la risorsa;
- un processo può tentare di rilasciare una risorsa che non ha mai richiesto;

- un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta.

Le stesse difficoltà si sono incontrate nell'uso dei semafori e sono, in realtà, simili alle difficoltà che condussero allo sviluppo del costrutto monitor. In precedenza ci si è preoccupati del corretto uso dei semafori, ora ci si deve preoccupare del corretto uso delle operazioni ad alto livello definite dal programmatore, senza poter avere, a questo livello, l'assistenza del compilatore.

Una possibile soluzione del problema precedente prevede l'inclusione delle operazioni d'accesso alle risorse all'interno del monitor `assegnazione_risorse`. Tuttavia, adottando questa soluzione, per lo scheduling delle risorse si userebbe l'algoritmo base di scheduling del monitor anziché quello che abbiamo codificato.

Per garantire che i processi rispettino le sequenze appropriate è necessario controllare tutti i programmi che usano il monitor `assegnazione_risorse` e la risorsa da esso gestita. Per stabilire la correttezza del sistema è necessario verificare le seguenti due condizioni: la prima, che i processi utenti devono sempre impiegare il monitor secondo una sequenza corretta; la seconda, che è necessario assicurare che un processo non cooperativo non cerchi di aggirare la mutua esclusione offerta dal monitor, e tenti di accedere direttamente alla risorsa condivisa senza usare i protocolli d'accesso. Soltanto se si assicurano queste due condizioni, si può garantire l'assenza di errori di sincronizzazione e che l'algoritmo di scheduling sia rispettato.

Questo controllo è possibile per sistemi statici di piccole dimensioni, mentre non è ragionevolmente applicabile a sistemi di grandi dimensioni o a sistemi dinamici. Questo problema di controllo dell'accesso si può risolvere solo introducendo ulteriori meccanismi, descritti nel Capitolo 17.

6.8 Liveness

Una possibile conseguenza dell'utilizzo degli strumenti di sincronizzazione per coordinare l'accesso alle sezioni critiche è che un processo attenda indefinitamente nel tentativo di entrare nella sua sezione critica. Ricordiamo che nel Paragrafo 6.2 abbiamo delineato tre criteri che devono essere soddisfatti dalle soluzioni al problema della sezione critica. L'attesa indefinita viola due di questi criteri: il progresso e l'attesa limitata.

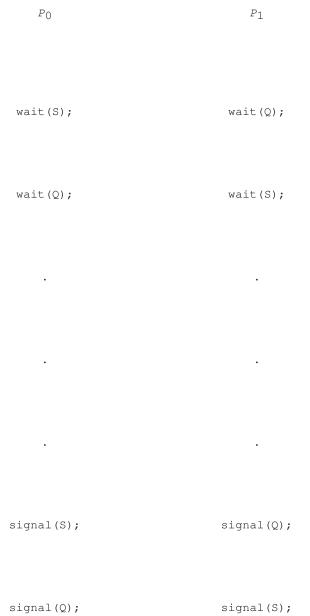
Il termine liveness fa riferimento a un insieme di proprietà che un sistema deve soddisfare per garantire che i processi facciano progressi durante il ciclo di vita della loro esecuzione. Un processo che attende indefinitamente nelle circostanze appena descritte è un esempio di "mancanza di liveness" (*liveness failure*).

Esistono molte forme diverse di mancanza di liveness, ma tutte sono generalmente caratterizzate da scarse prestazioni e scarsa reattività. Un esempio molto semplice è un ciclo infinito. Un ciclo di attesa attiva porta a una possibile mancanza di liveness, specialmente se un processo può ciclare per un tempo arbitrariamente lungo. Fornendo la mutua esclusione mediante strumenti come lock mutex e semafori è possibile indurre problemi di questo tipo nella programmazione concorrente. In questo paragrafo analizziamo due situazioni che possono portare a mancanza di liveness.

6.8.1 Stallo

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui due o più processi attendono indefinitamente un evento – l'esecuzione di un'operazione `signal()` – che può essere generato solo da uno dei processi in attesa. Quando si verifica una situazione di questo tipo si dice che i processi sono in stallo (*deadlocked*).

Per illustrare questo fenomeno si considerino due processi, P_0 e P_1 , ciascuno dei quali ha accesso a due semafori, `s` e `q`, impostati al valore 1:



Si supponga che P_0 esegua `wait(s)` e quindi P_1 esegua `wait(q)`; eseguita `wait(q)`, P_0 deve attendere che P_1 esegua `signal(q)`; analogamente, quando P_1 esegue `wait(s)`, deve attendere che P_0 esegua `signal(s)`. Poiché queste operazioni `signal()` non si possono eseguire, P_0 e P_1 sono in stallo.

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli *eventi di acquisizione e rilascio di risorse*, tuttavia anche altri tipi di eventi possono produrre situazioni di stallo (si veda il Capitolo 8, che descrive anche i meccanismi che servono ad affrontare questo tipo di problema).

Un'altra questione connessa alle situazioni di stallo è quella dell'attesa indefinita (nota anche col termine starvation). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio lifo (last-in, first-out).

INVERSIONE DI PRIORITÀ E IL MARS PATHFINDER

L'inversione di priorità può essere più di un semplice inconveniente nello scheduling. Su sistemi con vincoli temporali molto restrittivi (come i sistemi real-time) l'inversione di priorità può far sì che un processo non venga eseguito nei tempi richiesti. Quando ciò succede possono innescarsi errori a cascata in grado di provocare un malfunzionamento del sistema.

Si consideri il caso del Mars Pathfinder, una sonda spaziale della nasa che nel 1997 portò un robot, il Sojourner rover, su Marte per condurre un esperimento. Poco dopo che il Sojourner ebbe iniziato il suo lavoro, cominciarono ad aver luogo numerosi reset del sistema. Ciascun reset inizializzava di nuovo sia hardware che software, inclusi gli strumenti preposti alla comunicazione. Se il problema non fosse stato risolto, il Sojourner avrebbe fallito la sua missione.

Il problema era causato dal fatto che un processo ad alta priorità, di nome "bc_dist", impiegava più tempo del dovuto a portare a termine il suo compito. Questo processo era forzatamente in attesa di una risorsa condivisa utilizzata da un processo a priorità inferiore, denominato "asi/met", a sua volta prelazionato da diversi processi di priorità media. Il processo "bc_dist" andava quindi in stallo in attesa della risorsa condivisa, e il processo "bc_sched", rilevando il problema, eseguiva il reset. Il Sojourner soffriva dunque di un tipico caso di inversione di priorità.

Il sistema operativo installato sul Sojourner era il sistema operativo real-time VxWorks, che disponeva di una variabile globale per abilitare l'ereditarietà delle priorità su tutti i semafori. Dopo alcuni test, il valore della variabile del Sojourner (su Marte!) fu impostato correttamente, e il problema fu risolto.

Un resoconto completo del problema, della sua scoperta, e della sua soluzione è stato scritto dal responsabile del team software ed è disponibile all'indirizzo

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html.

6.8.2 Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ognualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa viene prelazionato da un processo a priorità più alta.

Assumiamo, per esempio, che vi siano tre processi L , M e H , le cui priorità seguono l'ordine $L < M < H$. Assumiamo che il processo H richiede il semaforo S al quale sta accedendo il processo L . Usualmente il processo H resterebbe in attesa che L liberi la risorsa S . Supponiamo però che M diventi eseguibile, con prelazione sul processo L . Avviene quindi, indirettamente, che un processo con priorità più bassa, il processo M , influenzi il tempo che H attenderà in attesa della risorsa S .

Questo problema di liveness è noto come inversione della priorità. Dato che l'inversione di priorità si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un protocollo di ereditarietà delle priorità, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale. Nell'esempio discusso in precedenza, un protocollo di ereditarietà delle priorità avrebbe permesso al processo L di ereditare temporaneamente la priorità di H , impedendo così al processo M di prelazionare la sua esecuzione. In un tale caso, una volta che il processo H avrà terminato con la risorsa S , rinuncerà alla priorità ereditata da H assumendo di nuovo la priorità originale. Poiché S sarà a questo punto disponibile, il processo H , e non il processo M , sarà il successivo processo eseguito.

6.9 Valutazione delle soluzioni

Abbiamo descritto diversi strumenti di sincronizzazione utilizzabili per risolvere il problema della sezione critica. Con una corretta implementazione e un utilizzo appropriato, questi strumenti possono essere efficacemente impiegati per garantire la mutua esclusione e affrontare problemi di liveness. Con la diffusione di programmi concorrenti in grado di sfruttare la potenza dei moderni sistemi multicore viene prestata sempre maggior attenzione alle prestazioni degli strumenti di sincronizzazione. Cercare di identificare quale strumento utilizzare e quando, tuttavia, può essere una sfida improba. In questo paragrafo presentiamo alcune semplici strategie per determinare quando utilizzare specifici strumenti di sincronizzazione.

Le soluzioni hardware descritte nel Paragrafo 6.4 sono considerate di livello molto basso e vengono tipicamente utilizzate come base per la costruzione di altri strumenti di sincronizzazione, come i lock mutex. Tuttavia, ci si è concentrati di recente sull'uso dell'istruzione cas per costruire algoritmi privi di lock che forniscono protezione dalle race condition senza l'overhead dei lock. Sebbene queste soluzioni lock-free stiano guadagnando popolarità grazie al basso overhead e alla loro scalabilità, gli algoritmi sono spesso difficili da sviluppare e testare.

Gli approcci basati su cas sono considerati ottimistici: prima si aggiorna fiduciosamente una variabile e poi si utilizza il rilevamento delle collisioni per verificare se un altro thread stia aggiornando la stessa variabile contemporaneamente. In tal caso, si riprova ripetutamente l'operazione fino a quando la variabile non viene aggiornata correttamente e senza conflitti. Il lock mutex, al contrario, è considerato una strategia pessimistica: si presuppone che un altro thread stia aggiornando contemporaneamente la variabile e in modo pessimistico si acquisisce il lock prima di apportare qualsiasi aggiornamento.

Le seguenti linee guida identificano delle regole generali per distinguere le prestazioni della sincronizzazione basata su cas e della sincronizzazione tradizionale (come i lock mutex e i semafori) al variare del livello di contesa.

- Nessuna contesa. Sebbene entrambe le opzioni siano generalmente veloci, la protezione cas sarà leggermente più veloce della sincronizzazione tradizionale.
- Contesa moderata. La protezione cas sarà più veloce e in alcuni casi molto più veloce rispetto alla sincronizzazione tradizionale.
- Alta contesa. Con carichi molto elevati, la sincronizzazione tradizionale sarà in definitiva più veloce della sincronizzazione basata su cas.

La contesa moderata è particolarmente interessante da esaminare. In questo scenario, l'operazione cas ha esito positivo per la maggior parte del tempo e, in caso di errore, itererà nel ciclo mostrato nella Figura 6.8 solo poche volte prima di avere successo. Per contro, con i lock mutex *qualsiasi* tentativo di acquisire un lock conteso comporterà l'esecuzione di un codice più complicato e oneroso che sospende un thread e lo colloca in una coda di wait, richiedendo un cambio di contesto verso un altro thread.

La scelta di un meccanismo per affrontare le race condition può anche influire notevolmente sulle prestazioni del sistema. Per esempio, gli interi atomici sono molto più leggeri dei lock tradizionali e sono generalmente più appropriati dei lock mutex o dei semafori per singoli aggiornamenti di variabili condivise come i contatori. Ciò è visibile anche nella progettazione dei sistemi operativi, in cui gli spinlock vengono utilizzati sui sistemi multiprocessore quando i lock vengono mantenuti per brevi periodi. In generale, i lock mutex sono più semplici e comportano meno overhead rispetto ai semafori; sono preferibili ai semafori binari per proteggere l'accesso a una sezione critica. Tuttavia, in alcuni casi, come nel controllo dell'accesso a un numero limitato di risorse, un semaforo contatore è generalmente più appropriato di un lock mutex. Analogamente, in alcuni casi, un lock lettore-scrittore può essere preferito a un lock mutex, in quanto consente un maggior grado di concorrenza (cioè più lettori).

L'attrattiva di strumenti di livello superiore come monitor e variabili condizionali si basa sulla loro semplicità e facilità d'uso. Tuttavia, tali strumenti potrebbero avere un overhead significativo e, a seconda della loro implementazione, potrebbero essere meno scalabili a situazioni con elevate contese.

Fortunatamente sono in corso molte ricerche per sviluppare strumenti scalabili ed efficienti che rispondano alle esigenze della programmazione concorrente. Alcuni esempi comprendono:

- progettazione di compilatori che generano codice più efficiente;
- sviluppo di linguaggi che forniscono supporto per la programmazione concorrente;
- miglioramento delle prestazioni delle librerie e delle api esistenti.

Nel prossimo capitolo esamineremo in che modo i vari sistemi operativi e le api disponibili agli sviluppatori implementano gli strumenti di sincronizzazione presentati in questo capitolo.

6.10 Sommario

- Una race condition si verifica quando i processi hanno accesso concorrente ai dati condivisi e il risultato finale dipende dal particolare ordine in cui si verificano gli accessi. Le race condition possono portare a valori corrotti dei dati condivisi.
- Una sezione critica è una porzione di codice in cui i dati condivisi possono essere manipolati e dove può verificarsi una race condition. Il problema della sezione critica consiste nel progettare un protocollo in base al quale i processi possano sincronizzare la loro attività per condividere in modo cooperativo i dati.
- Una soluzione al problema della sezione critica deve soddisfare i seguenti tre requisiti: (1) mutua esclusione, (2) progresso e (3) attesa limitata. La mutua esclusione garantisce che solo un processo alla volta sia attivo nella sua sezione critica. Il progresso assicura che i programmi stabiliscano in modo cooperativo quale processo entrerà nella sua sezione critica. L'attesa limitata pone un limite al tempo che un programma attenderà prima di poter entrare nella sua sezione critica.
- Le soluzioni software al problema della sezione critica, come la soluzione di Peterson, non funzionano bene con le moderne architetture elaborative.
- Il supporto hardware per il problema della sezione critica include le barriere di memoria, le istruzioni hardware come compare-and-swap e le variabili atomiche.
- Un lock mutex fornisce la mutua esclusione richiedendo che un processo acquisisca un lock prima di entrare in una sezione critica e lo rilasci all'uscita dalla sezione critica.
- I semafori, come i lock mutex, possono essere usati per fornire la mutua esclusione. Tuttavia, mentre un lock mutex ha un valore binario che indica se il blocco è disponibile o meno, un semaforo ha un valore intero e può quindi essere utilizzato per risolvere diversi problemi di sincronizzazione.
- Un monitor è un tipo di dati astratto che fornisce una forma di alto livello di sincronizzazione dei processi. Un monitor utilizza variabili condizionali per consentire ai processi di attendere che si verifichino determinate condizioni e di segnalarsi reciprocamente quando ciò avviene.
- Le soluzioni al problema della sezione critica possono soffrire di problemi di liveness, tra cui il deadlock.
- I vari strumenti che possono essere utilizzati per risolvere il problema della sezione critica e per sincronizzare l'attività dei processi possono essere valutati a seconda del livello di contesa. Alcuni strumenti funzionano meglio con un certo livello di contesa rispetto ad altri.

Esercizi

6.7 Lo pseudocodice della Figura 6.15 illustra le operazioni di base `push()` e `pop()` su uno stack basato su array. Supponendo che questo algoritmo possa essere utilizzato in un ambiente concorrente, rispondete alle seguenti domande:

```
push(item) {  
    if (top < SIZE) {  
        stack[top] = item;  
        top++;  
    }  
    else  
        ERROR  
    }  
  
pop() {  
    if (!is_empty()) {  
        top--;  
        return stack[top];  
    }  
    else  
        ERROR  
    }  
  
is_empty() {  
    if (top == 0)  
        return true;  
    else  
        return false;  
}
```

Figura 6.15 Stack basato su array per l'Esercizio 6.7.

- Quali dati presentano una corsa critica?
- Come potrebbe essere corretta questa condizione?

6.8 Le corse critiche sono possibili in molti sistemi informatici. Considerate un sistema di aste on-line in cui l'offerta corrente più alta per ciascun articolo debba essere memorizzata stabilmente. Una persona che desidera fare un'offerta su un oggetto chiama la funzione

`bid(amount)`, che mette a confronto l'importo offerto con la corrente offerta più alta. Se l'importo supera l'offerta più alta corrente, il valore dell'offerta più alta è impostato sul nuovo importo. Questa situazione è illustrata nel seguito:

```
void bid(double amount) {  
    if (amount > highestBid)  
        highestBid = amount;  
}
```

Descrivete com'è possibile una corsa critica in questa situazione e che cosa potrebbe essere fatto per evitare che tale condizione si verifichi.

6.9 Il seguente esempio di programma può essere utilizzato per sommare i valori dell'array di elementi di dimensione N in parallelo su un sistema contenente N core: (c'è un processore separato per ogni elemento dell'array):

```
for j = 1 to log_2(N) {  
    for k = 1 to N {  
        if ((k + 1) * pow(2, j) == 0) {  
            values[k] += values[k - pow(2, (j-1))]  
        }  
    }  
}
```

Questo ha l'effetto di sommare gli elementi dell'array come una serie di somme parziali, come mostrato nella Figura 6.16. Dopo che il codice è stato eseguito, la somma di tutti gli elementi nell'array è memorizzata nell'ultima posizione. Ci sono delle corse critiche nell'esempio di codice precedentemente riportato? In tal caso, identificate dove si presentano e illustratele con un esempio. In caso contrario, dimostrate perché questo algoritmo è libero dalla corsa critica.

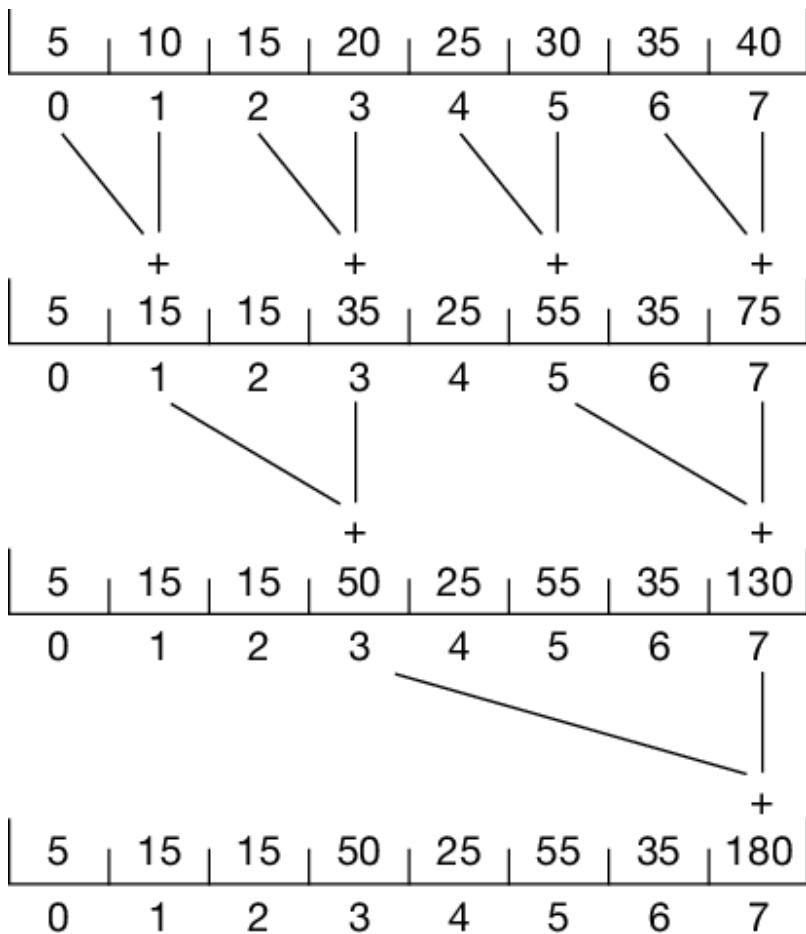


Figura 6.16 Somma di un array come serie di somme parziali per l'Esercizio 6.9.

6.10 L'istruzione `compare_and_swap()` può essere utilizzata per progettare strutture dati senza lock, come stack, code e liste. L'esempio di programma mostrato nella Figura 6.17 presenta una possibile soluzione per uno stack senza lock che utilizza istruzioni cas, in cui lo stack è rappresentato come un elenco collegato di elementi `Node` con la parte iniziale che rappresenta la parte superiore dello stack. Questa implementazione è priva di corsa critica?

```

typedef struct node {
    value_t data;
    struct node *next;
} Node;
Node *top; // cima dello stack

void push(value_t item) {
    Node *old_node;

```

```

Node *new_node;

new_node = malloc(sizeof(Node));

new_node->data = item;

do {

    old_node = top;

    new_node->next = old_node;

}

while (compare_and_swap(top,old_node,new_node) != old_node);

}

Value_t pop() {

Node *old_node;

Node *new_node;

do {

    old_node = top;

    if (old_node == NULL)

        return NULL;

    new_node = old_node->next;

}

while (compare_and_swap(top,old_node,new_node) != old_node);

return old_node->data;

}

```

Figura 6.17 Stack senza lock per l'Esercizio 6.10.

6.11 Un approccio all'utilizzo di `compare_and_swap()` per l'implementazione di uno spinlock è il seguente:

```

void lock_spinlock(int *lock) {

    while (compare_and_swap(lock, 0, 1) != 0)

        ; /* spin */

}

```

Un possibile approccio alternativo è quello di utilizzare la strategia “compare and compare-and-swap”, che controlla lo stato del blocco prima di richiamare l'operazione `compare_and_swap()`. (La logica dietro questo approccio è quella di invocare `compare_and_swap()` solo se il blocco è attualmente disponibile.)

Tale strategia è mostrata di seguito:

```

void lock_spinlock(int *lock) {
{
    while (true) {
        if (*lock == 0) {
            /* sembra che il lock sia disponibile */

            if (!compare_and_swap(lock, 0, 1))
                break;
        }
    }
}

```

Questo modello “compare and compare-and-swap” funziona in modo appropriato per l’implementazione di uno spinlock? Se è così, spiegate perché. In caso contrario, illustrate come l’integrità del lock venga compromessa.

6.12 Alcune implementazioni di semaforo forniscono una funzione `getValue()`, che restituisce il valore corrente del semaforo. Questa funzione può, per esempio, essere invocata prima di chiamare `wait()` in modo che un processo invochi la `wait()` solo se il valore del semaforo sia > 0, impedendo così il blocco in attesa del semaforo. Per esempio:

```

if (getValue(&sem) > 0)
    wait(&sem);

```

Molti sviluppatori criticano tale funzione e ne scoraggiano l’uso. Descrivete un potenziale problema che potrebbe verificarsi quando si utilizza la funzione `getValue()` in tale scenario.

6.13 L’algoritmo seguente, concepito da Dekker, è la prima soluzione software nota del problema della sezione critica per due processi. I due processi, P_0 e P_1 , condividono le seguenti variabili:

```

boolean flag[2]; /* inizialmente falsa* /

int turn;

```

La struttura del processo P_i ($i == 0$ oppure 1), dove P_j ($j == 1$ oppure 0) è l’altro processo, è mostrata nella Figura 6.18. Dimostrate che l’algoritmo soddisfa tutti e tre i requisiti per il problema della sezione critica.

```

while (true) {

    flag[i] = true;

    while (flag[j]) {

        if (turn == j) {

```

```

flag[i] = false;

while (turn == j)

; /* non fa niente */

flag[i] = true;

}

}

/* sezione critica */

turn = j;

flag[i] = false;

/* sezione non critica */

}

```

Figura 6.18 Struttura del processo P_i nell'algoritmo di Dekker.

6.14 La prima soluzione software nota del problema della sezione critica per n processi con un limite di $n - 1$ turni d'attesa è stata proposta da Eisenberg e McGuire. I processi condividono le seguenti variabili:

```

enum pstate {idle, want_in, in_cs};

pstate flag[n];

int turn;

```

Ciascun elemento di `flag` è inizialmente `idle` (inattivo); il valore iniziale di `turn` è irrilevante (compreso tra 0 e $n - 1$). La struttura del processo P_i è illustrata nella Figura 6.19. Dimostrate che tale algoritmo soddisfi tutti e tre i requisiti del problema della sezione critica.

```

while (true) {

while (true) {

flag[i] = want_in;

j = turn;

while (j != i) {

if (flag[j] != idle) {

j = turn;

```

```

    else

        j = (j + 1) % n;

    }

    flag[i] = in_cs;

    j = 0;

    while ( (j < n) && (j == i || flag[j] != in_cs))

        j++;

    if ( (j >= n) && (turn == i || flag[turn] == idle))

        break;

}

/* sezione critica */

j = (turn + 1) % n;

while (flag[j] == idle)

    j = (j + 1) % n;

turn = j;

flag[i] = idle;

/* sezione non critica */

}

```

Figura 6.19 Struttura del processo P_i nell'algoritmo di Eisenberg e McGuire.

6.15 Chiarite perché, per implementare le primitive di sincronizzazione, non sia corretto disabilitare le interruzioni di un sistema monoprocessoresso se le primitive stesse sono destinate a programmi utenti.

6.16 Si consideri l'implementazione di un lock mutex con l'utilizzo di un'istruzione `compare_and_swap()`. Supponete che sia disponibile la seguente struttura per definire il lock mutex:

```

typedef struct {

    int available;

} lock;

```

`(available == 0)` indica che il lock è disponibile e un valore pari a 1 indica che il lock non è disponibile. Utilizzando questa struttura, illustrate come le seguenti funzioni possono essere implementate con l'utilizzo di `compare_and_swap()`:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Assicuratevi di includere tutte le inizializzazioni necessarie.

6.17 Spiegate perché le interruzioni non costituiscono un metodo appropriato per implementare le primitive di sincronizzazione nei sistemi multiprocessoresso.

6.18 L'implementazione dei lock mutex esposta nel Paragrafo 6.5 soffre di attesa attiva. Descrivete quali cambiamenti sono necessari per fare in modo che un processo in attesa di acquisire un lock mutex venga bloccato e messo in una coda di attesa fino a quando il lock diventa disponibile.

6.19 Si supponga che un sistema disponga di più core di elaborazione. Per ciascuno dei seguenti scenari, individuate il miglior meccanismo di lock, tra uno spinlock e un lock mutex in cui i processi in attesa restano sospesi fino a quando il lock diventa disponibile:

- Il lock viene trattenuto per tempi brevi.
- Il lock viene trattenuto per tempi lunghi.
- Un thread può essere sospeso mentre è in possesso del lock.

6.20 Si supponga che un cambio di contesto richieda un tempo T . Stabilite un limite superiore (in termini di T) per il possesso di uno spinlock, tale per cui se lo spinlock viene mantenuto per un tempo maggiore, un lock mutex (in cui i thread in attesa vengono sospesi) diventa un'alternativa migliore.

6.21 Un server web multithread desidera tenere traccia del numero di richieste servite (questo numero è noto come hits). Considerate le due seguenti strategie per prevenire una race condition sulla variabile `hits`. La prima strategia è quella di utilizzare un lock mutex per l'aggiornamento di `hits`:

```
int hits;

mutex_lock hit_lock;

hit_lock.acquire();

hits++;

hit_lock.release();
```

Una seconda strategia è quella di utilizzare un numero intero atomico:

```
atomic_t hits;

atomic_inc(&hits);
```

Spiegate quale di queste due strategie sia più efficiente.

6.22 Si consideri l'esempio di codice per l'assegnazione e il rilascio di processi mostrato nella Figura 6.20.

```
#define MAX_PROCESSES 255

int number_of_processes = 0;

/* l'implementazione della fork() chiama questa funzione */

int allocate_process() {

    int new_pid;

    if (number_of_processes == MAX_PROCESSES)

        return -1;

    else {
```

```

    /* alloca le risorse di processo necessarie */

    ++number_of_processes;

    return new_pid;
}

}

/* l'implementazione della exit() chiama questa funzione */

void release_process() {

    /* libera le risorse di processo */

    --number_of_processes;
}

```

Figura 6.20 Allocazione e rilascio di processi per l'Esercizio 6.22.

a. Individuate la/le race condition.

b. Supponete di avere un lock mutex chiamato `mutex` con le operazioni di `acquire()` e `release()`. Indicate il punto in cui inserire il meccanismo di lock per evitare la/le race condition.

c. Possiamo sostituire la variabile intera

```

int number_of_processes = 0

con il numero intero atomico

atomic_t number_of_processes = 0

per prevenire la/le race condition?

```

6.23 I server possono essere progettati in modo da limitare il numero di connessioni aperte. In un dato momento, per esempio, un server può ammettere solo N connessioni socket per volta; dopo questo limite, il server non accetterà alcuna connessione entrante prima che una connessione esistente sia chiusa. Spiegate come il server possa usare i semafori per limitare il numero delle connessioni concorrenti.

6.24 Nel Paragrafo 6.7 un impiego scorretto dei semafori per risolvere il problema della sezione critica viene mostrato nel modo seguente:

```

wait(mutex);

...
sezione critica
...
wait(mutex);

```

Dimostrate che si tratta di un esempio di mancanza di liveness (*liveness failure*).

6.25 Dimostrate che monitor e semafori sono equivalenti in quanto consentono di risolvere gli stessi problemi di sincronizzazione.

6.26 Come si differenzia l'operazione `signal()` dei monitor dalla corrispondente operazione dei semafori?

6.27 Ipotizziamo che l'istruzione `signal()` possa apparire solo per ultima in una procedura monitor. Proponete un modo per semplificare, in questa situazione, l'implementazione descritta nel Paragrafo 6.7.

6.28 Considerate un sistema formato dai processi P_1, P_2, \dots, P_n , aventi priorità distinte, rappresentate da numeri interi. Scrivete un monitor grazie al quale tre identiche stampanti siano assegnate a tali processi, stabilendo l'ordine di allocazione in base alle priorità.

6.29 Un file deve essere condiviso fra processi diversi, ognuno dei quali è identificato da un numero univoco. Al file possono accedere simultaneamente vari processi, a patto che osservino la seguente prescrizione: la somma degli identificatori di tutti i processi che accedono al file deve essere minore di n . Scrivete un monitor per coordinare l'accesso al file.

6.30 Se un processo invoca `signal()` al verificarsi di una condizione all'interno di un monitor, esso può continuare la propria esecuzione, oppure cedere il controllo al processo che ha ricevuto il segnale. Come cambia la soluzione al precedente esercizio in queste due circostanze?

6.31 Scrivete un monitor per realizzare una "sveglia" con cui un programma chiamante possa differire la propria esecuzione per il numero prescelto di unità di tempo ("battiti" o "tic"). Si può ipotizzare l'esistenza di un orologio hardware che invochi una procedura `battito()` sul vostro monitor a intervalli regolari.

6.32 Discutete dei modi in cui il problema dell'inversione delle priorità possa essere affrontato in un sistema real-time. Discutete anche di come le soluzioni potrebbero venire implementate all'interno di uno scheduling a quote proporzionali.

6.33 Supponiamo che sia necessario gestire un numero finito di risorse dello stesso tipo. I processi possono richiedere un certo numero di queste risorse e le restituiranno una volta finito di utilizzarle. Per esempio, molti prodotti software commerciali forniscono un determinato numero di licenze, indicando il numero di applicazioni che possono essere eseguite contemporaneamente. Quando l'applicazione viene avviata, il numero delle licenze disponibili viene diminuito. Quando l'applicazione termina, il numero delle licenze viene incrementato. Se tutte le licenze sono in uso, le richieste di avvio dell'applicazione sono respinte. La richiesta sarà soddisfatta solo quando il titolare di una licenza esistente termina l'applicazione e la licenza viene restituita.

Il seguente frammento di programma viene utilizzato per gestire un numero finito di istanze di una risorsa disponibile. Il numero massimo di risorse e il numero di risorse disponibili sono dichiarati come segue:

```
#define MAX_RESOURCES 5

int available_resources = MAX_RESOURCES;
```

Quando un processo desidera ottenere un numero di risorse, invoca la funzione `decrease_count()`:

```
/* riduce available_resources del valore count */

/* restituisce 0 se sono disponibili risorse sufficienti, */

/* altrimenti restituisce -1 */

int decrease_count(int count) {

    if (available_resources < count)

        return -1;

    else {

        available_resources -= count;

        return 0;
    }
}
```

```
    }  
}  
}
```

Quando un processo vuole restituire un certo numero di risorse, chiama la funzione `increase_count()`:

```
/* aumento available_resources del valore count */  
  
int increase_count(int count) {  
  
    available_resources += count;  
  
    return 0;  
  
}
```

Il frammento di programma precedente produce una corsa critica. Fate quanto segue.

- Identificate i dati coinvolti nella corsa critica.
- Identificate la posizione (o le posizioni) nel codice in cui si verifica la corsa critica.
- Usando un semaforo o un mutex lock, correggete la condizione di corsa. È permesso modificare la funzione `decrease_count()` in modo che il processo chiamante venga bloccato fino a quando non siano disponibili risorse sufficienti.

6.34 La funzione `decrease_count()` nell'esercizio precedente al momento restituisce 0 se sono disponibili risorse sufficienti e -1 in caso contrario. Ciò porta a un codice poco elegante ed efficiente nel caso di un processo che desideri ottenere un certo numero di risorse:

```
while (decrease_count(count) == -1)  
    ;
```

Riscrivete il frammento di codice del gestore delle risorse usando un monitor e una variabile *condition* in modo che la funzione `decrease_count()` sospenda il processo fino a quando siano disponibili risorse sufficienti. Ciò consentirà a un processo di invocare `decrease_count()` semplicemente chiamando

```
decrease_count(count);
```

Il processo ritornerà da questa chiamata di funzione solo quando saranno disponibili risorse sufficienti.

CAPITOLO 7

Esempi di sincronizzazione

Nel Capitolo 6 abbiamo presentato il problema della sezione critica e ci siamo concentrati su come si possano verificare delle race condition quando più processi concorrenti condividono i dati. Abbiamo inoltre esaminato diversi strumenti che affrontano il problema della sezione critica impedendo il verificarsi di race condition. Questi strumenti spaziavano da soluzioni hardware di basso livello (come le barriere di memoria e l'operazione di compare-and-swap) a strumenti di più alto livello (dai lock mutex, ai semafori, ai monitor). Abbiamo anche discusso varie sfide nella progettazione di applicazioni che siano libere da race condition, compresi i problemi di liveness come i deadlock. In questo capitolo applichiamo gli strumenti presentati nel Capitolo 6 a diversi problemi classici di sincronizzazione. Esploriamo inoltre i meccanismi di sincronizzazione utilizzati dai sistemi operativi Linux, unix e Windows e descriviamo i dettagli delle api per Java e posix.

7.1 Classici problemi di sincronizzazione

In questo paragrafo s'illustrano diversi problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. Questi problemi sono utili per verificare quasi tutte le nuove proposte di schemi di sincronizzazione. Nelle soluzioni che proponiamo ai problemi s'impiegano i semafori, perché per tradizione le soluzioni vengono così presentate. Le implementazioni reali possono tuttavia utilizzare i lock mutex al posto dei semafori binari.

7.1.1 Produttore/consumatore con memoria limitata

Il problema del *produttore/consumatore con memoria limitata*, introdotto nel Paragrafo 6.1, si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento ad alcuna realizzazione particolare.

Nel nostro problema produttore e consumatore condividono le seguenti strutture dati:

```
int n;

semaphore mutex = 1;

semaphore empty = n;

semaphore full = 0
```

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con n posizioni, ciascuna capace di contenere un elemento. Il semaforo `mutex` garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1. I semafori `vuote` e `piene` conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo `vuote` si inizializza al valore n ; il semaforo `piene` si inizializza al valore 0.

La Figura 7.1 riporta la struttura del processo produttore, la Figura 7.2 quella del processo consumatore. È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

```
while (true) {

    ...

    /* produci un elemento in next_produced */

    ...

    wait(empty);

    wait(mutex);

    ...

    /* inserisci next_produced in buffer */

    ...

    signal(mutex);

    signal(full);

}
```

Figura 7.1 Struttura generale del processo produttore.

```

while (true) {

    wait(full);

    wait(mutex);

    . . .

    /* rimuovi un elemento da buffer e mettilo in next_consumed */

    . . .

    signal(mutex);

    signal(empty);

    . . .

    /* consuma l'elemento contenuto in next_consumed */

    . . .

}

```

Figura 7.2 Struttura generale del processo consumatore.

7.1.2 Problema dei lettori-scrittori

Si supponga che una base di dati sia da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto della base dati, mentre altri ne possono richiedere un aggiornamento, vale a dire una lettura e una scrittura. Questi due tipi di processi vengono distinti chiamando lettori quelli interessati alla sola lettura e scrittori gli altri. Naturalmente, se due lettori accedono nello stesso momento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo in fase di scrittura alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come problema dei lettori-scrittori. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità; la più semplice, cui si fa riferimento come al *primo* problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati. Il *secondo* problema dei lettori-scrittori richiede che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema e quella del secondo possono condurre a uno stato d'attesa indefinita (*starvation*), degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. Nel seguito si presenta una soluzione del primo problema dei lettori-scrittori; indicazioni attinenti a soluzioni del secondo problema immuni all'attesa indefinita si trovano nelle note bibliografiche.

La soluzione del primo problema dei lettori-scrittori prevede dunque la condivisione da parte dei processi lettori delle seguenti strutture dati:

```

semaphore rw_mutex = 1;

semaphore mutex = 1;

int read_count = 0;

```

I semafori binari `mutex` e `rw_mutex` sono inizializzati a 1; `read_count` è inizializzato a 0. Il semaforo `rw_mutex` è comune a entrambi i tipi di processi (lettori e scrittori). Il semaforo `mutex` si usa per assicurare la mutua esclusione al momento dell'aggiornamento di

`read_count`. La variabile `read_count` contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati. Il semaforo `rw_mutex` funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che entra o esce dalla sezione critica. Non serve, invece, ai lettori che entrano o escono mentre altri lettori si trovano nelle rispettive sezioni critiche.

La Figura 7.3 illustra la struttura di un processo scrittore; la Figura 7.4 presenta la struttura di un processo lettore. Occorre notare che se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a `rw_mutex` e $n - 1$ lettori a `mutex`. Inoltre, se uno scrittore esegue `signal(rw_mutex)` si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo lettore in attesa. La scelta è fatta dallo scheduler.

```
while (true) {  
  
    wait(rw_mutex);  
  
    . . .  
  
    /* esegui l'operazione di scrittura */  
  
    . . .  
  
    signal(rw_mutex);  
  
}
```

Figura 7.3 Struttura generale di un processo scrittore.

```
while (true) {  
  
    wait(mutex);  
  
    read_count++;  
  
    if (read_count == 1)  
  
        wait(rw_mutex);  
  
    signal(mutex);  
  
    . . .  
  
    /* esegui l'operazione di lettura */  
  
    . . .  
  
    wait(mutex);  
  
    read_count--;  
  
    if (read_count == 0)  
  
        signal(rw_mutex);  
  
    signal(mutex);  
  
}
```

Figura 7.4 Struttura generale di un processo lettore.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire lock di lettura-scrittura. Per acquisire un tale lock è necessario specificarne la modalità, scrittura o lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. È permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono utili soprattutto nelle situazioni seguenti.

- Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi.
- Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di lettura-scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock mutex, compensato però dalla possibilità di eseguire molti lettori in concorrenza.

7.1.3 Problema dei cinque filosofi (dining philosophers)

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette (in inglese chopstick). Si veda la Figura 7.5. Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

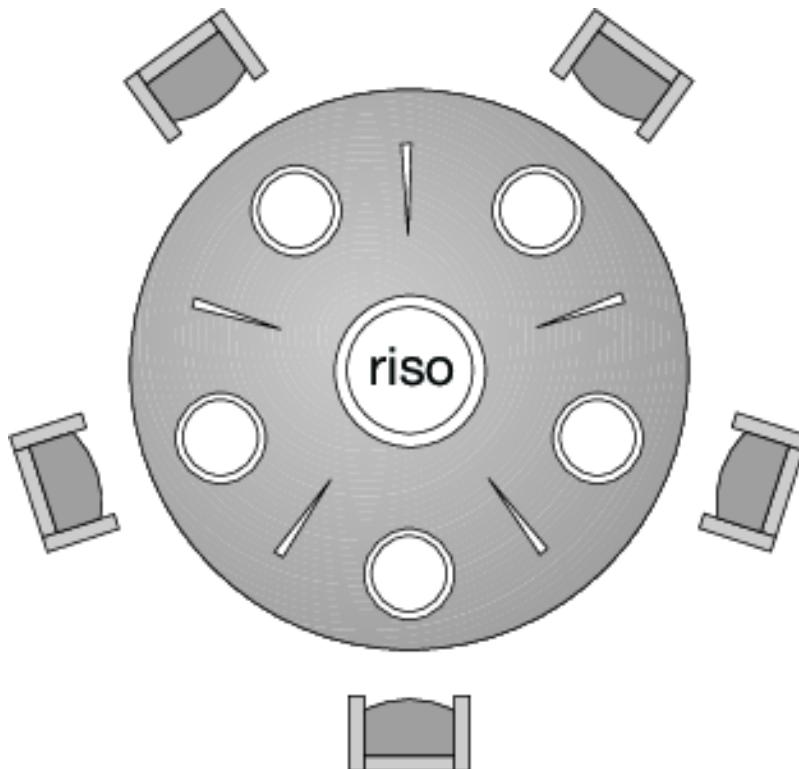


Figura 7.5 Situazione dei cinque filosofi (dining philosophers).

Il problema dei cinque filosofi (*dining philosophers*) è considerato un classico problema di sincronizzazione, non certo per la sua importanza pratica, e neanche per antipatia verso i filosofi da parte degli informatici, ma perché rappresenta una vasta classe di problemi di controllo della concorrenza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e d'attesa indefinita.

7.1.3.1 Soluzione con uso di semafori

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati. Quindi, i dati condivisi sono

```
semaphore chopstick[5];
```

dove tutti gli elementi `chopstick` sono inizializzati a 1. La struttura del filosofo i è illustrata nella Figura 7.6.

```
while (true) {  
  
    wait(chopstick[i]);  
  
    wait(chopstick[(i+1) % 5]);  
  
    . . .  
  
    /* mangia */  
  
    . . .  
  
    signal(chopstick[i]);  
  
    signal(chopstick[(i+1) % 5]);  
  
    . . .  
  
    /* pensa */  
  
    . . .  
}
```

Figura 7.6 Struttura del filosofo i .

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno afferra la bacchetta di sinistra; tutti gli elementi di `chopstick` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Tali situazioni di stallo possono essere evitate con i seguenti espedienti:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (quest'operazione si deve eseguire in una sezione critica);
- si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Nel Paragrafo 7.1.3.2 presentiamo una soluzione al problema dei cinque filosofi che assicura l'assenza di situazioni di stallo. Si noti tuttavia che qualsiasi soluzione soddisfacente per il problema dei cinque filosofi deve escludere la possibilità di situazioni d'attesa indefinita, in altre parole che uno dei filosofi muoia di fame (da qui il termine *starvation*) – una soluzione immune alle situazioni di stallo non esclude necessariamente la possibilità di situazioni d'attesa indefinita.

7.1.3.2 Soluzione per mezzo di monitor

Illustriamo i concetti relativi al costrutto monitor presentando una soluzione esente da stallo del problema dei cinque filosofi (*dining philosophers*). La soluzione impone il vincolo che un filosofo possa prendere le sue bacchette solo quando siano entrambe disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Il filosofo i può impostare la variabile `state[i] = EATING` solo se i suoi due vicini non stanno mangiando:

```
((state[(i + 4) % 5] != EATING) && (state[(i + 1) % 5] != EATING)).
```

Inoltre, occorre dichiarare la seguente struttura dati:

```
condition self[5];
```

che permette al filosofo i di ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

A questo punto si può descrivere la soluzione al problema dei cinque filosofi. La distribuzione delle bacchette è controllata dal monitor `DiningPhilosophers` (Figura 7.7). Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `pickup()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione `putdown()` e comincia a pensare. Il filosofo i deve quindi chiamare le operazioni `pickup()` e `putdown()` nella seguente sequenza:

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
```

```

if ((state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING)) {
    state[i] = EATING;
    self[i].signal();
}

}

Initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}

```

Figura 7.7 Una soluzione con monitor al problema dei cinque filosofi.

```

DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);

```

È facile dimostrare che questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. Occorre però notare che un filosofo può attendere indefinitamente. La soluzione di questo problema è lasciata come esercizio per il lettore.

7.2 Sincronizzazione all'interno del kernel

Si descrivono ora i meccanismi di sincronizzazione forniti dai sistemi operativi Windows e Linux. Abbiamo prescelto questi due sistemi perché offrono buoni esempi di approcci differenti rispetto alla sincronizzazione del kernel; come si vedrà nel corso del paragrafo i metodi di sincronizzazione messi a disposizione da questi sistemi variano in modo sottile, ma significativo.

7.2.1 Sincronizzazione in Windows

Il sistema operativo Windows ha un kernel multithread che offre anche il supporto alle applicazioni in tempo reale e alle architetture multiprocessore. Quando il kernel di Windows accede a una risorsa globale in un sistema con monoprocesso, disabilita temporaneamente le interruzioni con interrupt handler che potrebbero accedere alla stessa risorsa globale. In un sistema multiprocessore, Windows protegge l'accesso alle risorse globali con i semafori ad attesa attiva (*spinlock*), anche se il kernel usa i semafori ad attesa attiva solo per proteggere segmenti di codice brevi. Inoltre, per ragioni di efficienza, il kernel impedisce che un thread sia sottoposto a prelazione mentre detiene uno spinlock.

Per la sincronizzazione fuori dal kernel, Windows offre gli oggetti dispatcher, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi lock mutex, semafori, eventi e timer. I dati condivisi vengono protetti richiedendo che un thread entri in possesso di un mutex prima di potervi accedere, e rilasci il mutex al completamento dell'elaborazione di quei dati. Il comportamento dei semafori è analogo a quello illustrato nel Paragrafo 6.6. Gli eventi sono un meccanismo di sincronizzazione utilizzabile in modo simile alle variabili condizionali; cioè, possono notificare il verificarsi di una determinata condizione a un thread che l'attendeva. Infine i timer sono usati per informare un thread (o più di uno) della scadenza di uno specifico periodo di tempo.

Gli oggetti dispatcher possono essere nello stato *signaled* o nello stato *nonsignaled*. Uno stato signaled indica che l'oggetto è disponibile e che un thread che tentasse di accedere all'oggetto non sarebbe bloccato; uno stato nonsignaled indica che l'oggetto non è disponibile e che qualsiasi thread che tentasse di accedervi sarebbe bloccato. La Figura 7.8 illustra le transizioni di stato di un oggetto dispatcher di tipo lock mutex.

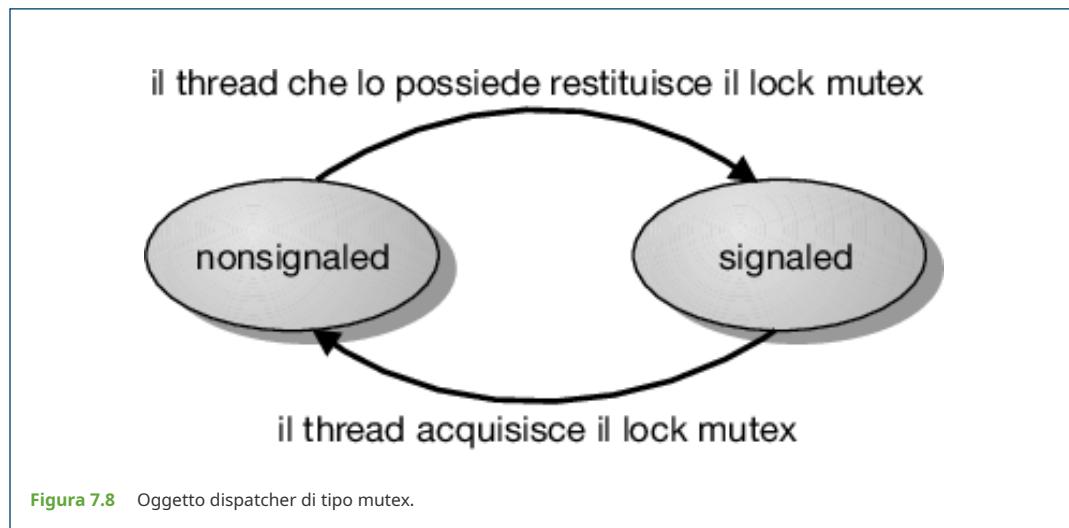


Figura 7.8 Oggetto dispatcher di tipo mutex.

C'è una relazione tra lo stato di un *oggetto dispatcher* e lo stato di un thread. Se un thread si blocca a un *oggetto dispatcher* nello stato *nonsignaled*, il suo stato cambia da pronto per l'esecuzione ad attesa, e il thread viene messo nella coda d'attesa per quell'oggetto. Quando lo stato dell'*oggetto dispatcher* diventa *signaled*, il kernel verifica se ci sono thread nella coda di attesa dell'oggetto, e in tal caso ne fa passare uno, o più d'uno, dallo stato di attesa allo stato di pronto per l'esecuzione, dal quale può riprendere l'esecuzione. Il numero dei thread che il kernel seleziona dalla coda d'attesa dipende dal tipo di *oggetto dispatcher* su cui attendono. Il kernel seleziona un solo thread dalla coda d'attesa nel caso di un mutex, poiché un oggetto mutex può essere posseduto da un solo thread. Nel caso di un oggetto evento, il kernel seleziona tutti i thread che attendono l'evento stesso.

Consideriamo un lock mutex come esempio per illustrare gli *oggetti dispatcher* e gli stati dei thread. Se un thread cerca di acquisire un *oggetto dispatcher* di tipo mutex che è nello stato *nonsignaled*, viene sospeso e messo in una coda d'attesa per l'oggetto mutex. Quando il mutex passa allo stato *signaled* (come risultato del rilascio del lock mutex da parte di un altro thread), il thread in attesa in testa alla coda del mutex passa dallo stato d'attesa allo stato di pronto per l'esecuzione e acquisisce il lock mutex.

Un oggetto sezione critica è un mutex in modalità utente che può spesso essere acquisito e rilasciato senza l'intervento del kernel. In un sistema multiprocessore, un oggetto sezione critica usa inizialmente uno spinlock in attesa che l'altro thread rilasci l'oggetto. Se l'attesa è troppo lunga, il thread alloca un mutex del kernel e cede la sua cputime. Gli oggetti sezione critica sono particolarmente

efficienti perché il mutex del kernel viene allocato solo quando c'è contesa per l'oggetto. Nella pratica vi sono ben poche contese, dunque il risparmio è notevole.

7.2.2 Sincronizzazione dei processi in Linux

Prima della versione 2.6, Linux adoperava un kernel senza prelazione; ciò significa che un processo in esecuzione in modalità kernel non poteva essere prelazionato – neppure nel caso in cui processi con priorità più alta fossero pronti per l'esecuzione. Ora, per contro, il kernel di Linux ha adottato compiutamente il procedimento della prelazione, cosicché i task attivi nel kernel possono essere sottoposti a prelazione.

Linux fornisce diversi meccanismi per la sincronizzazione nel kernel. Dato che la maggior parte delle architetture fornisce istruzioni per le versioni atomiche di semplici operazioni matematiche, la tecnica di sincronizzazione più semplice nel kernel di Linux è l'intero atomico, rappresentato mediante il tipo di dato opaco `atomic_t`. Come suggerito da questo nome, tutte le operazioni matematiche che usano numeri interi atomici vengono eseguite senza interruzioni. Si prenda in considerazione un programma composto da un intero atomico `counter` e da un intero `value`.

```
atomic_t counter;  
  
int value;
```

Il seguente codice illustra l'effetto dell'esecuzione di alcune operazioni atomiche:

Operazioni atomiche Effetto

```
atomic_set(&counter, 5); /* counter = 5 */  
  
atomic_add(10, &counter); /* counter = counter + 10 */  
  
atomic_sub(4, &counter); /* counter = counter - 4 */  
  
atomic_inc(&counter); /* counter = counter + 1 */  
  
value = atomic_read(&counter); /* value = 12 */
```

Gli interi atomici sono particolarmente efficienti in situazioni in cui deve essere aggiornata una variabile intera, per esempio un contatore, in quanto non risentono dell'overhead dei meccanismi di lock. Il loro utilizzo è tuttavia limitato a questi tipi di scenario. In situazioni in cui vi sono diverse variabili che contribuiscono a una possibile race condition, devono essere utilizzati strumenti di lock più sofisticati.

In Linux sono disponibili i lock mutex, utili per proteggere le sezioni critiche all'interno del kernel. In caso di loro utilizzo, un task deve invocare la funzione `mutex_lock()` prima di entrare in una sezione critica e la funzione `mutex_unlock()` dopo l'uscita dalla sezione critica. Se il lock mutex non è disponibile, il task che ha invocato la `mutex_lock()` viene sospeso; verrà risvegliato quando il proprietario del lock invoca `mutex_unlock()`.

Linux fornisce anche spinlock e semafori (nonché la variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel. Su macchine smp, il meccanismo fondamentale è lo spinlock: il kernel è progettato in modo da mantenere attivi gli spinlock solo per brevi periodi di tempo. Sulle macchine monoprocesso, come nel caso di sistemi embedded con un solo core di elaborazione, gli spinlock sono inadatti, e si ricorre all'abilitazione e inibizione del diritto di prelazione nel kernel. Su tali macchine, in pratica, anziché attivare uno spinlock, il kernel inibisce la prelazione; anziché rimuovere lo spinlock, abilita la prelazione. In sintesi:

monoprocessore

multiprocessore

Inibisce la prelazione a livello kernel. Attiva spinlock.

monoprocessore	multiprocessore
----------------	-----------------

Abilita la prelazione a livello kernel.	Rimuove spinlock.
---	-------------------

Nel kernel di Linux gli spinlock e i lock mutex non sono ricorsivi, il che significa che se un thread ha acquisito uno di questi lock, non può acquisirlo una seconda volta senza prima rilasciarlo. In caso contrario, il secondo tentativo di acquisizione del lock si bloccherà.

Il modo impiegato da Linux per abilitare e inibire il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`. Tuttavia non è possibile sottoporre il kernel a prelazione se un task attivo nel kernel possiede un lock. Per implementare questa regola, ogni task nel sistema possiede una struttura, `thread_info`, in cui un contatore, `preempt_count`, indica il numero dei lock detenuti dal task. Quando un lock viene acquisito, `preempt_count` aumenta di uno, mentre diminuisce di uno quando viene rilasciato. Qualora il valore di `preempt_count` per il task in esecuzione sia maggiore di zero, sarebbe rischioso sottoporre a prelazione il kernel, dato che il task possiede un lock. Se il valore è 0, il kernel può subire l'interruzione (assumendo che non vi siano chiamate in sospeso a `preempt_disable()`).

Gli spinlock, insieme all'abilitazione e inibizione della prelazione, sono utilizzati nel kernel solo quando si ricorre per breve tempo a un lock (o alla disabilitazione della prelazione del kernel). Quando vi sia necessità di mantenere un lock attivo più a lungo, è opportuno utilizzare i semafori o i lock mutex.

7.3 Sincronizzazione POSIX

I metodi di sincronizzazione discussi nel paragrafo precedente riguardano la sincronizzazione all'interno del kernel e sono quindi disponibili solo agli sviluppatori del kernel. L'api posix è invece a disposizione dei programmatori a livello utente e non fa parte di alcun particolare kernel. (Ovviamente, l'implementazione deve, in ultima analisi, utilizzare gli strumenti forniti dal sistema operativo ospite).

In questo paragrafo vengono illustrati i lock mutex, i semafori e le variabili condizionali disponibili nelle api Pthreads e posix. Queste api sono ampiamente utilizzate per la creazione e la sincronizzazione di thread da parte degli sviluppatori su sistemi unix, Linux e macos.

7.3.1 Lock mutex POSIX

I lock mutex rappresentano la tecnica di sincronizzazione fondamentale in ambiente Pthreads. La loro finalità è di proteggere le sezioni critiche del codice: un thread acquisisce un lock prima di entrare in una sezione critica; quindi, al momento di uscirne, lo rilascia. Pthreads utilizza il tipo di dato `pthread_mutex_t` per i lock mutex. Un mutex viene creato mediante la funzione `mutex_pthread_init()`. Il primo parametro è un puntatore al mutex. Passando `NULL` come secondo parametro si inizializza il mutex agli attributi predefiniti, come illustrato di seguito:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* crea e inizializza il lock mutex */

pthread_mutex_init(&mutex,NULL);
```

Il mutex viene acquisito e rilasciato con le funzioni `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente. Se il lock mutex non è disponibile quando viene invocata la `pthread_mutex_lock()`, il thread chiamante viene bloccato finché il proprietario richiama `pthread_mutex_unlock()`. Il codice seguente mostra come proteggere una sezione critica con i lock mutex.

```
/* acquisisci il lock mutex */

pthread_mutex_lock(&mutex);

/* sezione critica */

/* rilascia il lock mutex */

pthread_mutex_unlock(&mutex);
```

Tutte le funzioni mutex restituiscono 0 in caso di corretto funzionamento; se si verifica un errore restituiscono un codice di errore diverso da zero.

7.3.2 Semafori POSIX

Diversi sistemi che implementano Pthreads forniscono anche i semafori, anche se i semafori non appartengono allo standard posix, ma all'estensione posix sem. posix definisce due tipi di semafori: con nome (*named*) e senza nome (*unnamed*). Le due tipologie sono abbastanza simili, ma differiscono nel metodo di creazione e condivisione tra processi dei semafori. Poiché le tecniche utilizzate sono comuni, discutiamo entrambe le tipologie in questo paragrafo. A partire dalla versione 2.6 del kernel, i sistemi Linux forniscono supporto sia per i semafori con nome che per quelli senza nome.

7.3.2.1 Semafori POSIX con nome

La funzione `sem_open()` viene utilizzata per creare e aprire un semaforo posix con nome:

```
#include <semaphore.h>

sem_t *sem;

/* Crea il semaforo e inizializzalo a 1 */

sem = sem_open("SEM", O_CREAT, 0666, 1);
```

In questo caso stiamo dando al semaforo il nome `SEM`. Il flag `O_CREAT` indica che il semaforo, se non esiste già, verrà creato. Inoltre, il semaforo fornisce accesso in lettura e scrittura agli altri processi (tramite il parametro `0666`) ed è inizializzato a 1.

Il vantaggio dei semafori con nome è che più processi non correlati possono facilmente utilizzare un semaforo comune come meccanismo di sincronizzazione, facendo semplicemente riferimento al nome del semaforo. Nell'esempio precedente, una volta che il semaforo `SEM` è stato creato, le successive chiamate a `sem_open()` (con gli stessi parametri) da parte di altri processi restituiscono un descrittore al semaforo esistente.

Nel Paragrafo 6.6 abbiamo descritto le classiche operazioni di `wait()` e `signal()`. In posix le stesse operazioni si chiamano `sem_wait()` e `sem_post()`, rispettivamente. Il seguente esempio di codice illustra la protezione di una sezione critica utilizzando il semaforo creato in precedenza:

```
/* acquisisce il semaforo */

sem_wait(sem);

/* sezione critica */

/* rilascia il semaforo */

sem_post(sem);
```

Sia i sistemi Linux sia quelli macos forniscono semafori posix con nome.

7.3.2.2 Semafori POSIX senza nome

Un semaforo senza nome viene creato e inizializzato mediante la funzione `sem_init()`, a cui vengono passati tre parametri:

1. un puntatore al semaforo;
2. un flag che indica il livello di condivisione;
3. il valore iniziale del semaforo.

L'uso di `sem_init()` è illustrato nel seguente codice:

```
#include <semaphore.h>

sem_t sem;

/* Crea il semaforo e inizializzalo a 1 */

sem_init(&sem, 0, 1);
```

In questo esempio il flag 0 indica che il semaforo può essere condiviso solo dai thread che appartengono al processo che lo ha creato (passando un valore diverso da zero possiamo consentire che il semaforo sia condiviso tra processi distinti, posizionandolo in una

regione di memoria condivisa). Inoltre, il semaforo è inizializzato al valore 1.

I semafori posix senza nome usano le stesse operazioni di quelli con nome, `sem_wait()` e `sem_post()`. Il codice che segue mostra come proteggere una sezione critica utilizzando il semaforo creato in precedenza:

```
/* acquisisci il semaforo */

sem_wait(&sem);

/* sezione critica */

/* rilascia il semaforo */

sem_post(&sem);
```

Proprio come per i lock mutex, tutte le funzioni sui semafori restituiscono 0 in caso di successo e un valore diverso da zero quando si verifica una condizione di errore.

7.3.3 Variabili condizionali POSIX

Le variabili condizionali in Pthreads si comportano in modo simile a quelle descritte nel Paragrafo 6.7. Tuttavia in tale paragrafo le variabili condizionali sono utilizzate nel contesto di un monitor, che fornisce un meccanismo di locking per assicurare l'integrità dei dati. Visto che Pthreads è tipicamente usato nei programmi C e che il C non ha i monitor, realizziamo il lock associando un lock mutex a una variabile condizionale.

Le variabili condizionali in Pthreads usano il tipo di dato `pthread_cond_t` e vengono inizializzate mediante la funzione `pthread_cond_init()`. Il codice seguente crea e inizializza una variabile condizionale e il lock mutex a essa associato:

```
pthread_mutex_t mutex;

pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);

pthread_cond_init(&cond_var,NULL);
```

Per l'attesa su una variabile condizionale viene usata la funzione `pthread_cond_wait()`.

Il seguente codice mostra come un thread può aspettare il verificarsi della condizione `a == b` utilizzando una variabile condizionale Pthreads:

```
pthread_mutex_lock(&mutex);

while (a != b)

pthread_cond_wait (&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Il lock mutex associato alla variabile condizionale deve essere bloccato prima della chiamata `pthread_cond_wait()`, in quanto viene utilizzato per proteggere i dati nell'istruzione condizionale da una possibile race condition. Una volta acquisito il lock, il thread può verificare la condizione. Se la condizione non è verificata, il thread richiama `pthread_cond_wait()`, passando il lock mutex e la variabile condizionale come parametri. La chiamata a `pthread_cond_wait()` rilascia il lock mutex, consentendo in tal modo a un altro thread di accedere al dato condiviso ed eventualmente aggiornare il suo valore in modo che la condizione restituisca true. (Per proteggersi contro eventuali errori è importante collocare l'istruzione condizionale all'interno di un ciclo, in modo che la condizione sia ricontrollata dopo che si è ricevuto un segnale).

Un thread che modifica i dati condivisi può richiamare la funzione `pthread_cond_signal()`, in modo da mandare un segnale a un thread in attesa sulla variabile condizionale, come mostrato di seguito:

```
pthread_mutex_lock(&mutex);  
  
a = b;  
  
pthread_cond_signal(&cond_var);  
  
pthread_mutex_unlock(&mutex);
```

È importante notare che il lock mutex non viene rilasciato dalla chiamata `pthread_cond_signal()`, ma dalla successiva chiamata `pthread_mutex_unlock()`. Una volta che il lock mutex viene rilasciato, il thread che ha ricevuto il segnale diventa proprietario del lock e il controllo riprende dalla chiamata alla `pthread_cond_wait()`.

Alla fine di questo capitolo presenteremo diversi progetti e problemi di programmazione che utilizzano lock mutex, variabili condizionali Pthreads e semafori posix.

7.4 Sincronizzazione in Java

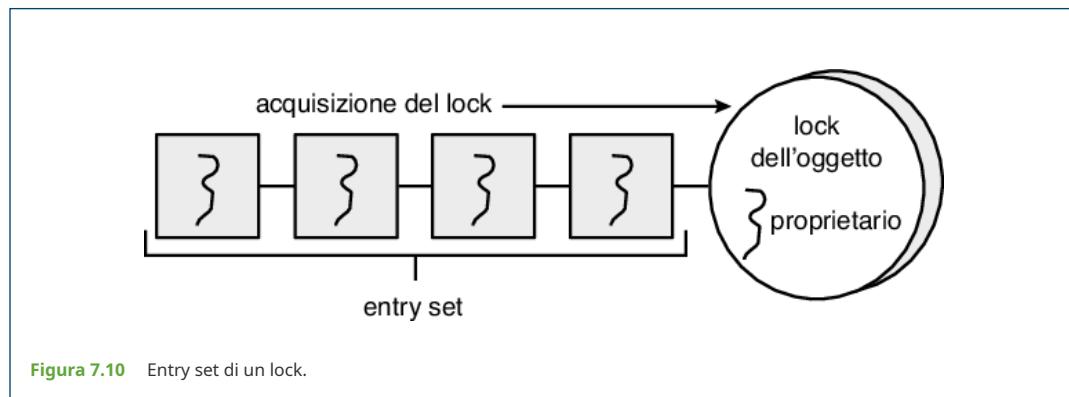
Il linguaggio Java e le sue api hanno fornito un supporto completo alla sincronizzazione dei thread sin dalle origini. In questo paragrafo tratteremo prima di tutto i monitor Java, il meccanismo di sincronizzazione originale del linguaggio. Analizzeremo poi tre meccanismi aggiuntivi introdotti dalla versione 1.5 di Java: i lock rientranti (*reentrant lock*), i semafori e le variabili condizionali. Tratteremo questi perché rappresentano i meccanismi di lock e sincronizzazione più comuni, tuttavia l'api Java fornisce molte funzionalità che non vedremo in questo testo, per esempio il supporto per le variabili atomiche e l'istruzione cas. Invitiamo i lettori interessati a consultare la bibliografia per ulteriori approfondimenti.

7.4.1 Monitor Java

Java fornisce un meccanismo di concorrenza per la sincronizzazione dei thread simile ai monitor. Illustriamo questo meccanismo con la classe `BoundedBuffer` (Figura 7.9), che implementa una soluzione al problema del buffer limitato in cui il produttore e il consumatore invocano rispettivamente i metodi `insert()` e `remove()`.

Ciascun oggetto, in Java, ha associato un singolo lock. Quando si dichiara un metodo `synchronized`, per invocare il metodo occorre possedere il lock dell'oggetto. Si dichiara `synchronized` un metodo inserendo la parola chiave nella definizione del metodo, come avviene, per esempio, con i metodi `insert()` e `remove()` nella classe `BoundedBuffer`.

Per richiamare un metodo `synchronized` è necessario possedere il lock su un'istanza dell'oggetto `BoundedBuffer`. Se il lock è già di proprietà di un altro thread, il thread che invoca il metodo `synchronized` si blocca e viene inserito in una lista d'attesa, detta entry set, per il lock dell'oggetto. L'entry set è formata dall'insieme di thread che attendono la disponibilità del lock. Se il lock è disponibile quando viene chiamato un metodo `synchronized`, il thread chiamante diventa il proprietario del lock dell'oggetto e può accedere al metodo. Il lock ritorna disponibile quando il thread termina l'esecuzione del metodo. Se al rilascio del lock l'entry set del lock non è vuoto, la jvm seleziona arbitrariamente da questo insieme di thread il nuovo proprietario del lock (quando diciamo "arbitrariamente" intendiamo che la specifica non richieda che i thread siano organizzati in un ordine particolare; tuttavia, nella pratica, la maggior parte delle macchine virtuali ordina i thread dell'entry set in base a un criterio fifo). La Figura 7.10 illustra il funzionamento dell'entry set.



A ogni oggetto, oltre a un lock, è anche associato un insieme di thread in attesa, detto wait set e inizialmente vuoto. Quando un thread entra in un metodo sincronizzato, possiede il lock per l'oggetto. Tuttavia questo thread potrebbe determinare che gli è impossibile continuare l'esecuzione, perché una data condizione non è soddisfatta. Ciò avverrà, per esempio, se il produttore chiama il metodo `insert()` e il buffer è pieno. Il thread rilascerà quindi il lock e attenderà fino a quando non sarà soddisfatta la condizione che gli consentirà di continuare.

Quando un thread chiama il metodo `wait()` accade quanto segue.

1. Il thread rilascia il lock dell'oggetto.
2. Lo stato del thread è impostato su `blocked`.
3. Il thread viene inserito nel wait set dell'oggetto.

Si consideri l'esempio nella Figura 7.11. Se il produttore chiama il metodo `insert()` e osserva che il buffer è pieno, allora invoca il metodo `wait()`. Questa chiamata rilascia il lock, blocca il produttore e inserisce il produttore nel wait set dell'oggetto. Poiché il produttore ha rilasciato il lock, il consumatore prima o poi accederà al metodo `remove()`, dove sarà liberato spazio nel buffer per il produttore. La Figura 7.12 illustra l'entry set e il wait set di un lock. Si noti che, sebbene la `wait()` possa sollevare un'`InterruptedException`, abbiamo scelto di ignorarla, per chiarezza e semplicità del codice.

```

public class BoundedBuffer<E>

{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;

    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* I produttori chiamano questo metodo */

    public synchronized void insert(E item) {

        /* Si veda la Figura 7.11 */

    }

    /* I consumatori chiamano questo metodo */

    public synchronized E remove() {

        /* Si veda la Figura 7.11 */

    }
}

```

Figura 7.9 Buffer limitato con l'uso della sincronizzazione Java.

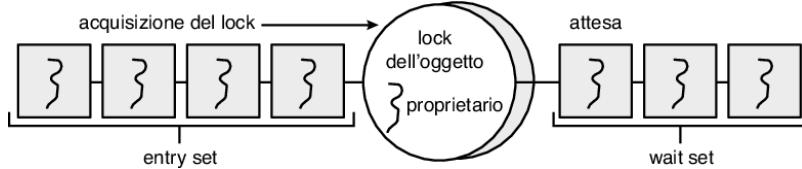


Figura 7.12 Entry set e wait set.

```
/* I produttori chiamano questo metodo */

public synchronized void insert(E item) {

    while (count == BUFFER_SIZE) {

        try {
            wait();
        }

        catch (InterruptedException ie) { }

    }

    buffer[in] = item;

    in = (in + 1) % BUFFER_SIZE;

    count++;

    notify();
}

/* I consumatori chiamano questo metodo */

public synchronized E remove() {

    E item;

    while (count == 0) {

        try {
            wait();
        }

        catch (InterruptedException ie) { }

    }

    item = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

    count--;

    notify();

    return item;
}
```

Figura 7.11 Implementazione di `insert()` e `remove()` con l'uso di `wait()` e `notify()`.

Come fa il thread consumatore a segnalare che il produttore può procedere? Solitamente, quando un thread esce da un metodo `synchronized` rilascia solo il lock associato all'oggetto, eventualmente rimuovendo un thread dall'entry set e assegnandogli la proprietà del lock. Tuttavia, alla fine dei metodi `insert()` e `remove()` è presente una chiamata al metodo `notify()`, che:

1. seleziona un thread arbitrario τ dall'elenco di thread nel wait set;
2. sposta τ dal wait set all'entry set;
3. modifica lo stato di τ da blocked a runnable (eseguibile).

τ è ora in grado di competere per il lock con gli altri thread. Una volta che τ ha riacquistato il controllo del lock, esegue il ritorno dalla chiamata `wait()` e può controllare nuovamente il valore di `count`. Anche in questo caso, la selezione di un thread arbitrario è in accordo con le specifiche Java: nella pratica, la maggior parte delle macchine virtuali Java ordina i thread nel wait set secondo una politica fifo.

Descriviamo ora il funzionamento dei metodi `wait()` e `notify()` mediante il codice mostrato nella Figura 7.11. Supponiamo che il buffer sia pieno e che il lock dell'oggetto sia disponibile.

- Il produttore chiama il metodo `insert()`, vede che il lock è disponibile e inizia l'esecuzione del metodo. Una volta nel metodo, determina che il buffer è pieno e chiama `wait()`. La chiamata a `wait()` rilascia il lock dell'oggetto, imposta lo stato del produttore su `blocked` e mette il produttore nel wait set dell'oggetto.
- A un certo punto il consumatore invoca il metodo `remove()` e vi accede, poiché il lock dell'oggetto è ora disponibile. Il consumatore rimuove un elemento dal buffer e chiama `notify()`. Si noti che il consumatore possiede ancora il lock.
- La chiamata a `notify()` rimuove il produttore dal wait set dell'oggetto, sposta il produttore nell'entry set e imposta lo stato del produttore su `runnable`.
- Il consumatore esce dal metodo `remove()`. L'uscita da questo metodo rilascia il lock per l'oggetto.
- Il produttore cerca di riacquisire, con successo, il lock. Riprende quindi l'esecuzione dalla chiamata alla `wait()`. Il produttore esegue il test del ciclo `while`, determina che vi sia spazio disponibile nel buffer e procede con il resto del metodo `insert()`. Se non ci sono thread nel wait set dell'oggetto, la chiamata a `notify()` viene ignorata. Quando il produttore esce dal metodo, rilascia il lock.

I meccanismi `synchronized`, `wait()` e `notify()` hanno fatto parte di Java fin dalle sue origini. Tuttavia, le revisioni successive dell'api Java hanno introdotto meccanismi di blocco molto più flessibili e robusti, alcuni dei quali verranno esaminati nei prossimi paragrafi.

SINCRONIZZAZIONE DI BLOCCHI

Il tempo che intercorre tra il momento in cui un lock viene acquisito e il momento in cui viene rilasciato viene chiamato **ambito (scope) del lock**. Un metodo `synchronized` in cui solo una piccola percentuale di codice è destinata alla manipolazione di dati condivisi può produrre un ambito eccessivamente esteso. In tal caso, può essere meglio sincronizzare solo il blocco di codice che manipola i dati condivisi piuttosto che l'intero metodo: ciò consente di avere un ambito di lock più piccolo. Per questa ragione, oltre a offrire la possibilità di dichiarare i metodi `synchronized`, Java permette la sincronizzazione di blocchi, come illustrato di seguito. In questo caso, solo l'accesso al codice della sezione critica richiede il possesso del lock dell'oggetto.

```
public void someMethod() {  
  
    /* sezione non critica */  
  
    synchronized(this) {  
  
        /* sezione critica */  
  
    }  
  
    /* resto del codice */  
}
```

7.4.2 Lock rientranti

Il meccanismo di locking più semplice disponibile nell'api è probabilmente il lock rientrante (`ReentrantLock`). Per molti versi, il lock rientrante si comporta come la dichiarazione `synchronized` descritta nel Paragrafo 7.4.1: un lock rientrante è di proprietà di un singolo thread e viene utilizzato per fornire l'accesso mutuamente esclusivo a una risorsa condivisa. Tuttavia, il lock rientrante offre diverse funzionalità aggiuntive, tra cui la possibilità di impostare un parametro di equità (*fairness*) che favorisce la concessione del lock al thread in attesa da più tempo (si ricordi che le specifiche della jvm non indicano che i thread nel wait set del lock di un oggetto devono essere ordinati in un modo particolare).

Un thread acquisisce un lock rientrante richiamando il suo metodo `lock()`. Se il lock è disponibile, o se il thread che invoca `lock()` lo possiede già (motivo per cui il lock viene definito *rientrante*), il metodo `lock()` assegna la proprietà del lock al chiamante e restituisce il controllo. Se il blocco non è disponibile, il thread chiamante si blocca fino a quando, a un certo punto, non gli viene assegnato il lock, in seguito alla chiamata `unlock()` da parte del suo proprietario. `ReentrantLock` implementa l'interfaccia `Lock` ed è usato come segue:

```
Lock key = new ReentrantLock();

key.lock();

try {
    /* sezione critica */

}

finally {
    key.unlock();
}
```

L'utilizzo di `try` e `finally` nel programma richiede una breve spiegazione. Se il lock viene acquisito tramite il metodo `lock()` è importante che anche il rilascio avvenga in modo simile. Includendo `unlock()` in una clausola `finally`, si assicura che il lock venga rilasciato una volta completata la sezione critica, oppure se si verifica un'eccezione all'interno del blocco `try`. Si noti che non inseriamo la chiamata a `lock()` all'interno della clausola `try`, poiché `lock()` non solleva alcuna eccezione controllata. Si pensi a ciò che succederebbe se mettessimo `lock()` all'interno della clausola `try` e si verificasse un'eccezione non controllata quando viene effettuata la chiamata (per esempio un `OutOfMemoryError`). In questo caso la clausola `finally` attiverebbe la chiamata a `unlock()`, che quindi solleverebbe l'eccezione incontrollata `IllegalMonitorStateException`, in quanto il lock non è mai stato acquisito. La `IllegalMonitorStateException` sostituisce l'eccezione non controllata che si è verificata alla chiamata di `lock()`, oscurando così il motivo per cui il programma non ha funzionato.

Anche se un `ReentrantLock` offre la mutua esclusione, può trattarsi di una strategia troppo conservativa se più thread si limitano a leggere, senza scrivere, dati condivisi (abbiamo descritto questo scenario nel Paragrafo 7.1.2). Per rispondere a questa esigenza, l'api Java fornisce anche un `ReentrantReadWriteLock`, un lock che consente più lettori concorrenti, ma un solo scrittore.

7.4.3 Semafori

L'api Java fornisce anche i semafori contatore, descritti nel Paragrafo 6.6. Il costruttore del semaforo è il seguente:

```
Semaphore(int value);
```

dove `value` specifica il valore iniziale del semaforo (è consentito anche un valore negativo). Il metodo `acquire()` genera un'`InterruptedException` se il thread che deve acquisire il semaforo viene interrotto. Il seguente esempio illustra l'uso di un semaforo per la mutua esclusione:

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* sezione critica */
}
```

```

        catch (InterruptedException ie) {}

        finally {
            sem.release();
        }
    }
}

```

Si noti che la chiamata a `release()` si trova nella clausola `finally`, in modo da assicurare che il semaforo sia rilasciato.

7.4.4 Variabili condizionali

L'ultima funzionalità dell'api Java che trattiamo è la variabile condizionale. Proprio come `ReentrantLock` è simile alla dichiarazione `synchronized` di Java, le variabili condizionali forniscono funzionalità simili ai metodi `wait()` e `notify()`. Pertanto, per garantire una mutua esclusione, una variabile condizionale deve essere associata a un lock rientrante.

È possibile creare una variabile condizionale creando prima un lock rientrante e invocando il suo metodo `newCondition()`, che restituisce un oggetto `Condition` che rappresenta la variabile condizionale per il lock rientrante associato. Questi passaggi sono illustrati nel codice che segue:

```

Lock key = new ReentrantLock();

Condition condVar = key.newCondition();

```

Una volta ottenuta la variabile condizionale, possiamo invocare i suoi metodi `await()` e `signal()`, che funzionano come i comandi `wait()` e `signal()` descritti nel Paragrafo 6.7.

Ricordiamo che, come descritto nel Paragrafo 6.7 relativo ai monitor, le operazioni `wait()` e `signal()` possono essere applicate alle variabili condizionali con nome, consentendo a un thread di attendere il verificarsi di una specifica condizione o di essere avvisato quando una data condizione viene soddisfatta. A livello di linguaggio, Java non fornisce il supporto per le variabili condizionali con nome. Ogni monitor Java è associato a una sola variabile senza nome e le operazioni `wait()` e `notify()` descritte nel Paragrafo 7.4.1 si applicano solo a questa singola variabile condizionale. Quando un thread Java viene risvegliato da una `notify()`, non riceve alcuna informazione sul motivo della notifica e spetta al thread stesso verificare se la condizione per la quale si trovava in attesa sia stata soddisfatta. Le variabili condizionali rimediano a questo problema consentendo di notificare un thread specifico.

Illustriamo quanto detto con un esempio. Supponiamo di avere cinque thread, numerati da 0 a 4, e una variabile condivisa `turn` che indichi a quale thread spetti il turno. Quando un thread desidera svolgere qualche attività chiama il metodo `doWork()` mostrato nella Figura 7.13, passandogli il suo numero di thread. Solo il thread il cui valore `threadNumber` corrisponde al valore di `turn` può procedere, mentre gli altri thread devono attendere il proprio turno.

```

/*
 * threadNumber è il thread che desidera effettuare qualche attività */
public void doWork(int threadNumber)

{
    lock.lock();

    try {
        /**
         * Se non è il mio turno,
         * attendo di ricevere un segnale.

    }

    if (threadNumber != turn)


```

```

condVars[threadNumber].await();

/**
 * Esegui qualche attività...
 */

/**
 * Invia il segnale al prossimo thread.
 */

turn = (turn + 1) % 5;

condVars[turn].signal();

}

catch (InterruptedException ie) { }

finally {

lock.unlock();

}

}

```

Figura 7.13 Utilizzo delle variabili condizionali in Java.

È anche necessario creare un lock rientrante e cinque variabili condizionali (che rappresentano le condizioni attese dai thread) per poter inviare un segnale al thread a cui spetta il turno successivo, come mostrato di seguito:

```

Lock lock = new ReentrantLock();

Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)

condVars[i] = lock.newCondition();

```

Quando un thread entra in `doWork()`, se il suo `threadNumber` è diverso da `turn`, invoca il metodo `await()` sulla sua variabile condizionale associata, in modo da poter riprendere quando riceve un segnale da un altro thread. Dopo che un thread ha completato il suo lavoro, invia un segnale alla variabile condizionale associata al thread successivo.

È importante notare che `doWork()` non ha bisogno di essere dichiarato `synchronized`, perché il lock rientrante offre la mutua esclusione. Quando un thread richiama `await()` sulla variabile condizionale rilascia il lock rientrante associato, consentendo a un altro thread di acquisire il lock di mutua esclusione. Analogamente, quando viene invocato `signal()`, viene inviato un segnale solo alla variabile condizionale. Il lock viene rilasciato invocando `unlock()`.

7.5 Approcci alternativi

Con l'emergere dei sistemi multicore si è visto un aumento della pressione per lo sviluppo di applicazioni multithread in grado di sfruttare la presenza di più core di elaborazione. Le applicazioni multithread presentano però un rischio maggiore di race condition e situazioni di stallo. Tradizionalmente, per risolvere questi problemi sono state utilizzate tecniche come i lock mutex, i semafori e i monitor, ma al crescere del numero di core diventa sempre più difficile progettare applicazioni multithread che siano esenti da race condition e stalli. In questo paragrafo esploriamo diverse funzionalità presenti sia nei linguaggi di programmazione sia a livello hardware a supporto del progetto di applicazioni concorrenti sicure.

7.5.1 Memoria transazionale

Molto spesso in informatica le idee provenienti da un'area di studio possono essere utilizzate per risolvere problemi in altre aree. Il concetto di memoria transazionale, che ha avuto origine nella teoria dei database, per esempio, fornisce una strategia per la sincronizzazione dei processi. Una transazione di memoria è una sequenza atomica di operazioni di lettura-scrittura. Se tutte le operazioni di una transazione sono eseguite, la transazione di memoria viene completata, altrimenti le operazioni devono essere annullate e deve essere ripristinata la situazione precedente l'inizio della transazione. I vantaggi della memoria transazionale possono essere sfruttati mediante l'aggiunta di nuove funzionalità a un linguaggio di programmazione.

Si consideri il seguente esempio. Si supponga di avere a disposizione una funzione `update()` che modifica dati condivisi. Questa funzione potrebbe essere scritta in maniera tradizionale usando i lock mutex (o i semafori):

```
void update ()  
{  
    acquire();  
    /* modifica dati condivisi */  
    release();  
}
```

Tuttavia, l'utilizzo di meccanismi di sincronizzazione come lock mutex e semafori implica molti potenziali problemi, incluso lo stallo dei processi. Inoltre, in seguito alla crescita del numero di thread, i meccanismi tradizionali basati sui lock non si adattano al meglio, perché il livello di contesa tra i thread per la proprietà del lock diventa molto elevato.

Un'alternativa consiste nell'aggiungere ai linguaggi di programmazione nuove funzionalità che sfruttano il vantaggio dato dalla memoria transazionale. Nel nostro esempio, supponiamo di aggiungere il costrutto `atomic{S}` che assicura che le operazioni in S siano eseguite come transazione. Potremo riscrivere il metodo `update()` così:

```
void update ()  
{  
    atomic {  
        /* modifica dati condivisi */  
    }  
}
```

Il vantaggio di utilizzare tale meccanismo al posto dei lock sta nel fatto che sia il sistema di memoria transazionale, e non il programmatore, a garantire l'atomicità. Inoltre, poiché non sono coinvolti i lock, non possono verificarsi situazioni di stallo. Inoltre, questo sistema è in grado di identificare le istruzioni nei blocchi atomici che possono essere eseguite in concorrenza, come accessi concorrenti in lettura a una variabile condivisa. È comunque certamente possibile per un programmatore identificare queste situazioni e utilizzare i lock di lettura-scrittura, ma il compito si complica al crescere del numero di thread in un'applicazione.

La memoria transazionale può essere implementata via software oppure via hardware. La memoria transazionale software (stm), come il nome suggerisce, implementa la memoria transazionale esclusivamente via software, senza la necessità di hardware particolare. In questo schema si inserisce del codice ausiliario nelle transazioni.

Esso è prodotto e inserito da un compilatore e gestisce ciascuna transazione esaminando quali istruzioni possono essere eseguite in concorrenza, e quando sono necessari dei lock a basso livello. La memoria transazionale hardware (htm) utilizza gerarchie di cache hardware e protocolli di coerenza della cache per gestire e risolvere conflitti riguardanti dati condivisi residenti in memorie cache di processori distinti. htm non richiede una particolare strumentazione software e ha quindi un minor overhead rispetto a stm. Tuttavia, richiede che le gerarchie di cache esistenti e i protocolli di coerenza della cache siano modificati per il supporto di memorie transazionali.

Le memorie transazionali esistono da diversi anni, ma per diverso tempo non hanno conosciuto una vasta diffusione. Soltanto di recente il crescente utilizzo dei sistemi multicore e la maggior enfasi sulla programmazione concorrente e parallela hanno focalizzato molta ricerca su questo settore, sia da parte delle istituzioni accademiche sia da parte dei produttori di hardware e software.

7.5.2 OpenMP

Nel Paragrafo 4.5.2 abbiamo introdotto Openmp e il suo supporto alla programmazione parallela in ambienti a memoria condivisa. Ricordiamo che Openmp comprende una serie di direttive del compilatore e una api. Il codice che segue la direttiva `#pragma omp parallel` viene identificato come regione parallela e viene eseguito da un numero di thread pari al numero di core di elaborazione del sistema. Il vantaggio di Openmp (e di strumenti analoghi) è che la creazione e la gestione dei thread è gestita dalla libreria Openmp e non è sotto la responsabilità degli sviluppatori di applicazioni.

Oltre alla direttiva del compilatore `#pragma omp parallel`, Openmp include la direttiva `#pragma omp critical`, che specifica che la regione di codice dopo la direttiva è una sezione critica in cui solo un thread alla volta può essere attivo. In questo modo Openmp fornisce il supporto per garantire che i thread non generino race condition.

Come esempio dell'uso della direttiva per la sezione critica, si assuma che la variabile condivisa `counter` possa essere modificata dalla funzione `update()` come segue:

```
void update(int value)

{
    counter += value;
}
```

Se la funzione `update()` potesse far parte di una sezione critica (o essere invocata da questa), sarebbe possibile una race condition sulla variabile `counter`.

La direttiva del compilatore per la sezione critica può essere usata per porre rimedio a questa race condition nel seguente modo:

```
void update(int value)

{
    #pragma omp critical
    {
        counter += value;
    }
}
```

La direttiva per la sezione critica si comporta come un semaforo binario o un lock mutex, assicurando che solo un thread alla volta sia attivo nella sezione critica. Se un thread tenta di entrare in una sezione critica mentre un altro thread è attualmente attivo in quella sezione (cioè, possiede la sezione), il thread chiamante viene bloccato fino all'uscita del thread proprietario. Se è necessario utilizzare più sezioni critiche, a ciascuna sezione può essere assegnato un nome distinto e una regola può specificare che non più di un thread può essere attivo simultaneamente in una sezione critica con lo stesso nome.

Un vantaggio nell'utilizzo della direttiva del compilatore per la sezione critica in Openmp è che è generalmente considerato più semplice rispetto ai lock mutex standard. Tuttavia, tocca ancora agli sviluppatori di applicazioni di individuare possibili race condition e proteggere adeguatamente i dati condivisi utilizzando la direttiva del compilatore. Inoltre, poiché la direttiva per la sezione critica si comporta in modo molto simile a un lock mutex, sono possibili situazioni di stallo quando vengono definite due o più sezioni critiche.

7.5.3 Linguaggi di programmazione funzionali

La maggior parte dei linguaggi di programmazione noti, come C, C++, Java e C#, sono linguaggi imperativi (o procedurali). I linguaggi imperativi sono utilizzati per l'implementazione di algoritmi basati sugli stati. In questi linguaggi il flusso dell'algoritmo è fondamentale per il suo corretto funzionamento e uno stato è rappresentato dalle variabili e dalle altre strutture dati. Naturalmente lo stato del programma cambia, visto che il valore delle variabili può essere modificato nel tempo.

Con l'enfasi che viene posta attualmente sulla programmazione concorrente e parallela per sistemi multicore, è aumentata l'attenzione verso i linguaggi di programmazione funzionale, che seguono un paradigma di programmazione molto diverso da quello proposto dai linguaggi imperativi. La differenza fondamentale tra i linguaggi imperativi e quelli funzionali è che i linguaggi funzionali non mantengono uno stato. In altre parole, una volta che una variabile è stata definita e inizializzata, il suo valore è immutabile, non può cambiare. Poiché i linguaggi funzionali non permettono la mutazione di stato, non si devono preoccupare di questioni come le race condition e gli stalli. In sostanza, la maggior parte dei problemi affrontati in questo capitolo non esiste nei linguaggi funzionali.

Diversi linguaggi funzionali sono attualmente in uso e noi menzioniamo molto brevemente due di questi: Erlang e Scala. Il linguaggio Erlang è stato oggetto di una significativa attenzione grazie al suo supporto alla concorrenza e alla facilità con cui può essere utilizzato per sviluppare applicazioni per sistemi paralleli. Scala è un linguaggio funzionale orientato agli oggetti. La sintassi di Scala è in gran parte simile a quella dei popolari linguaggi orientati agli oggetti Java e C#. I lettori interessati a Erlang e Scala, e in generale a ulteriori dettagli sui linguaggi funzionali, sono invitati a consultare la bibliografia alla fine del capitolo.

7.6 Sommario

- Tra i problemi classici di sincronizzazione dei processi vi sono il problema del buffer limitato, dei lettori-scrittori e dei dining philosophers. Le soluzioni a questi problemi possono essere sviluppate utilizzando gli strumenti presentati nel Capitolo 6, tra cui i lock mutex, i semafori, i monitor e le variabili condizionali.
- Windows utilizza oggetti dispatcher ed eventi per implementare gli strumenti di sincronizzazione dei processi.
- Linux utilizza diversi approcci per proteggere dalle race condition, incluse le variabili atomiche, gli spinlock e i lock mutex.
- L'api posix fornisce i lock mutex, i semafori e le variabili condizionali. posix fornisce due tipi di semafori: con nome e senza nome. Diversi processi non correlati possono facilmente accedere allo stesso semaforo con nome facendo riferimento al suo nome. I semafori senza nome non possono essere condivisi con la stessa facilità: per rendere possibile la condivisione è necessario collocare il semaforo in una regione di memoria condivisa.
- Java ha una ricca libreria e una api per la sincronizzazione. Gli strumenti disponibili includono i monitor (forniti a livello di linguaggio) e i lock rientranti, i semafori e le variabili condizionali (supportati dall'api).
- Tra gli approcci alternativi alla soluzione del problema della sezione critica vi sono la memoria transazionale, Openmp e i linguaggi funzionali. I linguaggi funzionali sono particolarmente interessanti, perché offrono un paradigma di programmazione diverso dai linguaggi procedurali. A differenza dei linguaggi procedurali, i linguaggi funzionali non mantengono lo stato e sono quindi generalmente immuni dalle race condition e dalle sezioni critiche.

Esercizi di ripasso

7.1 Spiegate perché Windows e Linux implementano multipli meccanismi di locking. Descrivete le circostanze in cui questi sistemi operativi utilizzano spinlock, mutex, semafori, lock mutex adattivi e variabili condizionali. Spiegate, in ciascun caso, perché occorre un tale meccanismo.

7.2 Windows offre uno strumento di sincronizzazione leggero chiamato lock srw (slim reader/writer, *lettore-scrittore leggero*). Mentre la maggior parte delle implementazioni dei lock lettore-scrittore favorisce i lettori o gli scrittori, oppure ordina i thread in attesa usando una politica fifo, i lock srw non favoriscono né lettori né scrittori, né tantomeno ordinano i thread in attesa in una coda fifo. Spiegate i vantaggi di un tale strumento di sincronizzazione.

7.3 Descrivete quali modifiche sono necessarie ai processi produttore e consumatore nelle Figure 7.1 e 7.2 per utilizzare un lock mutex invece di un semaforo binario.

7.4 Descrivete come può verificarsi una situazione di stallo nel problema dei dining philosophers.

7.5 Spiegate la differenza tra stati signaled e nonsignaled degli oggetti dispatcher di Windows.

7.6 Supponete che `val` sia un intero atomico in un sistema Linux. Qual è il valore di `val` dopo che sono state completate le seguenti operazioni?

```
atomic_set(&val,10);

atomic_sub(8,&val);

atomic_inc(&val);

atomic_inc(&val);

atomic_add(6,&val);

atomic_sub(3,&val);
```

Esercizi

7.7 Descrivete due strutture dati di un kernel in cui possono verificarsi le cosidette race condition. Assicuratevi di includere una descrizione della modalità in cui queste situazioni possono verificarsi.

7.8 Nel kernel di Linux un processo non può trattenere uno spinlock mentre tenta di acquisire un semaforo. Giustificate l'esistenza di questa politica di gestione.

7.9 Progettate un algoritmo per un monitor a buffer limitato in cui i buffer siano incorporati nel monitor stesso.

7.10 All'interno di un monitor, la mutua esclusione stretta fa sì che il monitor con memoria limitata dell'Esercizio 7.9 sia adatto soprattutto buffer piccoli.

a. Spiegate perché questa affermazione è vera.

b. Progettate un nuovo schema idoneo a buffer grandi.

7.11 Discutete il tradeoff tra equità e throughput delle operazioni nel problema dei lettori-scrittori. Proponete un metodo per risolvere il problema dei lettori-scrittori senza che si determini attesa indefinita.

7.12 Spiegate perché la chiamata al metodo `lock()` in un `ReentrantLock` di Java non viene inserita nella clausola `try` per la gestione delle eccezioni ma, piuttosto, è inserita in una clausola `finally`.

7.13 Spiegate la differenza tra la memoria transazionale software e la memoria transazionale hardware.

7.14 L'Esercizio 3.20 richiedeva di progettare un gestore dei pid cheassegnasse un identificatore di processo univoco per ogni processo. L'Esercizio 4.28 richiedeva di modificare la vostra soluzione all'Esercizio 3.20 scrivendo un programma che creava un certo numero di thread che richiedeva e rilasciava identificativi di processo. Utilizzando dei lock mutex modificate la vostra soluzione all'Esercizio 4.28, in modo da garantire che la struttura dei dati utilizzata per rappresentare la disponibilità degli identificatori di processo sia al sicuro da race condition.

7.15 Nell'Esercizio 4.27 avete scritto un programma per generare la sequenza di Fibonacci. Il programma richiedeva che il thread padre attendesse che il thread figlio finisse la sua esecuzione prima di stampare i valori calcolati. Se lasciamo che il thread padre acceda ai numeri di Fibonacci non appena sono stati calcolati dal thread figlio – piuttosto che aspettare che il thread figlio termini – quali cambiamenti sarebbero necessari alla soluzione dell'esercizio? Implementate la vostra soluzione modificata.

7.16 Il programma C `stack-ptr.c` (disponibile nel codice sorgente da scaricare) contiene un'implementazione di uno stack che utilizza una lista linkata. Un esempio del suo utilizzo è il seguente:

```
StackNode *top = NULL;

push(5, &top);

push(10, &top);

push(15, &top);

int value = pop(&top);

value = pop(&top);

value = pop(&top);
```

Questo programma al momento presenta una corsa critica e non è appropriato per un ambiente concorrente. Usando i lock mutex Pthreads (descritti nel Paragrafo 7.3.1), risolvete la race condition.

7.17 L'Esercizio 4.24 chiedeva di progettare un programma multithreading che stimasse π utilizzando la tecnica Monte Carlo. In quell'esercizio vi era stato richiesto di creare un singolo thread che generava punti casuali, memorizzando il risultato in una variabile globale. Una volta terminato quel thread, il thread principale eseguiva il calcolo che stima il valore di π . Modificate quel programma in modo da creare diversi thread, ciascuno dei quali genera punti casuali e determini se i punti rientrano nel cerchio. Ogni thread dovrà

aggiornare il numero globale di punti che rientra nel cerchio. Proteggetevi contro le race condition sugli aggiornamenti della variabile globale condivisa utilizzando i lock mutex.

7.18 L'Esercizio 4.25 chiedeva di progettare, usando OpenMP, un programma che stimasse π usando la tecnica Monte Carlo. Esaminate la vostra soluzione per quel programma alla ricerca di eventuali corse critiche. Se si identifica una race condition proteggetevi contro di essa usando la strategia descritta nel Paragrafo 7.5.2.

7.19 Una *barriera* è uno strumento per sincronizzare l'attività di un certo numero di thread. Quando un thread raggiunge un punto di barriera, non può procedere finché tutti gli altri thread non hanno raggiunto anch'essi questo punto. Quando l'ultimo thread raggiunge il punto di barriera, tutti i thread vengono rilasciati e possono riprendere l'esecuzione concorrente. Supponiamo che la barriera sia inizializzata a N – il numero di thread che devono attendere al punto di barriera:

```
init (N);
```

Ogni thread quindi esegue una certa quantità di lavoro fino a raggiungere il punto di barriera:

```
/* esegue un po' di lavoro */

barrier point();

/* esegue un po' di lavoro */
```

Utilizzando gli strumenti di sincronizzazione posix o Java descritti in questo capitolo creare una barriera che implementi la seguente api:

- `int init(int n)` – Inizializza la barriera alla dimensione specificata.
- `int barrier_point(void)` – identifica il punto di barriera. Tutti i thread vengono rilasciati dalla barriera quando l'ultimo thread raggiunge questo punto.

Il valore restituito da ciascuna funzione viene utilizzato per identificare le condizioni di errore. Ogni funzione restituirà 0 in condizioni normali e restituirà -1 se si verifica un errore. Un codice di test è fornito nel codice sorgente scaricabile, per testare la vostra implementazione della barriera.

Progetto 1 – Progettazione di un thread pool

I pool di thread o thread pool sono stati introdotti nel Paragrafo 4.5.1. Quando vengono utilizzati i thread pool, un task viene inviato al pool ed eseguito da un thread del pool. Il task viene inviato al pool utilizzando una coda e un thread disponibile rimuove il task dalla coda. Se non ci sono thread disponibili, il task rimane in coda fino a quando uno diventa disponibile. Se non c'è lavoro da compiere, i thread attendono le notifiche finché un task non diventi disponibile.

Questo progetto prevede la creazione e la gestione di un pool di thread e può essere realizzato utilizzando le primitive di sincronizzazione Pthread posix oppure Java. Di seguito forniamo i dettagli relativi a ciascuna specifica tecnologia.

I. POSIX

La versione posix di questo progetto comporterà la creazione di un numero di thread utilizzando l'api Pthreads e utilizzando i lock mutex e i semafori posix per la sincronizzazione.

Il client

Gli utenti del thread pool utilizzeranno la seguente api:

- `void pool_init()` – Inizializza il thread pool.
- `int pool_submit(void (*somefunction) (void *p), void *p)` – dove `somefunction` è un puntatore alla funzione che verrà eseguita da un thread del pool e `p` è un parametro passato alla funzione.
- `void pool_shutdown(void)` – chiude il thread pool una volta che tutti i task hanno completato.

Forniamo un esempio di programma `client.c` nel codice sorgente scaricabile, che illustra come usare il thread pool utilizzando queste funzioni.

Implementazione del thread pool

Nel codice sorgente scaricabile forniamo il file C `threadpool.c` come implementazione parziale del thread pool. Dovrete implementare il funzioni chiamate dai client, oltre a numerose funzioni aggiuntive che supportano i componenti interni del thread pool. L'implementazione richiederà le seguenti attività.

1. La funzione `pool_init()` creerà i thread all'avvio come pure inizializzerà lock mutex e semafori.

2. La funzione `pool_submit()` è parzialmente implementata e attualmente posiziona la funzione da eseguire, nonché i suoi dati, in una struct task. La struct task rappresenta il lavoro che verrà completato da un thread del pool. `pool_submit()` aggiungerà questi task alla coda invocando la funzione `enqueue()` e i thread di lavoro chiameranno `dequeue()` per recuperare il lavoro dalla coda. La coda può essere implementata staticamente (usando array) o dinamicamente (usando una lista linkata).

La funzione `pool_init()` restituisce un valore `int` utilizzato per indicare se il task è stato inviato correttamente al pool (0 indica successo, 1 indica fallimento). Se la coda è implementata usando array, `pool_init()` restituirà 1 se c'è un tentativo di inviare un lavoro e la coda è piena. Se la coda è implementata come una lista linkata, `pool_init()` dovrebbe sempre restituire 0, a meno che non si verifichi un errore di allocazione di memoria.

3. La funzione `worker()` viene eseguita da ciascun thread nel pool, dove ciascuno thread attenderà il lavoro disponibile. Una volta che il lavoro diventa disponibile, il thread lo rimuoverà dalla coda e invocherà `execute()` per eseguire la funzione specificata.

Un semaforo può essere usato per inviare una notifica a un thread in attesa quando un lavoro viene inviato al thread pool. Possono essere usati semafori con o senza nome. Fate riferimento al Paragrafo 7.3.2 per ulteriori dettagli sull'uso dei semafori posix.

4. Un lock mutex è necessario per evitare corse critiche durante l'accesso o la modifica della coda. (Il Paragrafo 7.3.1 fornisce dettagli sui lock mutex Pthreads.)

5. La funzione `pool_shutdown()` cancellerà tutti i thread di lavoro e quindi dovrete attendere che ogni thread termini chiamando `pthread_join()`. Fate riferimento al Paragrafo 4.6.3 per dettagli sulla cancellazione dei thread posix. (L'operazione del semaforo `sem_wait()` è un punto di cancellazione che consente di cancellare un thread in attesa su un semaforo.)

Fate riferimento al codice sorgente scaricabile per ulteriori dettagli su questo progetto. In particolare, il file `README` descrive i file sorgente e di intestazione, oltre al `Makefile` per la realizzazione del progetto.

II. Java

La versione Java di questo progetto può essere completata utilizzando gli strumenti di sincronizzazione Java come descritto nel Paragrafo 7.4. La sincronizzazione può essere realizzata da (a) monitor che usano `synchronized/wait()/notify()` (Paragrafo 7.4.1) o (b) semafori e lock rientranti (Paragrafo 7.4.2 e Paragrafo 7.4.3).

I thread Java sono descritti nel Paragrafo 4.4.3.

Implementazione del thread pool

Il vostro thread pool implementerà la seguente api:

- `ThreadPool()` – Crea un pool di thread di dimensioni predefinite.
- `ThreadPool(int size)` – Crea un pool di thread di dimensione size.
- `void add(Runnable task)` – Aggiunge un'attività da eseguire da un thread del pool.
- `void shutdown()` – Interrompe tutti i thread nel pool.

Nel codice sorgente scaricabile forniamo il file sorgente Java `ThreadPool.java` come implementazione parziale del thread pool. Dovrete implementare i metodi che vengono chiamati dai client, oltre a molti altri metodi che supportano i componenti interni del thread pool. L'implementazione richiederà le seguenti attività.

1. Il costruttore creerà innanzitutto un numero di thread inattivi che attendono un lavoro.

2. Il lavoro verrà inviato al pool tramite il metodo `add()`, che aggiunge un task che implementa l'interfaccia `Runnable`. Il metodo `add()` posizionerà il task `Runnable` in una coda (è possibile utilizzare una struttura disponibile nell'api Java come `java.util.List`).

3. Una volta che un thread nel pool diventa disponibile per un lavoro, controllerà la coda dei task `Runnable`. Se è presente un task, il thread inattivo rimuoverà il task dalla coda e invocherà il suo metodo `run()`. Se la coda è vuota, il thread inattivo aspetterà di essere avvisato quando un lavoro diventa disponibile. (Il metodo `add()` può implementare le notifiche usando `notify()` o le funzioni di un semaforo quando colloca un task `Runnable` in coda in modo da risvegliare, potenzialmente, un thread inattivo in attesa di lavoro.)

4. Il metodo `shutdown()` interromperà tutti i thread nel pool richiamando il loro metodo `interrupt()`. Questo, ovviamente, richiede che i task `Runnable` in esecuzione nel thread pool controllino il loro stato di interruzione (Paragrafo 4.6.3).

Fate riferimento al codice sorgente scaricabile per ulteriori dettagli su questo progetto. In particolare, il file `README` descrive i file sorgente Java, oltre che ulteriori dettagli sull'interruzione dei thread Java.

Progetto 2 – L'assistente che dorme

Un dipartimento di informatica ha un assistente (ta) che aiuta gli studenti universitari nei loro esercizi di programmazione, in orario d'ufficio. L'ufficio del ta è piuttosto piccolo e ha spazio solo per una scrivania con una sedia e per un computer. Ci sono tre sedie in corridoio, davanti all'ufficio, dove gli studenti possono sedersi e aspettare quando il ta è impegnato ad aiutare un altro studente. Quando non ci sono studenti che hanno bisogno di aiuto, il ta si siede alla scrivania e fa un pisolino. Se uno studente arriva in orario d'ufficio e trova il ta che dorme, deve sveglierlo per chiedergli aiuto. Se uno studente arriva e trova il ta che sta aiutando un altro studente, si siede su una delle sedie in corridoio e aspetta. Se non ci sono sedie disponibili, lo studente dovrà tornare in un secondo momento.

Utilizzando i thread posix, i lock mutex e i semafori implementate una soluzione per coordinare le attività del ta e degli studenti. Riportiamo di seguito i dettagli di questo progetto.

Gli studenti e il TA

Utilizzando Pthread (Paragrafo 4.4.1) iniziate dalla creazione di n studenti. Ogni studente e il ta verranno eseguiti come thread distinti. I thread degli studenti si alterneranno fra un periodo di tempo di programmazione e un periodo alla ricerca di aiuto dal ta. Se il ta è disponibile, otterranno aiuto. In caso contrario, si siederanno su una sedia nel corridoio o, in mancanza di sedie disponibili, riprenderanno l'attività di programmazione e cercheranno aiuto in un secondo momento. Se uno studente arriva e si accorge che il ta sta dormendo, deve notificare il suo arrivo al ta con un semaforo. Quando il ta finisce di aiutare uno studente deve controllare in corridoio per vedere se ci sono studenti in attesa di aiuto. In caso affermativo, il ta deve aiutare, a turno, tutti gli studenti. Se non sono presenti studenti, il ta può tornare a dormire. Probabilmente l'opzione migliore per simulare gli studenti che programmano e il ta che fornisce aiuto a uno studente è di sospendere i thread coinvolti per un periodo di tempo casuale.

I lock mutex e i semafori posix sono trattati nel Paragrafo 7.3. Consultate il paragrafo per ulteriori dettagli.

Progetto 3 – Il problema dei cinque filosofi (dining philosophers)

Nel Paragrafo 7.1.3 abbiamo fornito uno schema di soluzione al problema dei dining philosophers con l'utilizzo dei monitor. Questo progetto richiede l'implementazione di una soluzione usando i lock mutex posix e le variabili condizionali oppure le variabili condizionali Java. Le soluzioni saranno basate sull'algoritmo illustrato nella Figura 7.7.

Entrambe le implementazioni richiederanno la creazione di cinque filosofi, ciascuno identificato da un numero 0 . . 4. Ogni filosofo verrà eseguito come un thread separato. I filosofi alternano le attività di pensare e mangiare. Sospendete i thread per un periodo di tempo casuale compreso tra uno e tre secondi.

I. POSIX

La creazione di thread con l'uso di Pthreads è trattata nel Paragrafo 4.4.1. Quando un filosofo vuole mangiare, invoca la funzione

```
pickup_forks (int philosopher_number)
```

dove `philosopher_number` identifica il numero del filosofo che desidera mangiare. Quando un filosofo finisce di mangiare, invoca

```
return_forks (int philosopher_number)
```

La vostra implementazione richiederà l'uso di variabili condizionali posix, spiegate nel Paragrafo 7.3.

II. Java

Quando un filosofo desidera mangiare, invoca il metodo `Forks(philosopherNumber)`, dove `philosopherNumber` identifica il numero del filosofo che desidera mangiare. Quando un filosofo finisce di mangiare, invoca `returnForks(philosopherNumber)`.

La vostra soluzione implementerà la seguente interfaccia:

```

public interface DiningServer

{

    /* chiamato da un filosofo che vuole mangiare */

    public void takeForks(int philosopherNumber);

    /* chiamato da un filosofo quando ha finito di mangiare */

    public void returnForks(int philosopherNumber);

}

```

Richiederà l'utilizzo di variabili condizionali Java, illustrate nel Paragrafo 7.4.4.

Progetto 4 – Il problema produttore/consumatore

Nel Paragrafo 7.1.1 abbiamo proposto una soluzione al problema produttore/consumatore che si basa su semafori e utilizza un buffer limitato. In questo progetto daremo soluzione al problema del buffer limitato mediante i processi produttore e consumatore schematizzati nelle Figure 5.9 e 5.10, rispettivamente. La soluzione presentata nel Paragrafo 7.1.1 utilizza tre semafori, `empty` e `full`, che enumerano le posizioni vuote e piene del buffer, e `mutex`, un semaforo binario (ossia a mutua esclusione), che protegge l'inserimento nel (o l'estrazione dall') buffer. In questo progetto i semafori `empty` e `full` saranno semafori contatori standard, mentre, per rappresentare `mutex`, si userà un lock mutex in luogo di un semaforo binario. Il produttore e il consumatore – eseguiti come thread separati – trasferiscono gli oggetti a/da un buffer sincronizzato con le strutture `empty`, `full` e `mutex`. Potete scegliere di risolvere il problema con Pthreads o con la api Windows.

Il buffer

Internamente, il buffer è costituito da un array di dimensione fissa avente tipo `buffer_item`, definito tramite `typedef`. L'array di oggetti `buffer_item` sarà trattato come una coda circolare. Definizione e dimensione relative possono essere poste in un file di intestazione come il seguente:

```

/* buffer.h */

typedef int buffer_item;

#define BUFFER_SIZE 5

```

Il buffer sarà manipolato da due funzioni, `insert_item()` e `remove_item()`, richiamate, rispettivamente, dal thread del produttore e dal thread del consumatore. Tali funzioni, in un primo abbozzo schematico, appaiono nella Figura 7.14.

```

#include "buffer.h"

/* il buffer */

buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {

    /* inserisce un elemento nel buffer

    restituisce 0 se ha successo, altrimenti

    restituisce -1 che indica una condizione di errore */

}

```

```

int remove_item(buffer_item *item) {

    /* rimuove un elemento dal buffer

    restituisce 0 se ha successo, altrimenti

    restituisce -1 che indica una condizione di errore */

}

```

Figura 7.14 Schematizzazione delle funzioni del buffer.

Le funzioni `insert_item()` e `remove_item()` sincronizzeranno il produttore e il consumatore usando gli algoritmi delineati nelle Figure 7.1 e 7.2. Il buffer richiederà, inoltre, una funzione per inizializzare l'oggetto `mutex`, come pure i semafori `empty` e `full`.

La funzione `main()` inizializzerà il buffer, creando due thread distinti per il produttore e il consumatore. Dopo aver creato tali thread, la funzione `main()` rimarrà inattiva per un certo tempo e, alla ripresa dell'attività, terminerà l'applicazione. La funzione `main()` riceverà tre parametri dalla riga di comando, indicanti:

1. il tempo di inattività prima di terminare;
2. il numero di thread produttori;
3. il numero di thread consumatori.

La struttura di questa funzione è illustrata nella Figura 7.15.

```

#include "buffer.h"

int main(int argc, char *argv[]) {

    /* 1. Riceve da linea di comando gli argomenti argv[1],argv[2],argv[3] */

    /* 2. Inizializza il buffer */

    /* 3. Crea i thread produttori */

    /* 4. Crea i thread consumatori */

    /* 5. Entra in sleep */

    /* 6. Esce */
}

```

Figura 7.15 Schematizzazione del main.

Thread produttori e consumatori

Il thread produttore si alternerà fra periodi di tempo di inattività casuali e l'inserimento di un intero casuale nel buffer. I numeri casuali saranno generati tramite la funzione `rand()`, che genera valori interi casuali compresi tra 0 e `RAND_MAX`. Anche il consumatore rimarrà in stato di inattività per un intervallo di tempo casuale; tornato attivo, tenterà di estrarre un oggetto dal buffer. Lo schema dei thread produttori e consumatori è illustrato nella Figura 7.16.

```

#include <stdlib.h> /* necessaria per rand() */

#include "buffer.h"

void *producer(void *param) {

    buffer_item item;

    while (true) {

        /* inattivo per un periodo di tempo casuale */

        sleep(...);

        /* genera un numero casuale */

        item = rand();

        if (insert_item(item))

            fprintf("condizione di errore");

        else

            printf("il produttore ha prodotto %d\n");

    }

}

void *consumer(void *param) {

    buffer_item item;

    while (true) {

        /* inattivo per un periodo di tempo casuale */

        sleep(...);

        if (remove_item(&item))

            fprintf("condizione di errore");

        else

            printf("il consumatore ha consumato %d\n");

    }

}

```

Figura 7.16 Schematizzazione dei thread produttore e consumatore.

Come osservato in precedenza, è possibile risolvere questo problema utilizzando Pthreads oppure l'api di Windows. Nei paragrafi seguenti forniamo ulteriori informazioni su ciascuna di queste scelte.

Creazione e sincronizzazione dei thread in Pthreads

La creazione di thread utilizzando l'api Pthreads è trattata nel Paragrafo 4.4.1. I lock mutex e i semafori in Pthreads sono introdotti nel Paragrafo 7.3. Fate riferimento a questi paragrafi per informazioni specifiche sulla creazione e sulla sincronizzazione dei thread in Pthreads.

Thread di Windows

Il Paragrafo 4.4.2 descrive la creazione di thread con l'api Windows. Fate riferimento a quel paragrafo per indicazioni specifiche sulla creazione dei thread.

Lock mutex di Windows

I lock mutex sono oggetti dispatcher, come si è visto nel Paragrafo 7.2.1. Di seguito si può osservare come creare un lock mutex utilizzando la funzione `CreateMutex()`:

```
#include <windows.h>

HANDLE Mutex;

Mutex = CreateMutex(NULL, FALSE, NULL);
```

Il primo parametro si riferisce a un attributo di sicurezza per il lock mutex. Impostando a `NULL` questo attributo, i processi figli del creatore del lock mutex non ereditano il riferimento al mutex stesso. Il secondo parametro indica se il processo che ha creato il lock mutex sia il possessore iniziale del lock: passando il valore `FALSE`, si asserisce che il thread non è il possessore iniziale; vedremo fra breve come acquisire i lock mutex. Il terzo parametro consente di attribuire un nome al mutex. Tuttavia, poiché si è scelto un valore `NULL`, non gli attribuiremo un nome. Se `CreateMutex()` ha avuto successo, restituirà il riferimento `HANDLE` al lock; altrimenti, `NULL`.

Nel Paragrafo 7.2.1 abbiamo descritto i due possibili stati, detti *signaled* e *nonsignaled*, degli oggetti dispatcher. Quando un oggetto dispatcher (per esempio un lock mutex) è nello stato *signaled* se ne può entrare in possesso; una volta acquisito passa allo stato *nonsignaled*. L'oggetto diviene *signaled* non appena viene rilasciato.

I lock mutex si acquisiscono richiamando la funzione `WaitForSingleObject()` con due parametri: il riferimento `HANDLE` al lock, e un flag che indica la durata dell'attesa. Il codice mostra come si può acquisire il lock mutex creato in precedenza:

```
WaitForSingleObject (Mutex, INFINITE);
```

Il valore `INFINITE` significa che non vi è limite a quanto si è disposti ad attendere che il lock diventi disponibile. Con altri valori il thread chiamante potrebbe troncare l'operazione, qualora il lock non tornasse disponibile entro un tempo definito. Se il lock è in stato *signaled*, `WaitForSingleObject()` ritorna immediatamente, e il lock diviene *nonsignaled*. Per rendere disponibile un lock (ossia, per far sì che entri nello stato *signaled*) si invoca `ReleaseMutex()`, come nel seguito:

```
ReleaseMutex (Mutex);
```

Semafori di Windows

Nella api di Windows anche i semafori sono oggetti dispatcher e pertanto adoperano lo stesso meccanismo di segnalazione dei lock mutex. I semafori si creano tramite il codice:

```
#include <windows.h>

HANDLE Sem;

Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

In analogia a quanto descritto per i lock mutex, il primo e l'ultimo parametro designano un attributo di sicurezza e un nome per il semaforo. Il secondo e il terzo parametro indicano il valore iniziale e il valore massimo del semaforo. In questo caso, il valore iniziale del semaforo è 1, mentre il suo valore massimo è 5. Se ha successo, `CreateSemaphore()` restituisce un riferimento `HANDLE` al semaforo; in caso contrario, restituisce `NULL`.

I semafori si acquisiscono con la stessa funzione trattata per i lock mutex, cioè `WaitForSingleObject()`. Per acquisire il semaforo `Sem` di questo esempio, si invoca:

```
WaitForSingleObject (Semaphore, INFINITE);
```

Se il valore del semaforo è > 0, esso si trova nello stato *signaled*, e viene quindi acquisito dal thread chiamante; altrimenti, il thread chiamante è sospeso (per un tempo impreciso, a causa del parametro `INFINITE`) in attesa che il semaforo ritorni nello stato *signaled*.

Per i semafori di Windows, l'equivalente dell'operazione `signal()` è la funzione `ReleaseSemaphore()`. Essa accetta tre parametri:

1. il riferimento `HANDLE` al semaforo;
2. il valore di cui incrementare il semaforo;
3. un puntatore al valore precedente del semaforo.

Si può aumentare `Sem` di 1 nel modo seguente:

```
ReleaseSemaphore(Sem, 1, NULL);
```

`ReleaseSemaphore()` e `ReleaseMutex()` restituiscono un valore non nullo se eseguite senza errori e 0 in caso contrario.

CAPITOLO 8

Stallo dei processi

In un ambiente con multiprogrammazione più thread possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il thread richiedente passa allo stato d'attesa. In alcuni casi, se le risorse richieste sono trattenute da altri thread, a loro volta nello stato d'attesa, il thread potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di stallo (*deadlock*). Questo argomento è stato trattato brevemente anche nel Capitolo 6 come un problema di liveness. In quel capitolo abbiamo definito lo stallo come una situazione in cui *ciascun processo in un insieme di processi attende un evento che può essere causato solo da un altro processo dell'insieme*.

Un efficace esempio di situazione di stallo si può ricavare da una legge dello stato del Kansas approvata all'inizio del ventesimo secolo, che in una sua parte recita: "Quando due treni convergono a un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato".

In questo capitolo si descrivono i metodi che un sistema operativo può usare per prevenire o affrontare le situazioni di stallo. Anche se esistono applicazioni che possono identificare i programmi suscettibili di stallo, la maggior parte dei sistemi operativi attuali non offre strumenti di prevenzione di queste situazioni, e progettare programmi che non rischiano lo stallo rimane una responsabilità dei programmatore. Date le tendenze attuali, con la richiesta sempre maggiore di concorrenza e parallelismo nei sistemi multicore, il problema dello stallo, così come altri problemi di liveness, può diventare soltanto più importante.

8.1 Modello di sistema

Un sistema è composto da un numero finito di risorse da distribuire tra più thread in competizione. Le risorse possono essere suddivise in tipi (o classi) differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di cpu, file e dispositivi di i/o (come interfacce di rete e lettori dvd), sono tutti esempi di tipi di risorsa. Se un sistema ha quattro cpu, il tipo di risorsa *cpu* ha quattro istanze. Analogamente, il tipo di risorsa *rete* può avere due istanze. Se un thread richiede un'istanza relativa a un tipo di risorsa, l'assegnazione di *qualsiasi* istanza di quel tipo dovrebbe soddisfare la richiesta. Se ciò non si verifica significa che le istanze non sono identiche e le classi di risorse non sono state definite correttamente.

Nel Capitolo 6 abbiamo descritto vari strumenti di sincronizzazione, tra cui i lock mutex e i semafori. Anche questi strumenti sono risorse di sistema e sono una tipica causa di situazioni di stallo nei moderni sistemi elaborativi. Tuttavia, in questo caso non si hanno problemi di definizione. Un lock è solitamente destinato alla protezione di una specifica struttura dati, per esempio un lock può essere utilizzato per proteggere l'accesso a una coda, un altro per proteggere l'accesso a una lista concatenata, e così via. Per questo motivo, a ogni lock viene tipicamente assegnata la propria classe di risorse.

Si noti che in questo capitolo trattiamo le risorse del kernel, ma i thread possono anche usare risorse di altri processi (per esempio per mezzo della comunicazione tra processi) e l'utilizzo di tali risorse può anch'esso portare a situazioni di stallo. Gli stalli di questo tipo non riguardano il kernel e non saranno dunque discussi in questo capitolo.

Prima di adoperare una risorsa, un thread deve richiederla e, dopo averla usata, deve rilasciarla. Un thread può richiedere tutte le risorse necessarie per eseguire il compito assegnatogli, anche se ovviamente il numero delle risorse richieste non può superare quello totale delle risorse disponibili nel sistema: per esempio un thread non può richiedere due interfacce di rete se il sistema ne ha solo una.

Nelle ordinarie condizioni di funzionamento un thread può servirsi di una risorsa soltanto se rispetta la seguente sequenza.

1. Richiesta. Il thread richiede la risorsa; se la richiesta non si può soddisfare immediatamente – per esempio, se un lock mutex è attualmente in possesso di un altro thread – il thread richiedente deve attendere finché non può acquisire tale risorsa.
2. Uso. Il thread può operare sulla risorsa (se, per esempio, la risorsa è un lock mutex, il thread può accedere alla regione critica).
3. Rilascio. Il thread rilascia la risorsa.

La richiesta e il rilascio di risorse avvengono tramite chiamate di sistema, come illustrato nel Capitolo 2. Ne sono esempi le chiamate di sistema `request()` e `release()` di una periferica, `open()` e `close()` di un file, `allocate()` e `free()` di una porzione di memoria. In modo simile, come descritto nel Capitolo 6, la richiesta e il rilascio si possono eseguire per mezzo delle operazioni `wait()` e `signal()` dei semafori o con l'utilizzo delle funzioni `acquire()` e `release()` dei lock mutex. Quindi, ogni volta che si usa una risorsa gestita dal kernel, il sistema operativo controlla che il thread utente ne abbia fatto richiesta e che questa gli sia stata assegnata. Una tabella di sistema registra lo stato di ogni risorsa e, se questa è assegnata, indica il thread relativo. Se un thread richiede una risorsa già assegnata a un altro thread, il thread richiedente può essere accodato agli altri thread che attendono tale risorsa.

Un gruppo di thread si trova in stallo quando ogni thread del gruppo attende un evento che può essere causato solo da un altro thread che si trova nel gruppo. Gli eventi che interessano maggiormente in questo contesto sono l'acquisizione e il rilascio di risorse. Le risorse sono tipicamente logiche (per esempio lock mutex, semafori e file); tuttavia, vi sono altri tipi di eventi che possono causare deadlock, come la lettura da una interfaccia di rete o l'utilizzo delle funzionalità di ipc discusse nel Capitolo 3.

Per esaminare una situazione di stallo si consideri il problema dei dining philosophers, discusso nel Paragrafo 7.1.3. In questa situazione, le risorse sono rappresentate dalle baccette. Se tutti i filosofi avessero fame allo stesso tempo e ogni filosofo afferrasse la baccetta alla sua sinistra, non ci sarebbero più baccette disponibili. Ogni filosofo viene quindi bloccato in attesa che la sua baccetta destra diventi disponibile.

Gli sviluppatori di applicazioni multithread devono essere consapevoli delle possibili situazioni di stallo. Gli strumenti di lock presentati nel Capitolo 6 sono progettati per evitare race condition, ma utilizzando questi strumenti gli sviluppatori devono prestare particolare attenzione a come i lock vengono acquisiti e rilasciati. In caso contrario possono verificarsi situazioni di stallo, come descritto nel seguito.

8.2 Situazioni di stallo in applicazioni multithread

Prima di esaminare come i problemi di stallo possano essere identificati e gestiti illustriamo come si possa incorrere in situazioni di stallo in un programma multithread di Pthread che usa i lock mutex. La funzione `pthread_mutex_init()` inizializza un mutex su cui non è attivo un lock. I lock mutex sono acquisiti e restituiti per mezzo di `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente. Se un thread tenta di acquisire un mutex già impegnato, l'invocazione di `pthread_mutex_lock()` blocca il thread finché il possessore del mutex non invochi `pthread_mutex_unlock()`.

Il segmento di codice seguente genera e inizializza due lock mutex:

```
pthread_mutex_t first_mutex;  
  
pthread_mutex_t second_mutex;  
  
pthread_mutex_init(&first_mutex,NULL);  
  
pthread_mutex_init(&second_mutex,NULL);
```

Si creano poi due thread, di nome `thread_one` e `thread_two`, che possono accedere a entrambi i lock mutex. I due thread sono eseguiti dalle funzioni `do_work_one()` e `do_work_two()`, rispettivamente, come mostrato nella Figura 8.1.

```
/* thread_one esegue in questa funzione */  
  
void *do_work_one(void *param)  
{  
  
    pthread_mutex_lock(&first_mutex);  
  
    pthread_mutex_lock(&second_mutex);  
  
    /**  
     * Fa qualcosa  
     */  
  
    pthread_mutex_unlock(&second_mutex);  
  
    pthread_mutex_unlock(&first_mutex);  
  
    pthread_exit(0);  
}  
  
/* thread_two esegue in questa funzione */  
  
void *do_work_two(void *param)  
{  
  
    pthread_mutex_lock(&second_mutex);  
  
    pthread_mutex_lock(&first_mutex);  
  
    /**  
     * Fa qualcosa  
     */  
  
}
```

```

pthread_mutex_unlock(&first_mutex);

pthread_mutex_unlock(&second_mutex);

pthread_exit(0);

}

```

Figura 8.1 Esempio di stallo.

In questo esempio `thread_one` tenta di acquisire i lock mutex nell'ordine (1) `first_mutex`, (2) `second_mutex`, mentre `thread_two` tenta di acquisire i lock nell'ordine (1) `second_mutex`, (2) `first_mutex`. Lo stallo è possibile se `thread_one` acquisisce `first_mutex` mentre `thread_two` acquisisce `second_mutex`.

Si noti che lo stallo, pur essendo possibile, non si verifica se `thread_one` è in grado di acquisire e rilasciare entrambi i lock prima che `thread_two` tenti a sua volta di impossessarsene. Naturalmente, l'ordine in cui i thread vengono eseguiti dipende da come vengono gestiti dallo scheduler della cpu. L'esempio evidenzia un problema importante per la gestione dello stallo: è difficile identificare e sottoporre a test gli stalli che si verificano solo in determinate condizioni di scheduling.

8.2.1 Situazioni di stallo attivo (livelock)

Lo stallo attivo (*livelock*) è un'altra forma di mancanza di liveness. È simile allo stallo, perché entrambi non permettono a due o più thread di procedere, ma nei due casi i motivi che portano i thread a non poter continuare sono differenti. Mentre uno stallo si verifica quando ogni thread di un insieme viene bloccato in attesa di un evento che può essere causato solo da un altro thread dell'insieme, lo stallo attivo si verifica quando un thread tenta continuamente un'azione che non ha successo. Lo stallo attivo è simile a ciò che a volte accade quando due persone si incontrano in un corridoio: uno si sposta alla sua destra, l'altro alla sua sinistra, e nessuna delle due persone può procedere. Poi una si sposta alla sua sinistra, e l'altra alla sua destra, e così via. Le due persone non sono bloccate, ma non stanno facendo alcun progresso nella direzione desiderata.

Un caso di stallo attivo può essere generato, per esempio, dalla funzione Pthreads `pthread_mutex_trylock()`, che tenta di acquisire un lock mutex senza bloccare il thread. Nel codice della Figura 8.2 riscriviamo l'esempio della Figura 8.1 in modo che ora usi `pthread_mutex_trylock()`. Questa situazione può portare a uno stallo attivo se `thread_one` acquisisce `first_mutex` e, in seguito, `thread_two` acquisisce `second_mutex`. Ogni thread richiama quindi la `pthread_mutex_trylock()`, che fallisce, e poi rilascia il lock acquisito in precedenza e ripete le stesse azioni indefinitamente.

```

/* thread_one esegue in questa funzione */

void *do_work_one(void *param)

{
    int done = 0;

    while (!done) {

        pthread_mutex_lock(&first_mutex);

        if (pthread_mutex_trylock(&second_mutex)) {

            /**
             * Fa qualcosa
             */
            pthread_mutex_unlock(&second_mutex);

            pthread_mutex_unlock(&first_mutex);
        }
    }
}

```

```

done = 1;

}

else

pthread_mutex_unlock(&first_mutex);

}

pthread_exit(0);

}

/* thread_two esegue in questa funzione */

void *do_work_two(void *param)

{

int done = 0;

while (!done) {

pthread_mutex_lock(&second_mutex);

if (pthread_mutex_trylock(&first_mutex)) {

/**/

* Fa qualcosa

*/



pthread_mutex_unlock(&first_mutex);

pthread_mutex_unlock(&second_mutex);

done = 1;

}

else

pthread_mutex_unlock(&second_mutex);

}

pthread_exit(0);

}

```

Figura 8.2 Esempio di stallo attivo.

Uno stallo attivo si verifica in genere quando i thread riprovano contemporaneamente le operazioni fallite. Pertanto, in generale, può essere evitato facendo in modo che ogni thread riprovi l'operazione fallita in momenti casuali. È proprio questo l'approccio adottato nelle reti Ethernet quando si verifica una collisione di rete. Invece di provare a ritrasmettere un pacchetto immediatamente dopo il verificarsi della collisione, un host coinvolto attenderà un periodo di tempo casuale prima di tentare di nuovo la trasmissione.

Il livelock è meno comune del deadlock, ma è comunque un problema complesso nella progettazione di applicazioni concorrenti e, come il deadlock, può verificarsi solo in determinate condizioni di scheduling.

8.3 Caratterizzazione delle situazioni di stallo

Nel paragrafo precedente abbiamo descritto come si possano verificare situazioni di stallo quando si utilizzano lock mutex nella programmazione multithread. Descriviamo ora in maniera più precisa le condizioni che generano una situazione di stallo.

8.3.1 Condizioni necessarie

Si può avere una situazione di stallo *solo se* in un sistema si verificano contemporaneamente le seguenti quattro condizioni.

1. Mutua esclusione. Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo thread alla volta.
Se un altro thread richiede tale risorsa, si deve ritardare il thread richiedente fino al rilascio della risorsa.
2. Possesso e attesa. Un thread deve essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri thread.
3. Assenza di prelazione. Le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal thread che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. Attesa circolare. Deve esistere un insieme $\{T_0, T_1, \dots, T_n\}$ di thread, tale che T_0 attende una risorsa posseduta da T_1 , T_1 attende una risorsa posseduta da T_2, \dots, T_{n-1} attende una risorsa posseduta da T_n e T_n attende una risorsa posseduta da T_0 .

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti; tuttavia è utile considerare separatamente ciascuna condizione, come vedremo nel Paragrafo 8.5.

8.3.2 Grafo di assegnazione delle risorse

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta grafo di assegnazione delle risorse. Si tratta di un insieme di vertici V e un insieme di archi E , con l'insieme di vertici V composto da due sottoinsiemi: $T = \{T_1, T_2, \dots, T_n\}$, che rappresenta tutti i thread del sistema, e $R = \{R_1, R_2, \dots, R_m\}$, che rappresenta tutti i tipi di risorsa del sistema.

Un arco diretto dal thread T_i al tipo di risorsa R_j si indica $T_i \rightarrow R_j$, e significa che il thread T_i ha richiesto un'istanza del tipo di risorsa R_j , e attualmente attende tale risorsa. Un arco diretto dal tipo di risorsa R_j al thread T_i si indica $R_j \rightarrow T_i$, e significa che un'istanza del tipo di risorsa R_j è assegnata al thread T_i . Un arco orientato $T_i \rightarrow R_j$ si chiama arco di richiesta, un arco orientato $R_j \rightarrow T_i$ si chiama arco di assegnazione.

Graficamente ogni thread T_i si rappresenta con un cerchio e ogni tipo di risorsa R_j si rappresenta con un rettangolo. Per esempio, il grafo di assegnazione delle risorse mostrato nella Figura 8.3 illustra la situazione di stallo relativa al programma nella Figura 8.1. Giacché il tipo di risorsa R_j può avere più di un'istanza, ciascuna di loro si rappresenta con un puntino all'interno del rettangolo. Occorre notare che un arco di richiesta è diretto soltanto verso il rettangolo R_j , mentre un arco di assegnazione deve designare anche uno dei puntini del rettangolo.

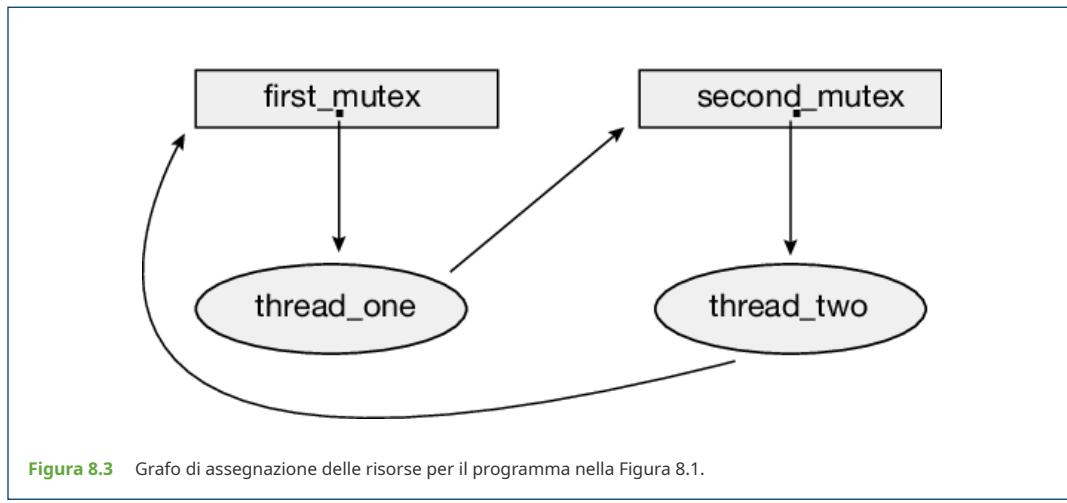


Figura 8.3 Grafo di assegnazione delle risorse per il programma nella Figura 8.1.

Quando il thread T_i richiede un'istanza del tipo di risorsa R_j , si inserisce un arco di richiesta nel grafo di assegnazione delle risorse. Se questa richiesta può essere esaudita, si trasforma *immediatamente* l'arco di richiesta in un arco di assegnazione, che al rilascio della risorsa viene cancellato.

Nel grafo di assegnazione delle risorse della Figura 8.4 è illustrata la seguente situazione.

- Insiemi T , R ed E :
- $T = \{T_1, T_2, T_3\}$

- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Istanze delle risorse:
- un'istanza del tipo di risorsa R_1
- due istanze del tipo di risorsa R_2
- un'istanza del tipo di risorsa R_3
- tre istanze del tipo di risorsa R_4
- Stati dei thread:
- il thread T_1 possiede un'istanza del tipo di risorsa R_2 e attende un'istanza del tipo di risorsa R_1
- il thread T_2 possiede un'istanza dei tipi di risorsa R_1 ed R_2 e attende un'istanza del tipo di risorsa R_3
- il thread T_3 possiede un'istanza del tipo di risorsa R_3

Data la definizione di grafo di assegnazione delle risorse, si può mostrare che, se il grafo non contiene cicli, nessun thread del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo.

Se ciascun tipo di risorsa ha esattamente un'istanza, allora l'esistenza di un ciclo implica la presenza di uno stallo; se il ciclo riguarda solo un insieme di tipi di risorsa, ciascuno dei quali ha solo un'istanza, si è verificato uno stallo. Ogni thread che si trovi nel ciclo è in stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria e sufficiente per l'esistenza di uno stallo.

Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente uno stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di uno stallo.

Per spiegare questo concetto conviene ritornare al grafo di assegnazione delle risorse della Figura 8.4. Si supponga che il thread T_3 richieda un'istanza del tipo di risorsa R_2 . Poiché attualmente non è disponibile alcuna istanza di risorsa, si aggiunge un arco di richiesta $T_3 \rightarrow R_2$ al grafo, com'è illustrato nella Figura 8.5. A questo punto nel sistema ci sono due cicli minimi:

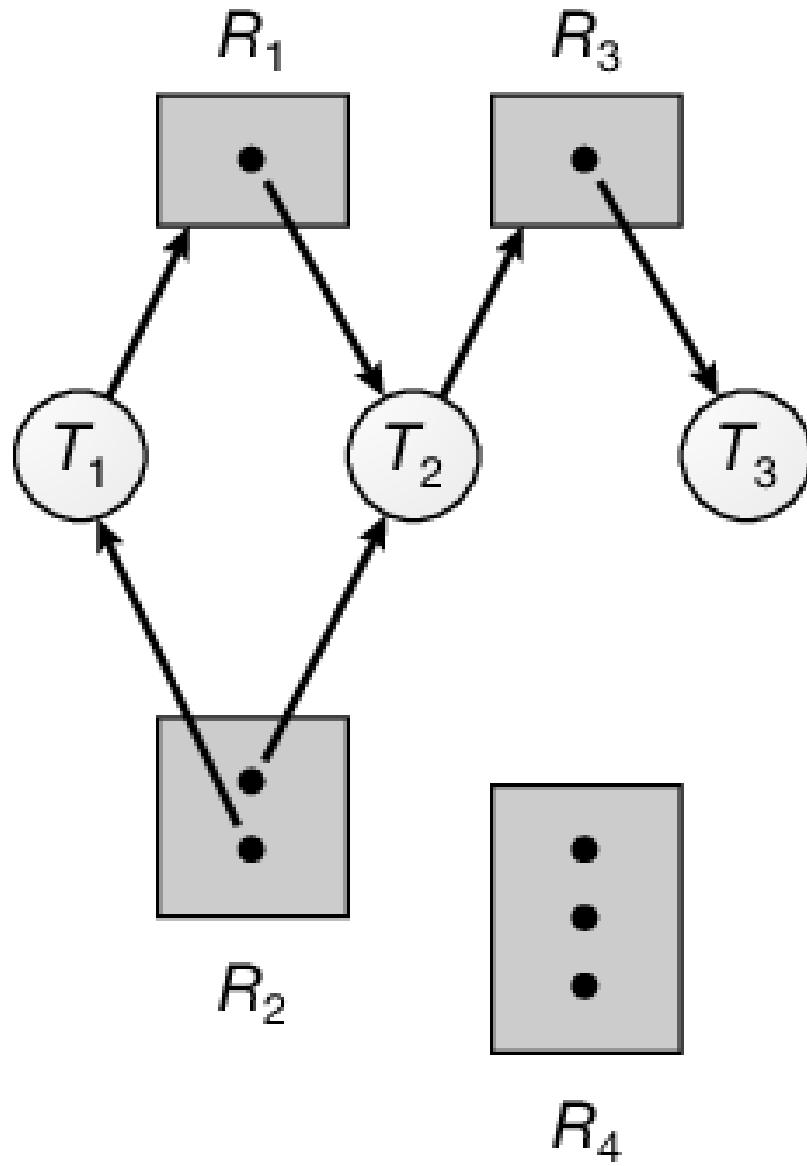


Figura 8.4 Grafo di assegnazione delle risorse.

- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

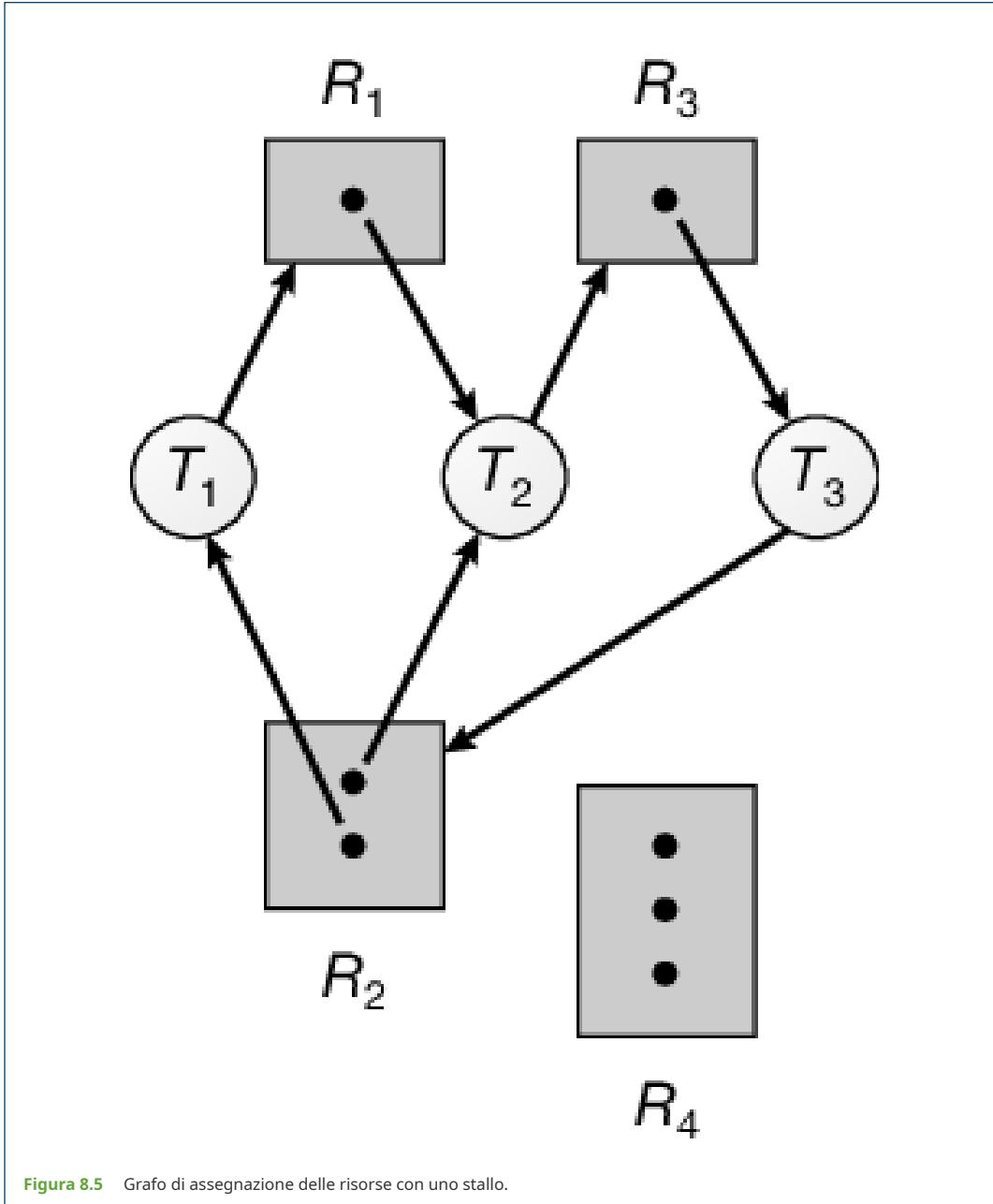
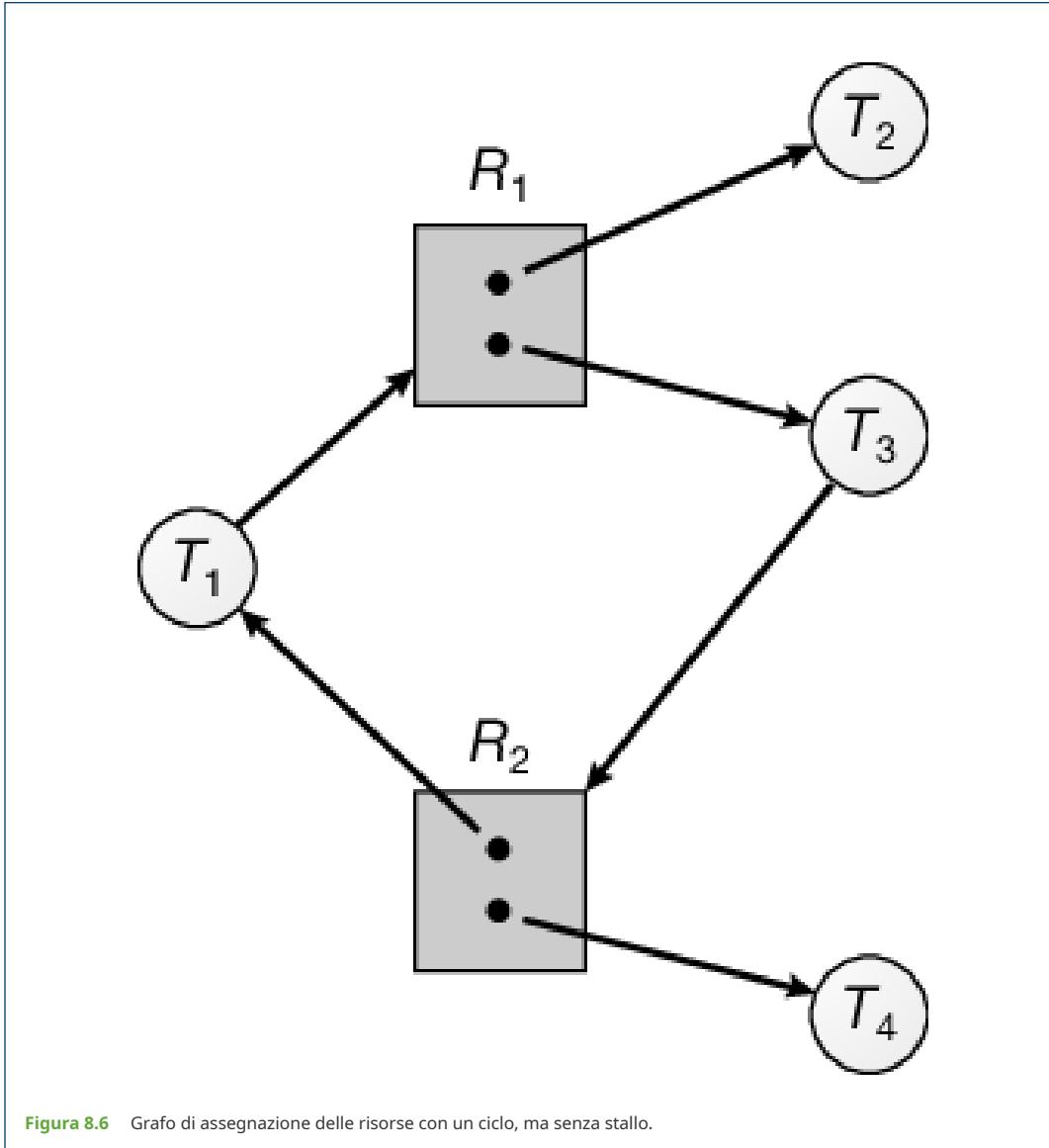


Figura 8.5 Grafo di assegnazione delle risorse con uno stallo.

I thread T_1 , T_2 e T_3 sono in stallo: il thread T_2 attende la risorsa R_3 , posseduta dal thread T_3 ; il thread T_3 , invece, attende che il thread T_1 o T_2 rilasci la risorsa R_2 ; inoltre il thread T_1 attende che il thread T_2 rilasci la risorsa R_1 .

Si consideri ora il grafo di assegnazione delle risorse della Figura 8.6. Anche in questo esempio c'è un ciclo:

- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$



In questo caso, però, non si ha alcuno stallo: il thread T_4 può rilasciare la propria istanza del tipo di risorsa R_2 , che si può assegnare al thread T_3 , rompendo il ciclo.

Per concludere, l'assenza di cicli nel grafo di assegnazione delle risorse implica l'*assenza* di situazioni di stallo nel sistema. Viceversa, la presenza di un ciclo non è sufficiente a implicare la presenza di uno stallo nel sistema. Questa osservazione è importante ai fini della gestione del problema delle situazioni di stallo.

8.4 Metodi per la gestione delle situazioni di stallo

Essenzialmente, il problema delle situazioni di stallo si può affrontare in tre modi:

- si può ignorare del tutto il problema, *fingendo* che le situazioni di stallo non possano mai verificarsi nel sistema;
- si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri *mai* in stallo;
- si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino.

La prima soluzione è quella adottata dalla maggior parte dei sistemi operativi, compresi Linux e Windows. La scrittura di programmi che gestiscono gli stalli diventa quindi un problema dei programmatori del kernel e dei programmatori applicativi, che usano solitamente approcci che implementano la seconda soluzione. Alcuni sistemi, come i database, adottano la terza soluzione, permettendo il verificarsi di situazioni di stallo e gestendo poi il ripristino.

Nel seguito sono spiegati brevemente tutti questi metodi. Gli algoritmi relativi sono presentati in modo dettagliato nei Paragrafi dal 8.5 al 8.8. Tuttavia, prima di procedere, vogliamo considerare anche l'opinione di quegli sviluppatori che hanno sostenuto il fatto che nessuno degli approcci di base si adatti da solo all'intero spettro di problemi di allocazione delle risorse nei sistemi operativi. Comunque tali approcci possono essere combinati, in modo da permettere la selezione del migliore per ciascuna classe di risorse del sistema.

Per assicurare che non si verifichi mai uno stallo, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. Prevenire le situazioni di stallo significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie (Paragrafo 8.3.1). Questi metodi, discussi nel Paragrafo 8.5, prevengono le situazioni di stallo controllando il modo in cui si devono fare le richieste delle risorse.

Per evitare le situazioni di stallo occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un thread richiederà e userà durante le sue attività. Con queste informazioni aggiuntive il sistema operativo può decidere se una richiesta di risorse da parte di un thread si può soddisfare o se il thread debba invece attendere. In tale thread di decisione il sistema tiene conto delle risorse correntemente disponibili, di quelle correntemente assegnate a ciascun thread e delle future richieste e futuri rilasci di ciascun thread. Questi metodi sono discussi nel Paragrafo 8.6.

Se un sistema non impiega né un algoritmo per prevenire né un algoritmo per evitare gli stalli, tali situazioni possono verificarsi. In un ambiente di questo tipo il sistema può servirsi di un algoritmo che ne esamina lo stato, al fine di stabilire se si è verificato uno stallo e in tal caso ricorrere a un secondo algoritmo per il ripristino del sistema. Tali argomenti sono discussi nei Paragrafi 8.7 e 8.8.

Se un sistema non fornisce alcun meccanismo per l'individuazione degli stalli e il ripristino del sistema, situazioni di stallo possono avvenire senza che ci sia la possibilità di capire che cos'è successo. In questo caso la presenza di situazioni di stallo non rilevate causerà un degrado delle prestazioni del sistema; infatti vi sono risorse assegnate a thread che non si possono eseguire e un numero crescente di thread richiede tali risorse ed entra in stallo, fino al blocco totale del sistema che dovrà essere riavviato manualmente.

Anche se questo metodo può non sembrare una valida soluzione al problema delle situazioni di stallo viene comunque utilizzato nella maggior parte dei sistemi operativi, come accennato in precedenza. Gli aspetti economici sono importanti: ignorare la possibilità di situazioni di stallo è più conveniente rispetto ad altri approcci. Dal momento che in molti sistemi le situazioni di stallo si verificano raramente (diciamo una volta al mese), sostenere una spesa aggiuntiva per utilizzare gli altri metodi non sembra essere così conveniente. Inoltre, i metodi utilizzati per risolvere altre situazioni possono essere utilizzati per gestire le situazioni di stallo. In alcune circostanze, un sistema si trova in uno stato di blocco (è congelato), ma non in una situazione di stallo. Si consideri, per esempio, la situazione determinata da un thread d'elaborazione in tempo reale che viene eseguito con la priorità più elevata (o qualsiasi thread in esecuzione in un sistema con scheduler senza prelazione) e che non restituisce il controllo al sistema operativo. Il sistema deve così disporre di meccanismi manuali di ripristino per queste situazioni, che può impiegare anche per le situazioni di stallo.

8.5 Prevenzione delle situazioni di stallo

Com'è evidenziato nel Paragrafo 8.3.1, affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può *prevenire* il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare. Questo metodo è analizzato trattando separatamente ciascuna delle quattro condizioni necessarie.

8.5.1 Mutua esclusione

Deve valere la condizione di mutua esclusione: almeno una risorsa deve essere non condivisibile. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file aperti per la sola lettura sono un buon esempio di risorsa condivisibile; se più thread richiedono l'apertura di un file a sola lettura, possono ottenere un accesso contemporaneo. Un thread non deve mai attendere una risorsa condivisibile. Ma poiché alcune risorse sono intrinsecamente non condivisibili, non si possono prevenire in generale le situazioni di stallo negando la condizione di mutua esclusione. Per esempio, un lock mutex non può essere contemporaneamente condiviso da diversi thread.

8.5.2 Possesso e attesa

Per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un thread che richiede una risorsa non ne possegga altre. Si può usare un protocollo che ponga la condizione che ogni thread, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano assegnate. A causa della natura dinamica della richiesta di risorse questa condizione è chiaramente poco pratica per la maggior parte delle applicazioni.

Un protocollo alternativo è quello che permette a un thread di richiedere risorse solo se non ne possiede: un thread può richiedere risorse e adoperarle, ma prima di richiedere ulteriori risorse deve rilasciare tutte quelle che possiede.

Entrambi i protocolli presentano due svantaggi principali. Innanzitutto, l'utilizzo delle risorse può risultare poco efficiente, poiché molte risorse possono essere assegnate, ma non utilizzate, per un lungo periodo di tempo. Per esempio, può essere assegnato un lock mutex a un thread per tutta la sua esecuzione, anche se ne ha bisogno soltanto per un breve periodo. Il secondo svantaggio è dovuto al fatto che si possono verificare situazioni di attesa indefinita. Un thread che richieda più risorse molto utilizzate può trovarsi nella condizione di attendere indefinitamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro thread.

8.5.3 Assenza di prelazione

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non sussista, si può impiegare il seguente protocollo. Se un thread che possiede una o più risorse ne richiede un'altra che non gli si può assegnare immediatamente (e quindi il thread deve attendere), allora si esercita la prelazione su tutte le risorse attualmente in suo possesso. Si ha cioè il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il thread sta attendendo; il thread viene nuovamente avviato solo quando può ottenere sia le vecchie risorse sia quella nuova che sta richiedendo.

In alternativa, quando un thread richiede alcune risorse, se ne verifica la disponibilità: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate a un thread che attende altre risorse. In tal caso si sottraggono le risorse desiderate a quest'ultimo thread e si assegnano al thread richiedente. Se le risorse non sono disponibili né sono possedute da un thread in attesa, il thread richiedente deve attendere. Durante l'attesa si può avere la prelazione di alcune sue risorse; ciò può accadere solo se un altro thread le richiede. Un thread si può avviare nuovamente solo quando riceve le risorse che sta richiedendo e recupera tutte quelle a esso sottratte durante l'attesa.

Questo protocollo è applicato spesso per risorse il cui stato si può salvare e recuperare facilmente in un secondo tempo, come i registri della cpu e le transazioni su un database, mentre non si può in generale applicare a risorse come i lock mutex e i semafori, ovvero a quei tipi di risorse che più comunemente generano situazioni di stallo.

8.5.4 Attesa circolare

Le tre opzioni finora presentate per la prevenzione degli stalli sono generalmente poco pratiche nella maggior parte delle situazioni. Tuttavia, la quarta e ultima condizione necessaria, l'attesa circolare, offre un'opportunità per una soluzione pratica che renda non valida una delle condizioni di stallo. Un modo per assicurare che tale condizione d'attesa non si verifichi consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun thread richieda le risorse in ordine crescente.

Si supponga che $R = \{R_1, R_2, \dots, R_m\}$ sia l'insieme dei tipi di risorse. A ogni tipo di risorsa si assegna un numero intero unico che permetta di confrontare due risorse e stabilire la relazione di precedenza nell'ordinamento. Formalmente, si definisce una funzione iniettiva, $F: R \rightarrow N$, dove N è l'insieme dei numeri naturali.

Questo schema si può implementare in un programma applicativo imponendo un ordine a tutti gli oggetti di sincronizzazione del sistema. Per esempio, l'ordinamento dei lock nel programma Pthread della Figura 8.1 potrebbe essere

```
F(first_mutex) = 1
```

```
F(second_mutex) = 5
```

Per prevenire il verificarsi di situazioni di stallo si può considerare il seguente protocollo: ogni thread può richiedere risorse solo seguendo un ordine crescente di numerazione. Ciò significa che un thread può richiedere inizialmente qualsiasi numero di istanze di

un tipo di risorsa, per esempio R_j , dopo di che il thread può richiedere istanze del tipo di risorsa R_j se e solo se $F(R_j) > F(R_i)$. Se sono necessarie più istanze dello stesso tipo di risorsa si deve presentare una singola richiesta per tutte le istanze. Per esempio, con la funzione definita precedentemente, un thread che deve impiegare contemporaneamente `first_mutex` e `second_mutex` deve prima richiedere `first_mutex` e poi `second_mutex`. In alternativa, si può stabilire che un thread, prima di richiedere un'istanza del tipo di risorsa R_j , rilasci qualsiasi risorsa R_i tale che $F(R_i) \geq F(R_j)$. Si noti inoltre che, se sono necessarie più istanze di uno stesso tipo di risorsa, queste istanze devono essere oggetto di un'unica richiesta.

Se si usa uno di questi due protocolli, la condizione di attesa circolare non può sussistere. Ciò si può dimostrare supponendo, per assurdo, che esista un'attesa circolare. Si supponga che l'insieme di thread coinvolti nell'attesa circolare sia $\{T_0, T_1, \dots, T_n\}$, dove T_1 attende una risorsa R_i , posseduta dal thread T_{i+1} . (Sugli indici si usa l'aritmetica modulare, quindi T_n attende una risorsa R_n posseduta da T_0 .) Poiché il thread T_{i+1} possiede la risorsa R_i mentre richiede la risorsa R_{i+1} , è necessario che sia verificata la condizione $F(R_i) < F(R_{i+1})$ per tutti gli i , ma ciò implica che $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. Per la proprietà transitiva, risulta che $F(R_0) < F(R_0)$, il che è impossibile; quindi, non può esservi attesa circolare.

Si tenga presente che la semplice esistenza di un ordinamento delle risorse non protegge dallo stallo: è infatti responsabilità degli sviluppatori di applicazioni scrivere programmi che rispettino tale ordinamento. Tuttavia, stabilire un ordine tra i lock può essere difficile, specialmente su un sistema con centinaia o migliaia di lock. Per affrontare questa sfida molti sviluppatori Java utilizzano il metodo `System.identityHashCode(Object)`, che restituisce il valore predefinito del codice hash del parametro `Object` che gli viene passato, come funzione per l'acquisizione ordinata dei lock.

È anche importante notare che imporre un ordinamento sui lock non garantisce l'assenza di situazioni di stallo quando i lock possono essere acquisiti dinamicamente. Per esempio, supponiamo di avere una funzione che consente di trasferire fondi tra due conti correnti. Per evitare una race condition, ogni conto ha un lock mutex associato ottenibile mediante la funzione `get_lock()`, come mostrato nella Figura 8.7.

```
void transaction(Account from, Account to, double amount)

{
    mutex lock1, lock2;

    lock1 = get_lock(from);

    lock2 = get_lock(to);

    acquire(lock1);

    acquire(lock2);

    withdraw(from, amount);

    deposit(to, amount);

    release(lock2);

    release(lock1);
}
```

Figura 8.7 Esempio di stallo con ordinamento dei lock.

Si potrebbe verificare una situazione di stallo se due thread invocassero contemporaneamente la funzione `transaction()`, passandole account distinti. In altre parole, un thread potrebbe invocare

```
transaction(checking_account, savings_account, 25.0);
```

e un altro potrebbe invocare

```
transaction(savings_account, checking_account, 50.0).
```

8.6 Evitare le situazioni di stallo

Gli algoritmi di prevenzione delle situazioni di stallo trattati nel Paragrafo 8.5 si basano sul controllo delle modalità di richiesta, così da assicurare che non si possa verificare almeno una delle condizioni necessarie perché si abbia uno stallo. Questo metodo può però causare effetti collaterali negativi, come uno scarso utilizzo dei dispositivi e una ridotta produttività del sistema.

Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. In un sistema con due risorse R_1 e R_2 , per esempio, il sistema potrebbe aver bisogno di sapere che il thread P intende richiedere prima R_1 e poi R_2 , prima di rilasciarle entrambe, mentre il thread Q richiederà prima R_2 e poi R_1 . Una volta acquisita la sequenza completa delle richieste e dei rilasci di ogni thread, il sistema può stabilire per ogni richiesta se il thread debba attendere o meno, per evitare una possibile situazione di stallo futura. In seguito a ogni richiesta, il sistema deve esaminare le risorse attualmente disponibili, le risorse attualmente assegnate a ogni thread e le richieste e i rilasci futuri per ciascun thread.

Gli algoritmi differiscono tra loro per la quantità e il tipo di informazioni richieste. Il modello più semplice e più utile richiede che ciascun thread dichiari il *numero massimo* delle risorse di ciascun tipo di cui necessita. Data questa informazione a priori, si può costruire un algoritmo capace di assicurare che il sistema non entri mai in stallo. Questo algoritmo per evitare lo stallo esamina dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare. Lo *stato* di assegnazione delle risorse è definito dal numero di risorse disponibili e assegnate e dalle richieste massime dei thread. Nei due paragrafi successivi esamineremo due algoritmi per evitare le situazioni di stallo.

LO STRUMENTO LOCKDEP DI LINUX

Assicurare che le risorse siano acquisite nell'ordine corretto è responsabilità degli sviluppatori del kernel e delle applicazioni. Tuttavia, possono essere utilizzati alcuni software per verificare che i lock vengano acquisiti in maniera ordinata. Per rilevare possibili stalli, Linux fornisce `lockdep`, uno strumento con ricche funzionalità che può essere utilizzato per verificare l'ordine dei lock nel kernel. `lockdep` è progettato per essere attivo su un kernel in esecuzione, poiché deve monitorare le sequenze di utilizzo delle acquisizioni e dei rilasci dei lock rispetto a un insieme di regole per l'acquisizione e il rilascio. Forniamo di seguito due esempi, ma si noti che `lockdep` offre molte più funzionalità rispetto a quanto qui descritto:

- L'ordine in cui vengono acquisiti i lock viene memorizzato in modo dinamico dal sistema. Se `lockdep` rileva che i lock vengono acquisiti in disordine, segnala una possibile condizione di stallo.
- In Linux, gli spinlock possono essere usati nella gestione delle interruzioni. Quando il kernel acquisisce uno spinlock che viene anche utilizzato in un gestore di interruzione si è in presenza di una possibile causa di stallo. Se l'interruzione si verifica mentre il lock è trattenuto, infatti, il gestore delle interruzioni esercita la prelazione sul codice del kernel che attualmente detiene il lock e quindi continua a ciclare tentando di acquisire il lock, causando uno stallo. La strategia generale per evitare questa situazione consiste nel disabilitare le interruzioni sul processore corrente prima di acquisire uno spinlock utilizzato anche in un gestore di interruzioni. Se `lockdep` rileva che le interruzioni sono abilitate mentre il codice del kernel acquisisce un lock utilizzato anche in un gestore di interruzioni, segnalerà una possibile situazione di stallo.

`lockdep` è stato sviluppato per essere usato come strumento per lo sviluppo o la modifica di codice del kernel e non per essere utilizzato su sistemi di produzione, poiché può rallentare in modo significativo un sistema. Il suo scopo è quello di verificare se un software come un nuovo driver di periferica o un nuovo modulo del kernel costituisce una possibile fonte di situazioni di stallo. I progettisti di `lockdep` hanno riferito che nel giro di pochi anni dal suo sviluppo, nel 2006, il numero di stalli segnalati dai report di sistema era stato ridotto di un ordine di grandezza. Sebbene `lockdep` sia stato originariamente progettato solo per l'uso nel kernel, le versioni recenti di questo strumento possono essere utilizzate per rilevare stalli nelle applicazioni utente che utilizzano i lock mutex Pthreads. Ulteriori dettagli sullo strumento `lockdep` sono disponibili all'indirizzo

<https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt>.

8.6.1 State sicuro

Uno stato si dice *sicuro* se il sistema è in grado di assegnare risorse a ciascun thread (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo. Più formalmente, un sistema si trova in stato sicuro solo se esiste una sequenza sicura. Una sequenza di thread $\langle T_1, T_2, \dots, T_n \rangle$ è una sequenza sicura per lo stato di assegnazione attuale se, per ogni T_i , le richieste che T_i può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i T_j con $j < i$. In questa situazione, se le risorse necessarie al thread T_i non sono disponibili immediatamente, allora T_i può attendere che tutti i T_j abbiano finito, e a quel punto T_i può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le risorse assegnate e terminare. Quando T_i termina, T_{i+1} può ottenere le risorse richieste, e così via. Se non esiste una sequenza di questo tipo, lo stato del sistema si dice *non sicuro*.

Uno stato sicuro non è di stallo. Viceversa, uno stato di stallo è uno stato non sicuro; tuttavia non tutti gli stati non sicuri sono stati di stallo (Figura 8.8). Uno stato non sicuro può condurre a uno stallo. Finché lo stato rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e di stallo. In uno stato non sicuro il sistema operativo non può impedire ai thread di richiedere risorse in modo da causare uno stallo: ciò che accade negli stati non sicuri dipende dal comportamento dei thread.

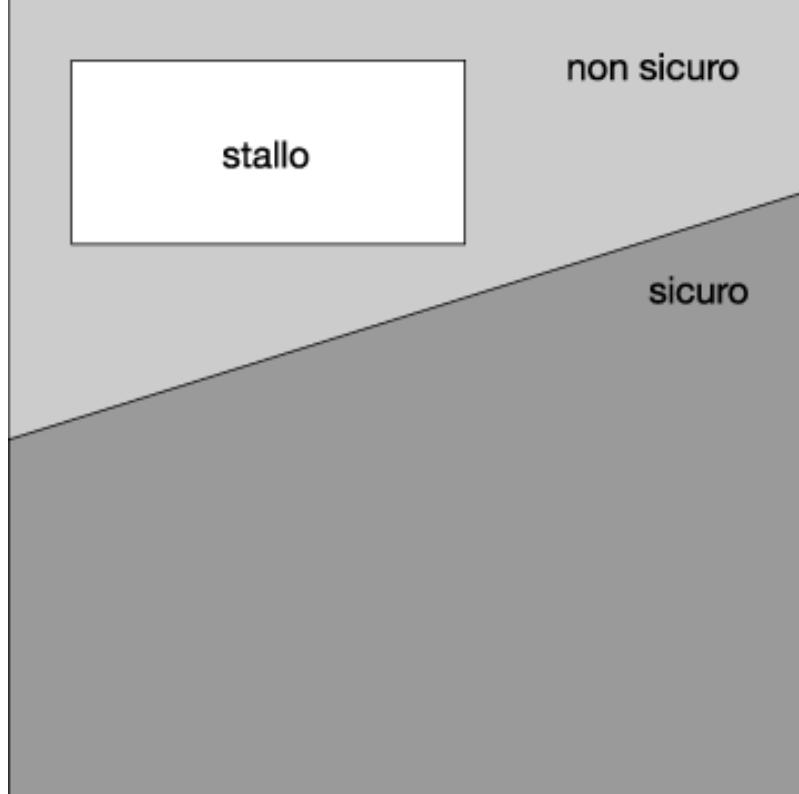


Figura 8.8 Spazi degli stati sicuri, non sicuri e di stallo.

Per illustrare meglio quel che si è detto sopra, si consideri un sistema con 12 risorse e 3 thread: T_0 , T_1 e T_2 . Il thread T_0 può richiedere 10 risorse, il thread T_1 può richiederne 4 e il thread T_2 può richiedere fino a 9. Supponendo che all'istante t_0 il thread T_0 possieda 5 risorse, e che i thread T_1 e T_2 ne possiedano 2 ciascuno, restano libere 3 risorse.

	Richieste massime	Unità
oerdue		
T_0	10	5
T_1	4	2
T_2	9	2

All'istante t_0 , il sistema si trova in uno stato sicuro. La sequenza $\langle T_1, T_0, T_2 \rangle$ soddisfa la condizione di sicurezza, poiché al thread T_1 si possono assegnare immediatamente tutte le risorse richieste, che saranno poi restituite (a quel punto saranno disponibili 5 risorse), quindi il thread T_0 può ottenere tutte le risorse richieste e restituirle (il sistema avrà 10 risorse disponibili) e infine il thread T_2 potrebbe ottenere tutte le sue risorse e restituirle (rendendo quindi disponibili tutte e 12 le risorse).

Un sistema può passare da uno stato sicuro a uno stato non sicuro. Si supponga che all'istante t_1 il thread T_2 richieda un'ulteriore risorsa e che questa gli sia assegnata: il sistema non si trova più nello stato sicuro. A questo punto, si possono assegnare tutte le risorse richieste soltanto al thread T_1 . Al momento della restituzione, il sistema avrà solo 4 risorse disponibili. Poiché al thread T_0 sono assegnate 5 risorse, ma il numero massimo è 10, il thread può richiederne altre 5; se lo fa, poiché queste non sono disponibili, il thread T_0 deve attendere. Analogamente, il thread T_2 può richiedere altre 6 risorse ed essere costretto ad attendere; il risultato è una situazione di stallo. L'errore è stato commesso nel soddisfare la richiesta di un'ulteriore risorsa fatta dal thread T_2 . Se T_2 fosse stato costretto ad attendere il termine di uno degli altri thread e il conseguente rilascio delle sue risorse, la situazione di stallo si sarebbe potuta evitare.

Dato il concetto di stato sicuro, si possono definire algoritmi che permettano di evitare le situazioni di stallo. L'idea è semplice: è sufficiente assicurare che il sistema rimanga sempre in uno stato sicuro. Il sistema si trova inizialmente in uno stato sicuro; ogni volta che un thread richiede una risorsa disponibile, il sistema deve stabilire se la risorsa può essere allocata oppure se il thread debba attendere. Si soddisfa la richiesta solo se l'assegnazione lascia il sistema in uno stato sicuro.

In questo modo, se un thread richiede una risorsa attualmente disponibile può essere comunque costretto ad attendere. Quindi, l'utilizzo delle risorse può essere inferiore rispetto a quello che si avrebbe in assenza di un algoritmo per evitare le situazioni di stallo.

8.6.2 Algoritmo con grafo di assegnazione delle risorse

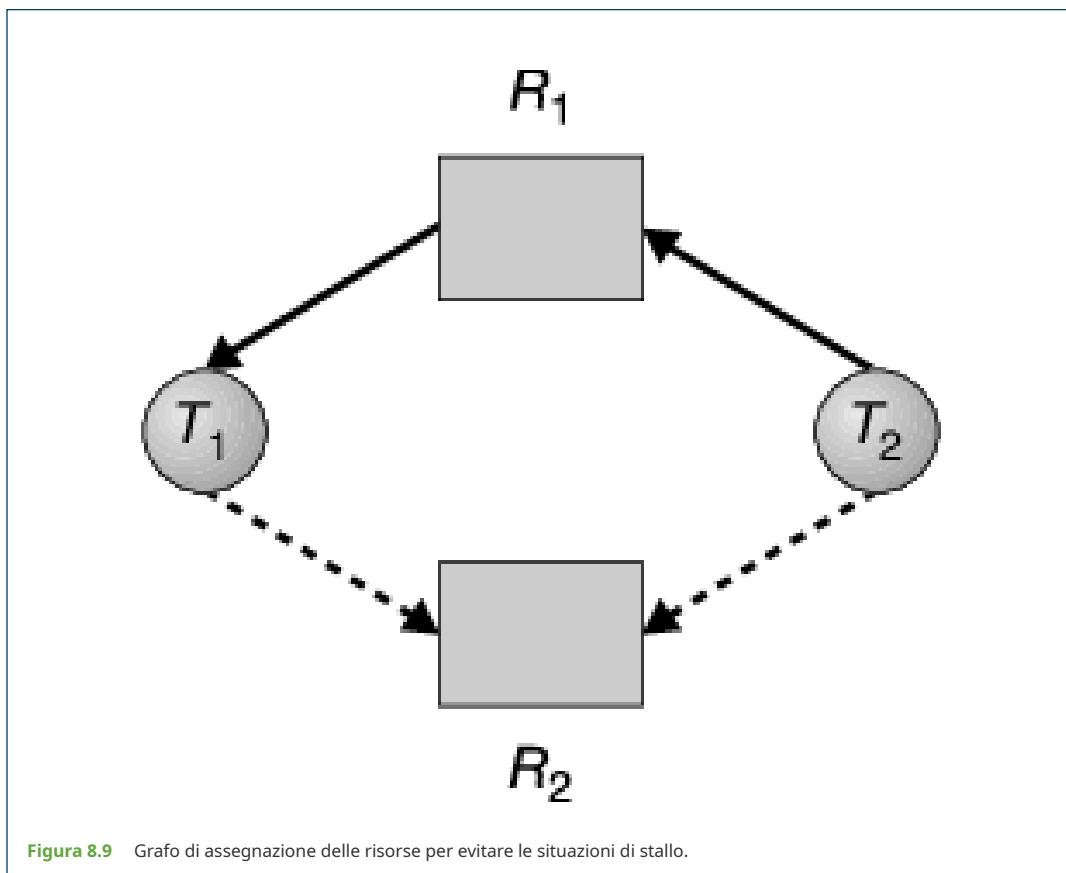
Quando il sistema per l'assegnazione delle risorse è tale che ogni tipo di risorsa ha una sola istanza, per evitare le situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse definito nel Paragrafo 8.3.2. Oltre agli archi di richiesta e di assegnazione, si introduce un nuovo tipo di arco, l'arco di "rivendicazione" (*claim edge*). Un arco di rivendicazione $T_i \rightarrow R_j$ indica che il thread T_i può richiedere la risorsa R_j in un qualsiasi momento futuro. Quest'arco ha la stessa direzione dell'arco di richiesta, ma si rappresenta con una linea tratteggiata. Quando il thread T_i richiede la risorsa R_j , l'arco di rivendicazione $T_i \rightarrow R_j$ diventa un arco di richiesta. Analogamente, quando T_i rilascia la risorsa R_j , l'arco di assegnazione $R_j \rightarrow T_i$ diventa un arco di rivendicazione $T_i \rightarrow R_j$.

Occorre sottolineare che le risorse devono essere rivendicate a priori nel sistema. Ciò significa che prima che il thread T_i inizi l'esecuzione, tutti i suoi archi di rivendicazione devono essere già inseriti nel grafo di assegnazione delle risorse. Questa condizione si può rendere meno stringente permettendo l'aggiunta di un arco di rivendicazione $T_i \rightarrow R_j$ al grafo solo se tutti gli archi associati al thread T_i sono archi di rivendicazione.

Si supponga che il thread T_i richieda la risorsa R_j . La richiesta si può soddisfare solo se la conversione dell'arco di richiesta $T_i \rightarrow R_j$ nell'arco di assegnazione $R_j \rightarrow T_i$ non causa la formazione di un ciclo nel grafo di assegnazione delle risorse. Possiamo verificare la condizione di sicurezza con un algoritmo di rilevamento dei cicli, che richiede un numero di operazioni dell'ordine di n^2 , dove n è il numero dei thread del sistema.

Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro e il thread T_i deve attendere che si soddisfino le sue richieste.

Per illustrare questo algoritmo si consideri il grafo di assegnazione delle risorse della Figura 8.9. Si supponga che T_2 richieda R_2 . Sebbene sia attualmente libera, R_2 non può essere assegnata a T_2 , poiché, com'è evidenziato nella Figura 8.10, quest'operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro. Se, a questo punto, T_1 richiedesse R_2 , si avrebbe uno stallo.



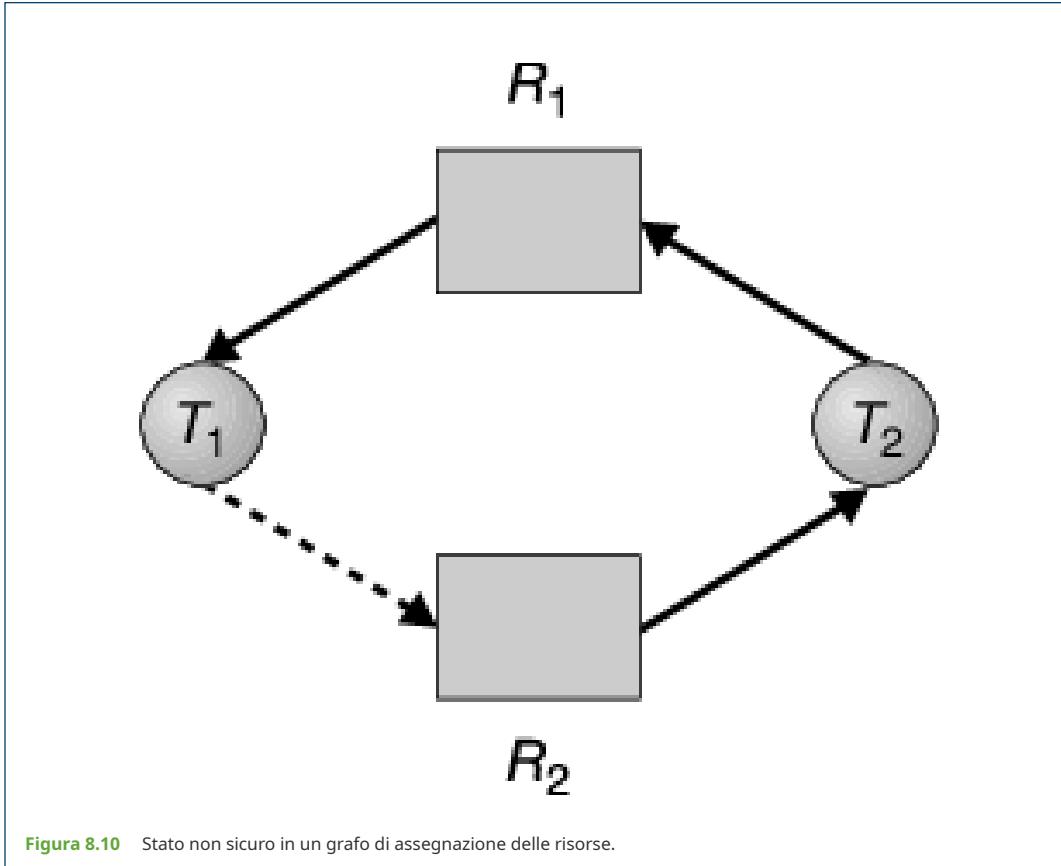


Figura 8.10 Stato non sicuro in un grafo di assegnazione delle risorse.

8.6.3 Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. L'algoritmo per evitare le situazioni di stallo descritte nel seguito, pur essendo meno efficiente dello schema con grafo di assegnazione delle risorse, si può applicare a tali sistemi, ed è noto col nome di algoritmo del banchiere. Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non assegna mai tutto il denaro disponibile, in modo da non poter più soddisfare le richieste di tutti i suoi clienti.

Quando si presenta al sistema, un nuovo thread deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui potrà aver bisogno. Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si rispetta tale condizione, si assegnano le risorse, altrimenti il thread deve attendere che qualche altro thread ne rilasci un numero sufficiente.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Sia n il numero di thread del sistema e m il numero dei tipi di risorsa. Sono necessarie le seguenti strutture dati.

- Disponibili. Un vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa; $Disponibili[j] = k$, significa che sono disponibili k istanze del tipo di risorsa R_j .
- Massimo. Una matrice $n \times m$ che definisce la richiesta massima di ciascun thread; $Massimo[i, j] = k$ significa che il thread T_i può richiedere un massimo di k istanze del tipo di risorsa R_j .
- Assegnate. Una matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni thread; $Assegnate[i, j] = k$ significa che al thread T_i sono correntemente assegnate k istanze del tipo di risorsa R_j .
- Necessità. Una matrice $n \times m$ che indica la necessità residua di risorse relativa a ogni thread; $Necessità[i, j] = k$ significa che il thread T_i , per completare il suo compito, può avere bisogno di altre k istanze del tipo di risorsa R_j . Si osservi che $Necessità[i, j] = Massimo[i, j] - Assegnate[i, j]$.

Col trascorrere del tempo, queste strutture dati variano sia nelle dimensioni sia nei valori.

Per semplificare la presentazione dell'algoritmo del banchiere, si usano le seguenti notazioni: supponendo che X e Y siano vettori di lunghezza n , si può affermare che $X \leq Y$ se e solo se $X[i] \leq Y[i]$ per ogni $i = 1, 2, \dots, n$. Per esempio, se $X = (1, 7, 3, 2)$ e $Y = (0, 3, 2, 1)$, allora $Y \leq X$; inoltre $Y < X$ se $Y \leq X$ e $Y \neq X$.

Si possono trattare le righe delle matrici $Assegnate$ e $Necessità$ come vettori chiamandole rispettivamente $Assegnate_i$ e $Necessità_i$. Il vettore $Assegnate_i$ specifica le risorse correntemente assegnate al thread T_i , mentre il vettore $Necessità_i$ specifica le risorse che il thread T_i può ancora richiedere per completare il suo compito.

8.6.3.1 Algoritmo di verifica della sicurezza

L'algoritmo utilizzato per scoprire se il sistema è o non è in uno stato sicuro si può descrivere come segue.

1. Siano *Lavoro* e *Fine* vettori di lunghezza rispettivamente m e n , si inizializza *Lavoro* = *Disponibili* e *Fine*[i] = falso, per $i = 0, 1, \dots, n - 1$;
2. si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - a) *Fine*[i] == falso
 - b) *Necessità* _{i} ≤ *Lavoro*
3. se tale i non esiste, si esegue il passo 4;
4. *Lavoro* = *Lavoro* + *Assegnate* _{i}
5. *Fine*[i] = vero
6. si va al passo 2
7. se *Fine*[i] == vero per ogni i , allora il sistema è in uno stato sicuro.

Per determinare se uno stato è sicuro tale algoritmo può richiedere un numero di operazioni dell'ordine di $m \times n^2$.

8.6.3.2 Algoritmo di richiesta delle risorse

Si descrive ora l'algoritmo che determina se le richieste possano essere soddisfatte mantenendo la condizione di sicurezza. Sia *Richieste* _{i} il vettore delle richieste per il thread T_i . Se *Richieste* _{i} [j] == k , allora il thread T_i richiede k istanze del tipo di risorsa R_j . Se il thread T_i effettua una richiesta di risorse, si svolgono le seguenti azioni:

1. se *Richieste* _{i} ≤ *Necessità* _{i} , si va al passo 2, altrimenti si riporta una condizione d'errore, poiché il thread ha superato il numero massimo di richieste;
2. se *Richieste* _{i} ≤ *Disponibili*, si esegue il passo 3, altrimenti T_i deve attendere poiché le risorse non sono disponibili;
3. si simula l'assegnazione al thread T_i delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:
 4. *Disponibili* = *Disponibili* - *Richieste* _{i}
 5. *Assegnate* _{i} = *Assegnate* _{i} + *Richieste* _{i}
 6. *Necessità* _{i} = *Necessità* _{i} - *Richieste* _{i}

Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al thread T_i si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro, T_i deve attendere *Richieste* _{i} e si ripristina il vecchio stato di assegnazione delle risorse.

8.6.3.3 Un esempio

Illustriamo l'uso dell'algoritmo del banchiere, considerando un sistema con cinque thread, da T_0 a T_4 , e tre tipi di risorse: *A*, *B*, e *C*. Il tipo di risorse *A* ha 10 istanze, il tipo *B* ha 5 istanze e il tipo *C* ha 7 istanze. Si supponga che all'istante T_0 si sia verificata la seguente situazione del sistema:

	Assegnate	Massimo	Disponibili
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
T_0	0 1 0	7 5 3	3 3 2
T_1	2 0 0	3 2 2	
T_2	3 0 2	9 0 2	
T_3	2 1 1	2 2 2	
T_4	0 0 2	4 3 3	

Il contenuto della matrice *Necessità* è definito come *Massimo* - *Assegnate*:

	Necessità
	<i>A B C</i>
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

Possiamo affermare che il sistema si trova attualmente in uno stato sicuro; infatti, la sequenza $< T_1, T_3, T_4, T_2, T_0 >$ soddisfa i criteri di sicurezza. Si supponga ora che il thread T_1 richieda un'altra istanza del tipo di risorsa *A* e due istanze del tipo *C*, quindi *Richieste*₁ = (1, 0, 2). Per stabilire se questa richiesta si possa soddisfare immediatamente verifichiamo la condizione *Richieste*₁ ≤ *Disponibili* (vale a dire (1, 0, 2) ≤ (3, 3, 2)), che risulta vera. A questo punto simuliamo che questa richiesta sia stata soddisfatta, e otteniamo il seguente nuovo stato:

	Assegnate	Necessità	Disponibili
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>

	Assegnate	Necessità	Disponibili
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 1	

Occorre stabilire se questo nuovo stato del sistema sia sicuro; a tale scopo si esegue l'algoritmo di verifica della sicurezza da cui risulta che la sequenza $\langle T_1, T_3, T_4, T_0, T_2 \rangle$ rispetta il requisito di sicurezza. Quindi si può soddisfare immediatamente la richiesta del thread T_1 .

Tuttavia, dovrebbe essere chiaro che, quando il sistema si trova in questo stato, una richiesta di $(3, 3, 0)$ da parte di T_4 non si può soddisfare perché non sono disponibili le risorse. Inoltre, una richiesta di $(0, 2, 0)$ da parte di T_0 non si può soddisfare, anche se le risorse sono disponibili, poiché lo stato risultante sarebbe non sicuro. L'implementazione dell'algoritmo del banchiere è lasciata come esercizio di programmazione.

8.7 Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo per prevenire o evitare lo stallo è possibile che una situazione di stallo si verifichi. In un ambiente di questo genere, il sistema può fornire i seguenti algoritmi:

- un algoritmo che esamina lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo.

Nell'analisi seguente sono trattati i suddetti argomenti sia per sistemi con una sola istanza di ciascun tipo di risorsa, sia per sistemi con più istanze. Tuttavia, a questo punto, occorre notare che uno schema di rilevamento e ripristino richiede un overhead che include non solo i costi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, ma anche i potenziali costi dovuti alle perdite di informazioni connesse al ripristino da una situazione di stallo.

8.7.1 Istanza singola di ciascun tipo di risorsa

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento di situazioni di stallo che fa uso di una variante del grafo di assegnazione delle risorse, detta grafo d'attesa, ottenuta dal grafo di assegnazione delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra i thread.

Più precisamente, un arco da T_i a T_j del grafo d'attesa implica che il thread T_i attende che il thread T_j rilasci una risorsa di cui T_i ha bisogno. Un arco $T_i \rightarrow T_j$ esiste nel grafo d'attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi $T_i \rightarrow R_q$ e $R_q \rightarrow T_j$ per qualche risorsa R_q . Nella Figura 8.11 sono illustrati un grafo di assegnazione delle risorse e il corrispondente grafo d'attesa.

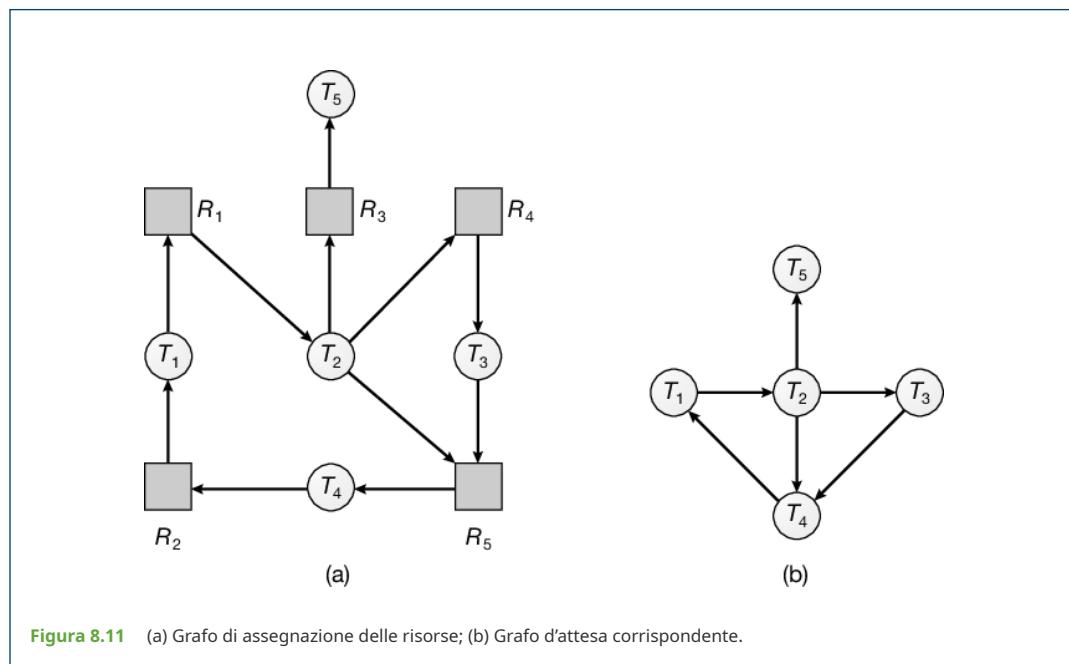


Figura 8.11 (a) Grafo di assegnazione delle risorse; (b) Grafo d'attesa corrispondente.

Come in precedenza, nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve *mantenere aggiornato* il grafo d'attesa e *invocare periodicamente un algoritmo* che cerchi un ciclo all'interno del grafo. L'algoritmo per il rilevamento di un ciclo all'interno di un grafo richiede un numero di operazioni dell'ordine di n^2 , dove con n si indica il numero dei vertici del grafo.

Il toolkit bcc descritto nel Paragrafo 2.10.4 fornisce uno strumento in grado di rilevare potenziali stalli con lock mutex Pthread in un processo utente in esecuzione su un sistema Linux. Lo strumento `deadlock_detector` di bcc funziona inserendo delle *probe* che tracciano le chiamate alle funzioni `pthread_mutex_lock()` e `pthread_mutex_unlock()`. Quando il processo specificato effettua una chiamata a una delle due funzioni, `deadlock_detector` costruisce un grafo d'attesa dei lock mutex in quel processo e segnala il possibile verificarsi di stalli se rileva un ciclo nel grafo.

Sebbene Java non fornisca un esplicito supporto per il rilevamento degli stalli, è possibile utilizzare un dump dei thread per analizzare un programma in esecuzione e determinare se si è verificato uno stallo. Un dump dei thread è un utile strumento di debug in grado di mostrare un'istantanea degli stati di tutti i thread in un'applicazione Java. I dump dei thread Java mostrano anche informazioni sui lock, inclusi i lock che un thread bloccato è in attesa di acquisire. Quando viene generato un dump dei thread, la JVM cerca il grafo d'attesa per individuare i cicli, segnalando eventuali stalli rilevati. Per generare un dump dei thread di un'applicazione in esecuzione occorre digitare dalla riga di comando:

```
Ctrl-L (unix, Linux o macos)
Ctrl-Break (Windows)
```

Nella sezione di download del codice sorgente relativa a questo testo forniamo un esempio Java del programma mostrato nella Figura 8.1 e descriviamo come generare un dump dei thread che riporta i thread Java in situazione di stallo.

8.7.2 Più istanze di ciascun tipo di risorsa

Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. Il seguente algoritmo di rilevamento di situazioni di stallo è, invece, applicabile a tali sistemi. Esso si serve di strutture dati variabili nel tempo, simili a quelle adoperate nell'algoritmo del banchiere (Paragrafo 8.6.3).

- Disponibili. Vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- Assegnate. Matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorse correntemente assegnate a ciascun thread.
- Richieste. Matrice $n \times m$ che indica la richiesta attuale di ciascun thread. Se $\text{Richieste}[i, j] = k$, significa che il thread T_i sta richiedendo altre k istanze del tipo di risorsa R_j .

La relazione \leq tra due vettori si definisce come nel Paragrafo 8.6.3. Per semplificare la notazione, le righe delle matrici *Assegnate* e *Richieste* si trattano come vettori e, nel seguito, sono indicate rispettivamente come *Assegnate_i* e *Richieste_i*. L'algoritmo di rilevamento descritto verifica ogni possibile sequenza di assegnazione per i thread che devono ancora essere completati. Questo algoritmo si può confrontare con quello del banchiere del Paragrafo 8.6.3.

1. Siano *Lavoro* e *Fine* vettori di lunghezza rispettivamente m e n , si inizializza *Lavoro* = *Disponibili*; per $i = 0, 1, \dots, n$, se *Assegnate_i* $\neq 0$, allora *Fine_i* = *falso*, altrimenti *Fine_i* = *vero*;
2. si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - Fine_i* = *falso*
 - Richieste_i* \leq *Lavoro*
3. se tale i non esiste, si esegue il passo 4;
4. *Lavoro* = *Lavoro* + *Assegnate_i*
5. *Fine_i* = *vero*
6. si torna al passo 2
7. se *Fine_i* = *falso* per qualche i , $0 \leq i < n$, allora il sistema è in stallo, inoltre, se *Fine_i* = *falso*, il thread T_i è in stallo.

Tale algoritmo richiede un numero di operazioni dell'ordine di $m \times n^2$ per controllare se il sistema è in stallo.

Ci si può chiedere perché le risorse del thread T_i siano liberate (passo 3) non appena risulta valida la condizione $\text{Richieste}_i \leq \text{Lavoro}$ (passo 2.b). Sappiamo che T_i non è correntemente coinvolto in uno stallo, quindi, assumendo un atteggiamento ottimistico, si suppone che T_i non intenda richiedere altre risorse per completare il proprio compito, e che restituisca presto tutte le risorse. Se non si verifica l'ipotesi fatta, si potrà verificare uno stallo, che sarà rilevato quando si richiamerà nuovamente l'algoritmo di rilevamento.

Per illustrare questo algoritmo, si consideri un sistema con cinque thread, da T_0 a T_4 , e tre tipi di risorse: *A*, *B*, e *C*. Il tipo di risorsa *A* ha 7 istanze, il tipo *B* ha 2 istanze e il tipo *C* ne ha 6. Si supponga di avere, all'istante T_0 , il seguente stato di assegnazione delle risorse:

	Assegnate	Richieste	Disponibili
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	

	Assegnate	Richieste	Disponibili
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

Il sistema non è in stallo. Infatti eseguendo l'algoritmo troviamo che la sequenza $\langle T_0, T_2, T_3, T_1, T_4 \rangle$ porta a $Fine[i] == \text{vero}$ per ogni i .

Si supponga ora che il thread T_2 richieda un'altra istanza di tipo C. La matrice *Richieste* viene modificata come segue:

	Richieste
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

Ora il sistema è in stallo. Anche se si possono liberare le risorse possedute dal thread T_0 , il numero delle risorse disponibili non è sufficiente per soddisfare le richieste degli altri thread, quindi si verifica uno stallo composto dai thread T_1, T_2, T_3 e T_4 .

8.7.3 Uso dell'algoritmo di rilevamento

Per sapere quando è necessario ricorrere all'algoritmo di rilevamento si devono considerare i seguenti fattori:

1. *frequenza* presunta con la quale si verifica uno stallo;
2. *numero* dei thread che sarebbero influenzati da tale stallo.

Se le situazioni di stallo sono frequenti, è necessario ricorrere spesso all'algoritmo per il loro rilevamento. Le risorse assegnate a thread in stallo rimangono inattive fino all'eliminazione dello stallo. Inoltre, il numero dei thread coinvolti nel ciclo di stallo può aumentare.

Le situazioni di stallo si verificano solo quando qualche thread fa una richiesta che non si può soddisfare immediatamente; questa può essere la richiesta che chiude una catena di thread in attesa. All'estremo, si può invocare l'algoritmo di rilevamento ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. In questo caso non si identifica soltanto il gruppo di thread in stallo, ma anche lo specifico thread che ha "causato" lo stallo, anche se, in verità, ciascuno dei thread in stallo è un elemento del ciclo all'interno del grafo di assegnazione delle risorse, quindi tutti i thread sono, congiuntamente, responsabili dello stallo. Se esistono tipi di risorsa diversi, una singola richiesta può causare più cicli nel grafo delle risorse, ciascuno dei quali viene completato da quest'ultima richiesta, causata da un thread identificabile.

Naturalmente, l'uso dell'algoritmo di rilevamento per ogni richiesta aumenta notevolmente il carico computazionale. Un'alternativa meno dispendiosa è quella in cui l'algoritmo di rilevamento s'invoca a intervalli definiti, per esempio una volta ogni ora, oppure ogni volta che l'utilizzo della cpu scende sotto il 40 per cento, poiché uno stallo rende inefficienti le prestazioni del sistema e quindi porta a una drastica riduzione dell'utilizzo della cpu. Se l'algoritmo di rilevamento viene invocato in momenti arbitrari, poiché nel grafo delle risorse possono coesistere molti cicli, normalmente non si può dire quale fra i tanti thread in stallo abbia "causato" lo stallo.

GESTIONE DEGLI STALLI NEI DATABASE

I sistemi di gestione dei database (dbms) forniscono un'utile esempio di come sia il software open source sia quello commerciale gestiscano gli stalli. Gli aggiornamenti a un database possono essere eseguiti come **transazioni** e, per garantire l'integrità dei dati, vengono generalmente utilizzati i lock. Una transazione può coinvolgere diversi lock, non sorprende quindi che siano possibili stalli in un database che esegue più transazioni concorrenti. Per gestire le situazioni di stallo la maggior parte dei dbms include un meccanismo di rilevamento degli stalli e di ripristino. Il database server cercherà periodicamente i cicli nel grafo d'attesa per rilevare situazioni di stallo in un insieme di transazioni. Quando viene rilevato uno stallo, viene designata una transazione vittima che viene interrotta e riportata allo stato iniziale ("rolled back"), permettendo così il rilascio dei lock trattenuti dalla transazione vittima e liberando le restanti transazioni dalla situazione di stallo. Una volta riprese le transazioni rimanenti, la transazione annullata viene riavviata.

La scelta di una transazione vittima dipende dal dbms; per esempio, MySQL tenta di selezionare le transazioni che riducono al minimo il numero di righe inserite, aggiornate o eliminate.

Bibliografia

- [Kim et al. 2009] J. Kim, Y. Oh, E. Kim, J. C. D. Lee e S. Noh, “Disk Schedulers for Solid State Drivers”, *Proceedings of the seventh ACM international conference on Embedded software*, p. 295–304, 2009.
- [Love 2010] R. Love, *Linux Kernel Development*, 3^a Ed., Developer’s Library, 2010.
- [McDougall e Mauro 2007] R. McDougall e J. Mauro, *Solaris Internals*, 2^a Ed., Prentice Hall, 2007.
- [Mesnier et al. 2003] M. Mesnier, G. Ganger e E. Ridel, “Object-based storage”, *IEEE Communications Magazine*, Vol. 41, Num. 8, p. 84–99, 2003.
- [Patterson et al. 1988] D. A. Patterson, G. Gibson e R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, p. 109–116, 1988.
- [Russinovich et al. 2017] M. Russinovich, D.A. Solomon e A. Ionescu, *Windows Internals—Part 1*, 7^a Ed., Microsoft Press, 2017.
- [Services 2012] E. E. Services, *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*, Wiley, 2012.

8.8 Ripristino da situazioni di stallo

Una volta rilevato uno stallo, questo si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza dello stallo, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico dalla situazione di stallo. Uno stallo si può eliminare in due modi: il primo prevede semplicemente la terminazione di uno o più thread per interrompere l'attesa circolare; il secondo consiste nell'esercitare la prelazione su alcune risorse in possesso di uno o più thread in stallo.

8.8.1 Terminazione di processi e thread

Per eliminare le situazioni di stallo attraverso la terminazione di processi o thread si possono adoperare due metodi; in entrambi il sistema recupera tutte le risorse assegnate ai processi terminati.

- Terminazione di tutti i processi in stallo. Chiaramente questo metodo interrompe il ciclo di stallo, ma l'operazione è molto onerosa; questi processi possono aver già fatto molti calcoli i cui risultati si scartano e probabilmente dovranno essere ricalcolati.
- Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo. Questo metodo comporta un notevole overhead, poiché, dopo aver terminato ogni processo, si deve invocare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo.

Procurare la terminazione di un processo può non essere un'operazione semplice: se il processo si trova nel mezzo dell'aggiornamento di un file, la terminazione lascia il file in uno stato scorretto; analogamente, se il processo si trova nel mezzo dell'aggiornamento di dati condivisi mentre è in possesso di un lock mutex, il sistema deve reimpostare lo stato del lock come disponibile, sebbene non sia possibile offrire alcuna garanzia sull'integrità dei dati condivisi.

Se si adopera il metodo di terminazione parziale, occorre determinare quale processo, o quali processi in situazione di stallo devono essere terminati. Analogamente ai problemi di scheduling della cpu, si tratta di seguire un criterio. Le considerazioni sono essenzialmente economiche: si dovrebbero arrestare i processi la cui terminazione causa il minimo costo. Sfortunatamente, il termine *minimo costo* non è preciso. La scelta dei processi è influenzata da diversi fattori, tra cui i seguenti:

1. la priorità del processo;
2. il tempo trascorso in computazione e il tempo ancora necessario per completare il compito assegnato al processo;
3. la quantità e il tipo di risorse impiegate dal processo (per esempio, se si può effettuare facilmente la prelazione delle risorse);
4. la quantità di ulteriori risorse di cui il processo ha ancora bisogno per completare l'esecuzione;
5. il numero di processi che si devono terminare.

8.8.2 Prelazione delle risorse

Per eliminare uno stallo si può esercitare la prelazione delle risorse: le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l'interruzione del ciclo di stallo.

Se per gestire le situazioni di stallo s'impiega la prelazione, si devono considerare i seguenti problemi.

1. Selezione di una vittima. Occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione. Come per la terminazione dei processi, è necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero delle risorse possedute da un processo in stallo, e la quantità di tempo già spesa durante l'esecuzione del processo.
2. Ristabilimento di un precedente stato sicuro. Occorre stabilire che cosa fare con un processo cui è stata sottratta una risorsa. Poiché è stato privato di una risorsa necessaria, la sua esecuzione non può continuare normalmente, quindi deve essere ricondotto a un precedente stato sicuro (*rollback*) dal quale essere riavviato. Poiché, in generale, è difficile stabilire quale stato sia sicuro, la soluzione più semplice consiste nel terminare il processo e quindi riavivarlo. Benché sia più efficace effettuare il rollback solo fino al punto necessario allo scioglimento della situazione di stallo, questo metodo richiede che il sistema mantenga più informazioni sullo stato di tutti i processi in esecuzione.
3. Attesa indefinita (starvation). È necessario assicurare che non si verifichino situazioni d'attesa indefinita, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo.
4. In un sistema in cui la scelta della vittima avviene soprattutto secondo fattori di costo, può accadere che si scelga sempre lo stesso processo; in questo caso il processo non riesce mai a completare il suo compito; si tratta di una situazione d'attesa indefinita che ogni sistema reale deve saper affrontare. Chiaramente è necessario assicurare che un processo possa essere prescelto come vittima solo un numero finito (e ridotto) di volte; la soluzione più diffusa prevede l'inclusione del numero di rollback tra i fattori di costo.

8.9 Sommario

- Uno stallo (*deadlock*) si verifica quando, in un insieme di processi, ciascun processo attende un evento che può essere causato solo da un altro processo dell'insieme.
- Vi sono quattro condizioni necessarie per lo stallo: (1) mutua esclusione, (2) possesso e attesa, (3) assenza di prelazione e (4) attesa circolare. Una situazione di stallo è possibile solo quando sono verificate tutte e quattro le condizioni.
- Gli stalli possono essere modellati mediante grafi di allocazione delle risorse in cui un ciclo indica uno stallo.
- È possibile prevenire gli stalli garantendo che non si verifichi una delle quattro condizioni necessarie per lo stallo. Tra le quattro possibilità, eliminare l'attesa circolare è l'unico approccio pratico.
- Lo stallo può essere evitato usando l'algoritmo del banchiere, che non concede risorse se così facendo si indurrebbe il sistema in uno stato non sicuro in cui si potrebbe verificare uno stallo.
- Un algoritmo di rilevamento degli stalli può esaminare i processi e le risorse su un sistema in esecuzione per determinare se un insieme di processi si trova in una situazione di stallo.
- Un sistema può tentare di recuperare da una situazione di stallo interrompendo uno dei processi nell'attesa circolare o esercitando la prelazione sulle risorse che sono state assegnate a un processo in stallo.

Esercizi di ripasso

8.1 Elencate tre esempi di stallo che non riguardino l'informatica.

8.2 Supponete che un sistema sia in uno stato non sicuro. Mostrate che è possibile che i thread completino l'esecuzione senza entrare in una situazione di stallo.

8.3 Considerate la seguente situazione istantanea di un sistema:

	Assegnate	Massimo	Disponibili
	<i>A B C D</i>	<i>A B C D</i>	<i>A B C D</i>
T_0	0 0 1 2	0 0 1 2	1 5 2 0
T_1	1 0 0 0	1 7 5 0	
T_2	1 3 5 4	2 3 5 6	
T_3	0 6 3 2	0 6 5 2	
T_4	0 0 1 4	0 6 5 6	

Rispondete alle seguenti domande servendovi dell'algoritmo del banchiere.

- Qual è il contenuto della matrice *Necessità*?
- Il sistema è in uno stato sicuro?
- Se arriva una richiesta dal thread T_1 di (0,4,2,0) tale richiesta può venire accolta immediatamente?

8.4 Un metodo possibile per prevenire gli stalli consiste nel disporre di una singola risorsa di ordine più elevato che deve essere richiesta prima di ogni altra risorsa. Lo stallo è possibile, per esempio, se più thread tentano di accedere agli oggetti di sincronizzazione $A \dots E$ (tali oggetti di sincronizzazione possono includere mutex, semafori, variabili condition, e simili). Possiamo prevenire gli stalli aggiungendo un sesto oggetto F . Ogni volta che un thread vuole acquisire un lock di sincronizzazione per ciascun oggetto $A \dots E$, deve prima acquisire il lock per l'oggetto F . Questa soluzione è conosciuta come inclusione (*containment*): i lock per gli oggetti $A \dots E$ sono inclusi all'interno del lock per l'oggetto F . Paragonate questo schema con quello ad attesa circolare del Paragrafo 8.5.4.

8.5 Dimostrate che l'algoritmo di sicurezza presentato nel Paragrafo 8.6.3 richiede un numero di operazioni dell'ordine di $m \times n^2$.

8.6 Considerate un sistema informatico che esegua 5000 task al mese e non disponga né di uno schema per prevenire gli stalli né di uno schema per evitarli. Gli stalli si verificano circa due volte al mese e l'operatore deve terminare e rieseguire circa 10 task per ogni stallo. Ogni task costa circa 2 dollari (in tempo del processore), e i task terminati tendono a essere circa a metà del loro lavoro quando vengono interrotti.

Un programmatore di sistema ha stimato che un algoritmo per evitare gli stalli (come l'algoritmo del banchiere) potrebbe essere installato nel sistema con un incremento del 10% circa del tempo medio di esecuzione per task. Siccome la macchina ha attualmente il 30% di periodo d'inattività, tutti i 5000 task al mese potrebbero ancora essere eseguiti, anche se il tempo di completamento aumenterebbe in media del 20 per cento circa.

- Quali sono i vantaggi dell'installazione dell'algoritmo per evitare gli stalli?
- Quali sono gli svantaggi dell'installazione dell'algoritmo per evitare gli stalli?

8.7 Un sistema può individuare che alcuni dei suoi thread sono in una situazione di attesa indefinita? Se la vostra risposta è affermativa, spiegate com'è possibile. Se è negativa, spiegate come il sistema può trattare il problema dell'attesa indefinita.

8.8 Considerate la seguente politica di allocazione delle risorse. Le richieste e i rilasci di risorse sono sempre possibili. Se una richiesta di risorse non può essere soddisfatta perché queste non sono disponibili, si controllano tutti i processi bloccati in attesa di risorse. Se un thread bloccato possiede le risorse desiderate, queste vengono prelevate e assegnate al thread che le richiede. Il vettore delle risorse attese dal thread bloccato è aggiornato in modo da includere le risorse sottratte al thread.

Si consideri per esempio un sistema con tre tipi di risorse e il vettore *Disponibili* inizializzato a (4,2,2). Se il thread T_0 richiede (2,2,1), le ottiene. Se T_1 richiede (1,0,1), le ottiene. Poi, se T_0 chiede (0,0,1) viene bloccato (risorsa non disponibile). Se T_2 ora chiede

(2,0,0) ottiene la risorsa disponibile (1,0,0) e quella che è stata assegnata a T_0 (poiché T_0 è bloccato). Il vettore *Assegnate* di T_0 scende a (1,2,1) e il suo vettore *Necessità* passa a (1,0,1).

a. Possono verificarsi stalli? Se la risposta è affermativa, fornite un esempio. In caso di risposta negativa, specificate quale condizione necessaria non si può verificare.

b. Può verificarsi un blocco indefinito? Spiegate la risposta.

8.9 Si consideri la seguente situazione istantanea di un sistema:

	Allocazione	Massimo
	A B C D	A B C D
T_0	3 0 1 4	5 1 1 7
T_1	2 2 1 0	3 2 1 1
T_2	3 1 2 1	3 3 2 1
T_3	0 5 1 0	4 6 1 2
T_4	4 2 1 2	6 3 2 5

Utilizzando l'algoritmo del banchiere determinate se ciascuno dei seguenti stati è non sicuro. Se lo stato è sicuro, illustrate l'ordine in cui i thread possono essere completati. In caso contrario, spiegate perché lo stato è non sicuro.

a. $Disponibile = (0, 3, 0, 1)$

b. $Disponibile = (1, 0, 0, 2)$

8.10 Supponete di aver codificato l'algoritmo di sicurezza per evitare gli stalli e che ora vi sia richiesto di implementare l'algoritmo di rilevamento delle situazioni di stallo. È possibile farlo utilizzando semplicemente il codice dell'algoritmo di sicurezza e ridefinendo $Massimo_i = Attesa_i + Assegnate_i$, dove $Attesa_i$ è un vettore che specifica le risorse attese dal thread i e $Assegnate_i$ è definito come nel Paragrafo 8.6? Motivate la risposta.

8.11 È possibile che uno stallo coinvolga un solo processo con un singolo thread? Argomentate la risposta.

Esercizi

8.12 Considerate lo stallo di traffico automobilistico illustrato nella Figura 8.12:

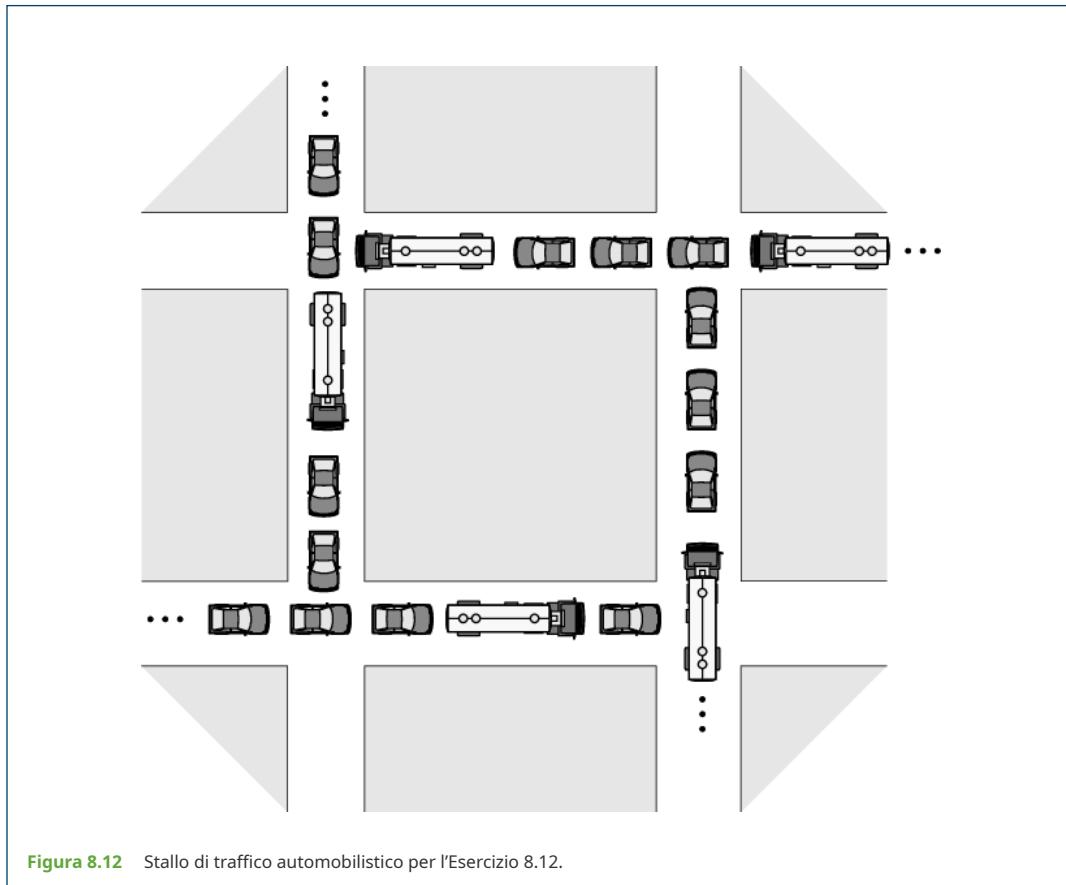


Figura 8.12 Stallo di traffico automobilistico per l'Esercizio 8.12.

- dimostrate che le quattro condizioni necessarie per lo stallo valgono anche in questo esempio;
- fissate una semplice regola che eviti le situazioni di stallo in questo sistema.

8.13 Tracciate il grafo di allocazione delle risorse che illustri lo stallo dall'esempio di programma mostrato nella Figura 8.1 (Paragrafo 8.2).

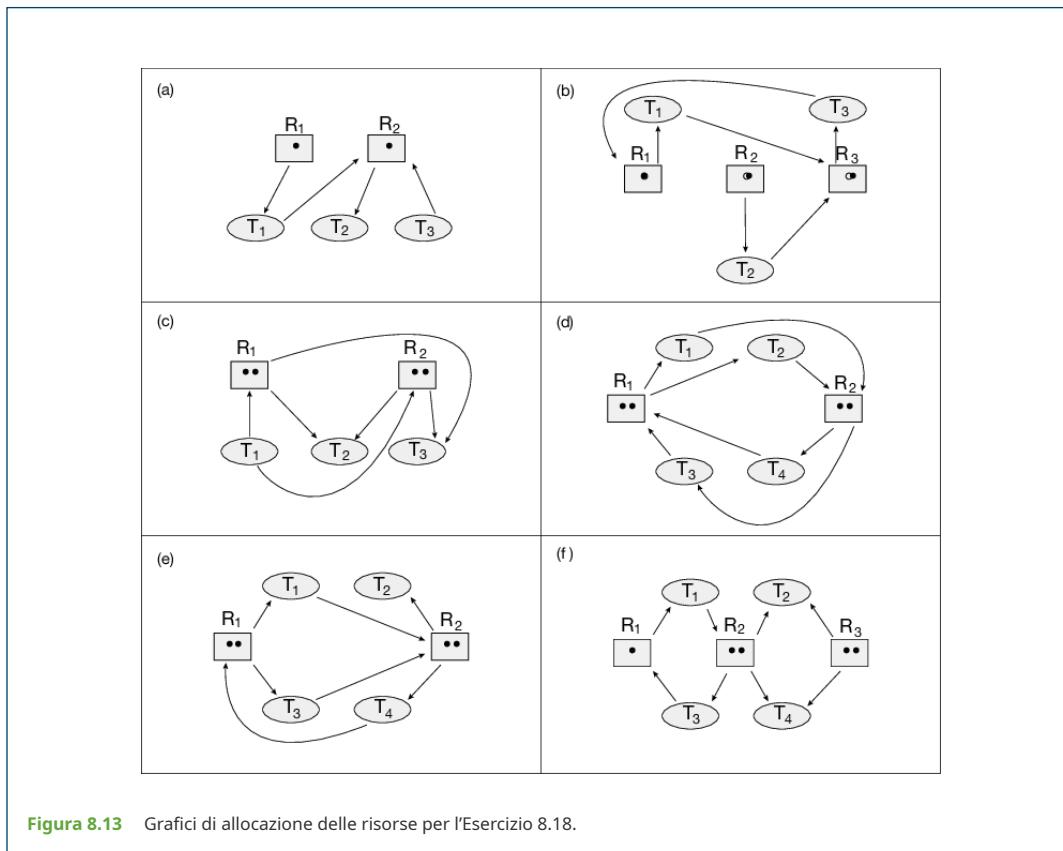
8.14 Nel Paragrafo 6.8.1 viene descritto un potenziale scenario di stallo che coinvolge i processi P_0 e P_1 e i semafori s e q . Tracciate il grafo di allocazione delle risorse che illustri lo stallo nello scenario presentato in quel paragrafo.

8.15 Supponiamo che un'applicazione multithread utilizzi per la sincronizzazione soltanto lock lettore-scrittore. Applicando le quattro condizioni necessarie per le situazioni di stallo, può ancora verificarsi uno stallo se si utilizzano più lock lettore-scrittore?

8.16 Il programma di esempio mostrato nella Figura 8.1 non conduce sempre a uno stallo. Descrivete il ruolo dello scheduler della cpu e come lo scheduler possa contribuire a uno stallo in questo programma.

8.17 Nel Paragrafo 8.5.4 abbiamo descritto una situazione in cui preveniamo gli stalli facendo in modo che tutti i lock vengano acquisiti in un certo ordine. Tuttavia abbiamo anche notato che gli stalli sono possibili in questa situazione se due thread invocano simultaneamente la funzione `transaction()`. Correggete la funzione `transaction()` in modo da prevenire gli stalli.

8.18 Quale dei sei grafici di allocazione delle risorse illustrati nella Figura 8.13 mostra una situazione di stallo? Applicate il ciclo di thread e risorse a quelle situazioni che sono in stallo. Nel caso non ci sia alcuno stallo illustrate l'ordine in cui i thread possono completare l'esecuzione.



8.19 Confrontate il modello dell'attesa circolare con le varie strategie di neutralizzazione dello stallo (quali l'algoritmo del banchiere) rispetto alle seguenti problematiche:

- a. overhead di esecuzione;
- b. produttività del sistema.

8.20 In un sistema reale, né le risorse disponibili, né le richieste di risorse da parte dei processi sono stabili in lunghi periodi (mesi). Le risorse si guastano o sono sostituite, nuovi processi vanno e vengono, si acquistano e si aggiungono nuove risorse al sistema. Se le situazioni di stallo si controllano con l'algoritmo del banchiere, dite quali tra le seguenti modifiche si possono apportare con sicurezza, ossia senza introdurre la possibilità di situazioni di stallo, e in quali circostanze:

- a. aumento di *Disponibili* (aggiunta di nuove risorse);
- b. riduzione di *Disponibili* (risorsa rimossa definitivamente dal sistema);
- c. aumento di *Massimo* per un processo (il processo necessita di più risorse di quante siano permesse);
- d. riduzione di *Massimo* per un processo (il processo decide che non ha bisogno di tante risorse);
- e. aumento del numero di processi;
- f. riduzione del numero di processi.

8.21 Considerate la seguente situazione istantanea di un sistema:

	Allocazione	Massimo
	A B C D	A B C D
T_0	2 1 0 6	6 3 2 7
T_1	3 3 1 3	5 4 1 5

	Allocazione	Massimo
T_2	2 3 1 2	6 6 1 4
T_3	1 2 3 4	4 3 4 5
T_4	3 0 3 0	7 2 6 1

Quali sono i contenuti della matrice *Necessità*?

8.22 Considerate un sistema composto da quattro risorse dello stesso tipo condivise da tre processi, ciascuno dei quali necessita di non più di due risorse. Dimostrate che non si possano verificare situazioni di stallo.

8.23 Considerate un sistema composto da m risorse dello stesso tipo, condivise da n processi. Le risorse possono essere richieste e rilasciate dai processi solo una alla volta. Dimostrate che non si possono verificare situazioni di stallo se si rispettano le seguenti condizioni:

- a. la richiesta massima di ciascun processo è compresa tra 1 e m risorse;
- b. la somma di tutte le richieste massime è minore di $m + n$.

8.24 Considerate il Problema dei cinque filosofi, supponendo che le bacchette siano disposte al centro del tavolo e che un filosofo ne possa utilizzare due qualsiasi. Supponete che le richieste di bacchette avvengano una per volta. Descrivete una semplice regola che determini se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.

8.25 Considerate lo stesso contesto del problema precedente. Ipotizzate, ora, che ogni filosofo richieda tre bacchette per mangiare e che, anche qui, le richieste siano avanzate separatamente. Descrivete delle semplici regole che determinino se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.

8.26 Potete ricavare un algoritmo del banchiere che sia adatto a un solo tipo di risorsa dall'algoritmo generale del banchiere, semplicemente riducendo la dimensione dei vari array a 1. Dimostrate con un esempio che l'algoritmo del banchiere generale non può essere implementato applicando separatamente a ciascun tipo di risorsa l'algoritmo per un solo tipo di risorsa.

8.27 Considerate la seguente situazione istantanea di un sistema:

	Allocazione	Massimo
	$A \ B \ C \ D$	$A \ B \ C \ D$
T_0	1 2 0 2	4 3 1 6
T_1	0 1 1 2	2 4 2 4
T_2	1 2 4 0	3 6 5 1
T_3	1 2 0 1	2 6 2 3
T_4	1 0 0 1	3 1 1 2

Servendovi dell'algoritmo del banchiere valutate se ciascuno dei seguenti stati sia sicuro o no. Se lo stato è sicuro illustrate l'ordine in cui i processi possano essere completati. Altrimenti spiegate perché lo stato non è sicuro.

- a. Disponibile = (2, 2, 2, 3)
- b. Disponibile = (4, 4, 1, 1)
- c. Disponibile = (3, 0, 1, 4)
- d. Disponibile = (1, 5, 2, 2)

8.28 Considerate la seguente situazione istantanea di un sistema:

	Allocazione	Massimo	Disponibili
	$A \ B \ C \ D$	$A \ B \ C \ D$	$A \ B \ C \ D$
T_0	3 1 4 1	6 4 7 3	2 2 2 4
T_1	2 1 0 2	4 2 3 2	
T_2	2 4 1 3	2 5 3 3	

Allocazione	Massimo	Disponibili
T_3 4 1 1 0	6 3 3 2	
T_4 2 2 2 1	5 6 7 5	

Rispondete alle seguenti domande usando l'algoritmo del banchiere:

- a. Mostrate che il sistema è in uno stato sicuro indicando un ordine in cui i processi possono essere completati.
- b. Se arriva una richiesta di T_4 per (2, 2, 2, 4), la richiesta può essere soddisfatta immediatamente?
- c. Se arriva una richiesta di T_2 per (0, 1, 1, 0), la richiesta può essere soddisfatta immediatamente?
- d. Se arriva una richiesta di T_3 per (2, 2, 1, 2), la richiesta può essere soddisfatta immediatamente?

8.29 Quale ipotesi ottimistica è alla base dell'algoritmo di rilevamento delle situazioni di stallo? Come potrebbe essere violata questa ipotesi?

8.30 Un ponte a corsia unica collega i due villaggi di North Tunbridge e South Tunbridge, nel Vermont. Gli agricoltori dei due villaggi usano tale ponte per portare i loro prodotti nella cittadina confinante. Il ponte rimane bloccato se un contadino diretto a nord e uno diretto a sud vi salgono contemporaneamente. (I contadini del Vermont sono ostinati: non farebbero mai marcia indietro.) Scrivete un algoritmo in pseudocodice che eviti lo stallo tramite i semafori e/o i lock mutex. Non curatevi, all'inizio, dell'attesa indefinita, ossia la situazione in cui i contadini diretti a nord impediscono a quelli diretti a sud di salire sul ponte, o viceversa.

8.31 Modificate la soluzione al problema dell'Esercizio 8.30 in modo da eliminare l'attesa indefinita.

8.32 Implementate la soluzione dell'Esercizio 8.30 usando la sincronizzazione posix. In particolare, rappresentate gli agricoltori diretti a nord e quelli diretti a sud come thread separati. Una volta che un contadino è sul ponte, il thread associato verrà sospeso per un periodo di tempo casuale, che rappresenta l'attraversamento del ponte. Progettate il vostro programma in modo da poter creare diversi thread per rappresentare gli agricoltori diretti a nord e quelli diretti a sud.

8.33 Nella Figura 8.7 illustriamo una funzione `transaction()` che acquisisce lock in maniera dinamica. Nel testo, descriviamo come questa funzione presenti difficoltà nell'acquisizione di lock in modo da evitare lo stallo. Utilizzando l'implementazione Java di `transaction()` fornita nel codice sorgente scaricabile per questo testo, modificalo utilizzando il metodo `System.identityHashCode()` in modo che i lock vengano acquisiti in ordine.

In questo progetto scriverete un programma che implementi l'algoritmo del banchiere discusso nel Paragrafo 8.6.3. Diversi clienti richiedono e rilasciano risorse della banca. Il banchiere soddisferà una richiesta solo se questa lascerà il sistema in uno stato sicuro. Una richiesta che lascia il sistema in uno stato non sicuro viene rifiutata. Anche se gli esempi di codice che illustrano questo progetto sono scritti in C, potete anche sviluppare una soluzione usando Java.

Il banchiere

Il banchiere prenderà in considerazione richieste da parte di n clienti per m tipi di risorse, come indicato nel Paragrafo 8.6.3. Il banchiere terrà traccia delle risorse utilizzando le seguenti strutture dati:

```
#define NUMBER_OF_CUSTOMERS 5

#define NUMBER_OF_RESOURCES 3

/* la quantità disponibile di ogni risorsa */

int available[NUMBER_OF_RESOURCES];

/* la richiesta massima di ogni cliente */

int maximum[NUMBER_OF_CUSTOMERS] [NUMBER_OF_RESOURCES];

/* la quantità attualmente assegnata a ogni cliente */

int allocation[NUMBER_OF_CUSTOMERS] [NUMBER_OF_RESOURCES];

/* le rimanenti necessità di ogni cliente */

int need[NUMBER_OF_CUSTOMERS] [NUMBER_OF_RESOURCES];
```

Il banchiere soddisferà una richiesta se questa soddisfa l'algoritmo di sicurezza delineato nel Paragrafo 8.6.3.1. Se la richiesta non lascia il sistema in uno stato sicuro, viene rifiutata. I prototipi delle funzioni per richiedere e rilasciare le risorse sono i seguenti:

```
int request_resources(int customer_num, int request[]);  
  
int release_resources(int customer_num, int release[]);
```

La funzione `request_resources()` restituisce 0 se ha successo e -1 in caso contrario.

Verifica della vostra implementazione

Create un programma interattivo che permetta all'utente di effettuare una richiesta di risorse, di rilasciare risorse o leggere i valori delle varie strutture dati (`available`, `maximum`, `allocation` e `need`) utilizzate dall'algoritmo del banchiere.

Invoke il programma passando da riga di comando il numero di risorse di ogni tipo. Per esempio, se ci sono quattro tipi di risorse, con dieci istanze del primo tipo, cinque del secondo tipo, sette del terzo tipo e otto del quarto, il programma va richiamato come segue:

```
./a.out 10 5 7 8
```

L'array `available` sarà inizializzato a questi valori.

Il tuo programma inizialmente leggerà un file contenente il numero massimo di richieste per ogni cliente. Per esempio, se ci sono cinque clienti e quattro risorse, il file di input appare come segue:

```
6,4,7,3  
4,2,3,2  
2,5,3,3  
6,3,3,2  
5,6,7,5
```

dove ogni riga nel file di input rappresenta la richiesta massima di ciascuno tipo di risorsa per ogni cliente. Il vostro programma inizializzerà l'array `maximum` a questi valori.

Il vostro programma attenderà quindi che l'utente inserisca i comandi che corrispondono a richiesta di risorse, rilascio di risorse o richiesta dei valori correnti delle diverse strutture dati. Utilizzate il comando 'RQ' per richiedere risorse, 'RL' per il rilascio di risorse e '*' per richiedere i valori delle diverse strutture dati. Per esempio, se il cliente 0 dovesse richiedere le risorse (3, 1, 2, 1), il comando inserito sarebbe il seguente:

```
RQ 0 3 1 2 1
```

Il vostro programma risponderebbe quindi se la richiesta è soddisfatta o negata utilizzando l'algoritmo di sicurezza descritto nel Paragrafo 8.6.3.1.

Allo stesso modo, se il cliente 4 dovesse rilasciare le risorse (1, 2, 3, 1), l'utente inserirebbe il seguente comando:

```
RL 4 1 2 3 1
```

Infine, se viene inserito il comando '*', il vostro programma scriverà in output i valori degli array `available`, `maximum`, `allocation` e `need`.

CAPITOLO 9

Memoria centrale

Nel Capitolo 5 si è descritto come è possibile condividere la cpu tra un insieme di processi. Uno dei risultati dello scheduling della cpu consiste nella possibilità di migliorare sia l'utilizzo della cpu sia la rapidità con cui il calcolatore risponde ai propri utenti; per ottenere questo aumento delle prestazioni, tuttavia, occorre tenere in memoria parecchi processi: la memoria deve, cioè, essere condivisa.

In questo capitolo si presentano diversi metodi di gestione della memoria. Gli algoritmi di gestione della memoria variano dall'approccio più semplice che accede all'hardware della macchina in maniera diretta ai metodi di paginazione e segmentazione. Ogni metodo presenta vantaggi e svantaggi. La scelta di un metodo specifico di gestione della memoria dipende da molti fattori, in particolar modo dall'architettura hardware del sistema, infatti molti algoritmi richiedono un supporto hardware. Per questa ragione in molti sistemi l'hardware e la gestione della memoria da parte del sistema operativo sono strettamente integrati.

9.1 Introduzione

Come abbiamo visto nel Capitolo 1, la memoria è fondamentale nelle operazioni di un moderno sistema di calcolo. La memoria consiste in un grande vettore di byte, ciascuno con il proprio indirizzo. La cpu preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; quindi viene decodificata (e ciò può comportare il prelievo di operandi dalla memoria) e poi eseguita sugli eventuali operandi; i risultati si possono scrivere in memoria. La memoria vede soltanto un flusso d'indirizzi di memoria, e non sa come sono generati (contatore di programma, indicizzazione, riferimenti indiretti, indirizzamenti immediati e così via), oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare *come* un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

L'analisi che sviluppiamo nel seguito comprende una panoramica degli aspetti basilari della gestione della memoria: l'hardware, la mappatura degli indirizzi simbolici in indirizzi fisici effettivi, e la distinzione fra indirizzamento logico e fisico. Concluderemo il paragrafo discutendo il linking dinamico e la condivisione delle librerie.

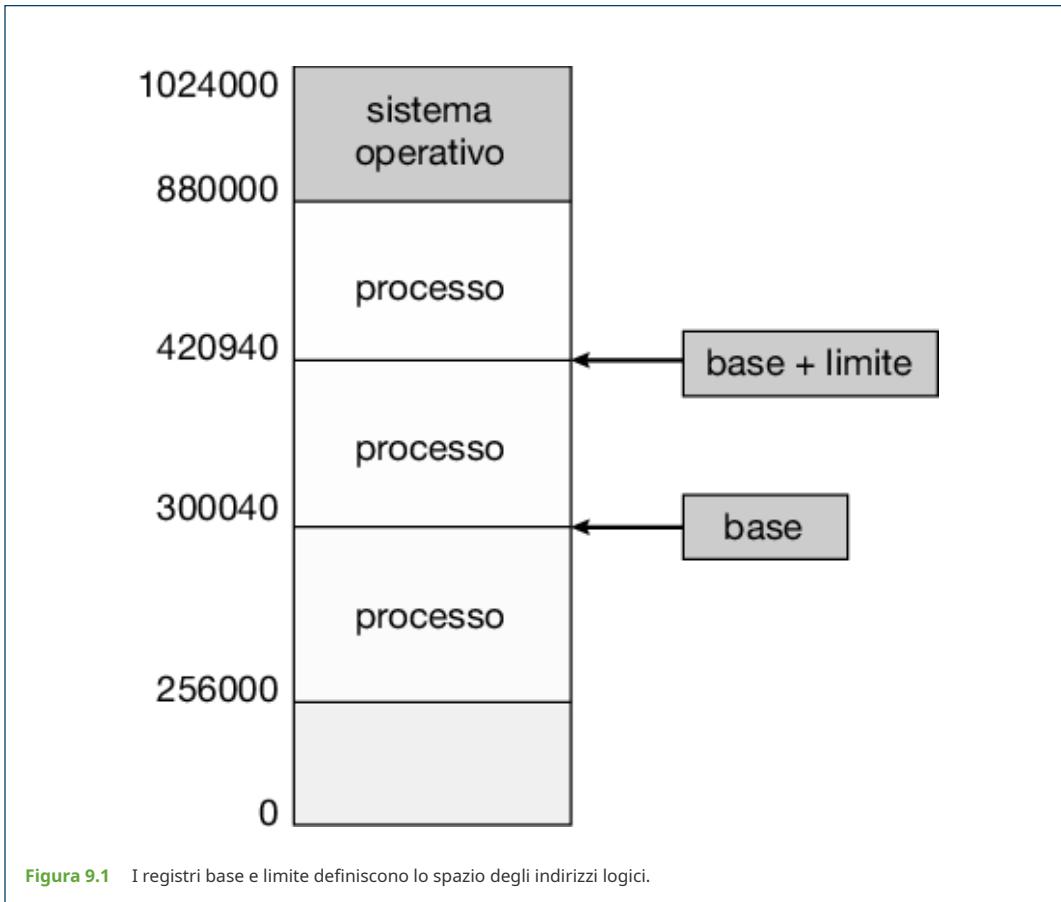
9.1.1 Hardware di base

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la cpu può accedere direttamente. Vi sono istruzioni macchina che accettano gli indirizzi di memoria come argomenti, ma nessuna accetta gli indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi di memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la cpu possa operare su di loro.

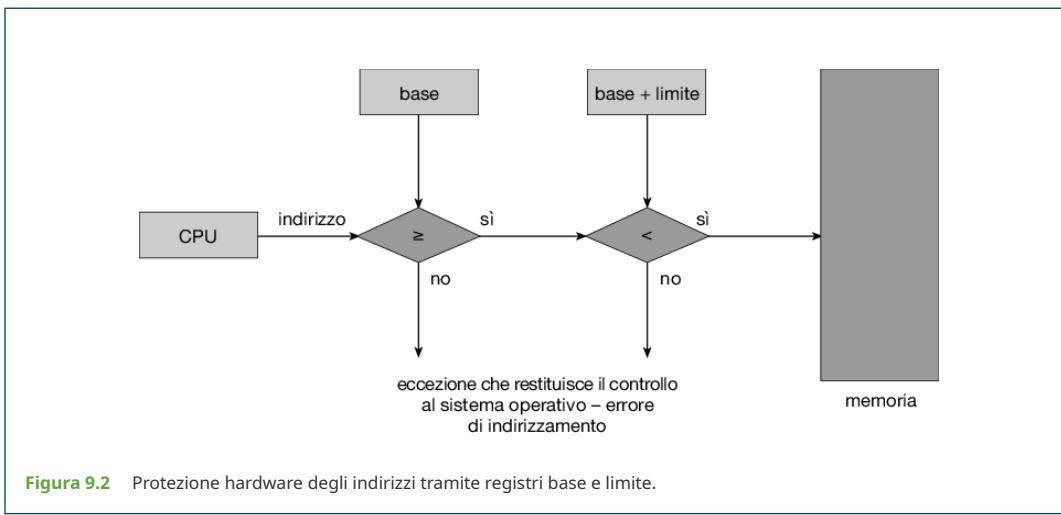
I registri incorporati nella cpu sono accessibili, in genere, nell'arco di un ciclo del clock della cpu. I core di alcune cpu sono capaci di decodificare istruzioni ed effettuare semplici operazioni sui contenuti dei registri alla velocità di una o più operazioni per ciclo. Ciò non vale per la memoria centrale, cui si accede tramite una transazione sul bus della memoria che può richiedere molti cicli di clock. In tal caso il processore entra necessariamente in stallo (*stall*), poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra cpu e memoria centrale. Di solito questo buffer di memoria, detto cache, si trova sul chip della cpu, in modo da garantire un accesso più rapido. La cache è stata descritta nel Paragrafo 1.5.5. Nella gestione di una cache interna alla cpu l'hardware si occupa direttamente di accelerare l'accesso alla memoria, senza l'intervento del sistema operativo. Ricordiamo dal Paragrafo 5.5.2 che durante uno stallo di memoria un core multithread può scambiare il thread hardware bloccato con un altro thread hardware.

Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica, ma occorre anche assicurare una corretta esecuzione delle operazioni. A tal fine, bisogna proteggere il sistema operativo dall'accesso dei processi utenti, e, in sistemi multiutente, salvaguardare i processi utenti l'uno dall'altro. Tale protezione deve essere messa in atto a livello hardware, perché solitamente il sistema operativo, per una questione di prestazioni, non interviene negli accessi della cpu alla memoria. Come si vedrà lungo tutto il capitolo, questa protezione può essere realizzata dall'hardware con meccanismi diversi. In questo paragrafo evidenziamo una delle possibili implementazioni.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato, in modo da proteggere i processi l'uno dall'altro: ciò è fondamentale per avere più processi caricati in memoria per l'esecuzione concorrente. Per separare gli spazi di memoria occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti registro base e registro limite, come illustrato nella Figura 9.1. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Per esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso agli indirizzi compresi tra 300040 e 420939, estremi inclusi.



Per mettere in atto il meccanismo di protezione, la cpu confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di una eccezione (*trap*) che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (Figura 9.2). Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati del sistema operativo o degli altri utenti.



Solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente in modalità kernel, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce tale operazione ai programmi utenti.

Grazie all'esecuzione in modalità kernel, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utenti nelle aree di memoria a loro riservate; di generare copie del contenuto di queste regioni di memoria (*dump*) a scopi diagnostici, qualora si verifichino errori; di modificare i parametri delle chiamate di sistema; di eseguire i/o da e verso la memoria utente e di fornire molti altri servizi. Consideriamo, per esempio, che un sistema operativo per un sistema multiprocesssing deve effettuare cambi di contesto, memorizzando lo stato di un processo dai registri nella memoria centrale prima di caricare il contesto del processo successivo dalla memoria centrale nei registri.

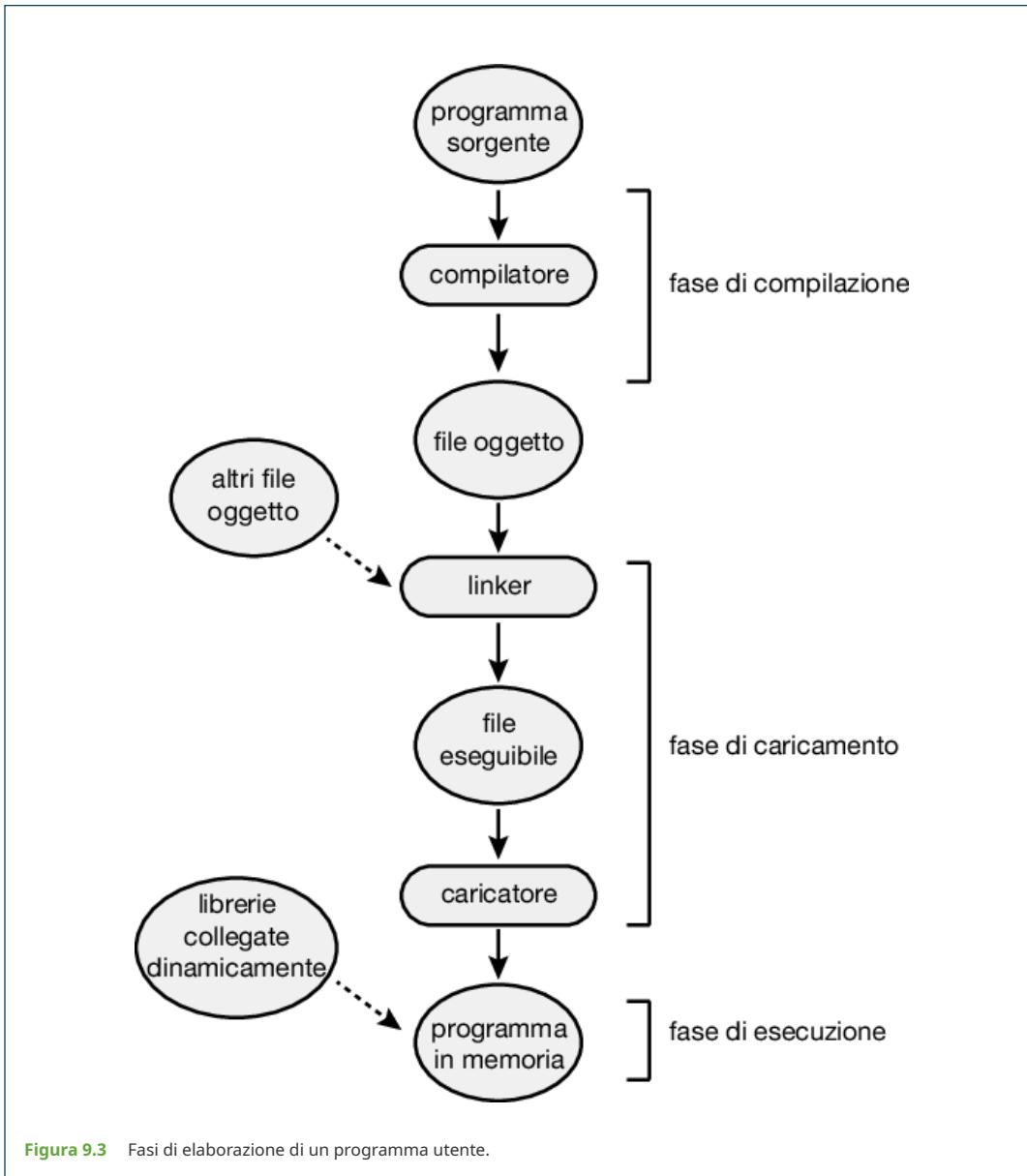
9.1.2 Associazione degli indirizzi

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito nel contesto di un processo (come descritto nel Paragrafo 2.5), dove diventa idoneo per l'esecuzione su una cpu disponibile.

Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria.

La maggior parte dei sistemi consente ai processi utenti di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio d'indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere 00000. Vedremo più avanti come un programma utente inserisce un processo nella memoria fisica.

Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari passi, alcuni dei quali possono essere facoltativi (Figura 9.3). In questi passi gli indirizzi sono rappresentabili in modi diversi. Generalmente gli indirizzi del programma sorgente sono simbolici (per esempio, la variabile *count*). Un compilatore di solito associa (*bind*) questi indirizzi simbolici a indirizzi rilocabili (per esempio, “14 byte dall'inizio di questo modulo”). L'editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (per esempio, 74014). Ogni associazione rappresenta una corrispondenza da uno spazio d'indirizzi a un altro.



Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi passo del seguente percorso.

- **Compilazione.** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare codice assoluto. Se, per esempio, è noto a priori che un processo utente inizia alla locazione R , anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice.
- **Caricamento.** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare codice rilocabile. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- **Esecuzione.** Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. Per realizzare questo schema è necessario disporre di hardware specializzato; questo argomento è trattato nel Paragrafo 9.1.3. La maggior parte dei sistemi operativi general-purpose impiega questo metodo.

Una gran parte di questo capitolo è dedicata alla spiegazione di come i vari tipi di associazione degli indirizzi si possano realizzare efficacemente in un calcolatore e, inoltre, alla discussione dell'appropriato supporto hardware per queste funzioni.

9.1.3 Spazi di indirizzi logici e fisici

Un indirizzo generato dalla cpu è normalmente chiamato indirizzo logico, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel registro dell'indirizzo di memoria (*memory address register*, mar) è normalmente chiamato indirizzo fisico.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con il metodo di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce, di solito, agli indirizzi logici col termine indirizzi virtuali; in questo testo si usano tali termini in modo intercambiabile. L'insieme di tutti gli indirizzi logici generati da un programma è lo spazio degli indirizzi logici; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo spazio degli indirizzi fisici. Quindi, con lo schema di associazione degli indirizzi nella fase d'esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto unità di gestione della memoria (*memory-management unit*, mmu), come mostrato nella Figura 9.4. Come viene discusso nei Paragrafi dal 9.2 al 9.3, si può scegliere tra diversi metodi di realizzazione di tale associazione. Per ora illustriamo un semplice schema di associazione degli indirizzi tramite mmu, che è una generalizzazione dello schema con registro di base descritto nel Paragrafo 9.1.1. Com'è illustrato nella Figura 9.5, il registro di base è ora denominato registro di rilocazione: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si somma a tale indirizzo il valore contenuto nel registro di rilocazione. Per esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 è mappato alla locazione 14346.

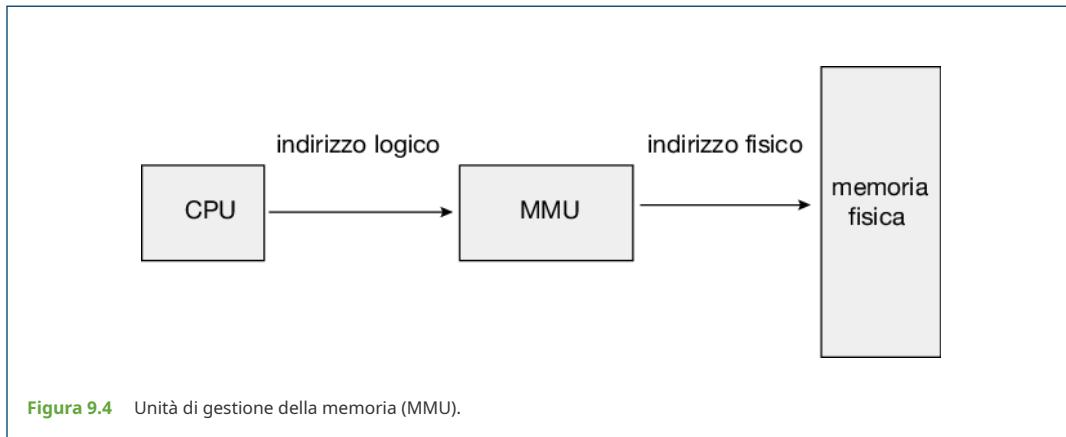


Figura 9.4 Unità di gestione della memoria (MMU).

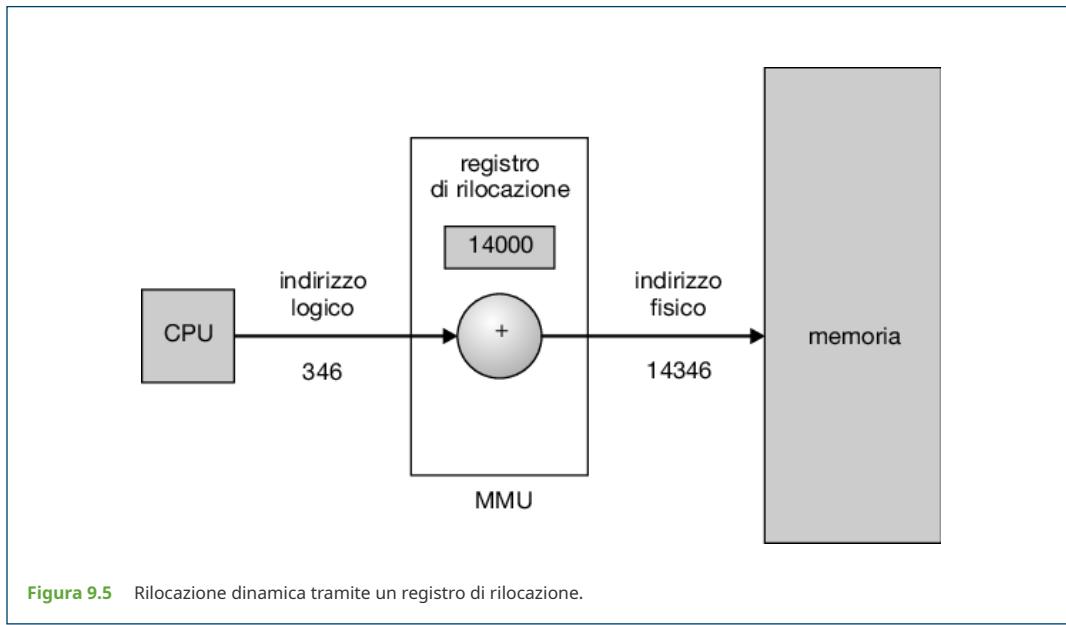


Figura 9.5 Rilocazione dinamica tramite un registro di rilocazione.

Il programma utente non vede mai i reali indirizzi fisici. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò sempre come il numero 346. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi logici, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici. Questa forma di collegamento nella fase d'esecuzione è stata trattata nel Paragrafo 9.1.2. La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a max) e gli indirizzi fisici (nell'intervallo da $r + 0$ a $r + max$ per un valore di base r). Il programma utente genera solo indirizzi logici e pensa che il processo sia eseguito nelle posizioni da 0 a max . Tuttavia questi indirizzi logici devono essere mappati in indirizzi fisici prima d'essere usati. Il concetto di *spazio d'indirizzi logici* mappato su uno *spazio d'indirizzi fisici* separato è fondamentale per una corretta gestione della memoria.

9.1.4 Caricamento dinamico

Nella discussione svolta fin'ora, era necessario che l'intero programma e i dati di un processo fossero presenti nella memoria fisica perché il processo potesse essere eseguito. La dimensione di un processo era quindi limitata alle dimensioni della memoria fisica. Per migliorare l'utilizzo della memoria si può ricorrere al caricamento dinamico (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono su disco in un formato di caricamento rilocabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, controlla innanzitutto che sia stata caricata. Se non è stata caricata, si richiama il *linking loader* rilocabile per caricare in memoria la procedura richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata.

Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura viene caricata solo quando serve. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola.

Il caricamento dinamico non richiede un supporto particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

9.1.5 Linking dinamico e librerie condivise

Le librerie collegate dinamicamente (dll) sono librerie di sistema che vengono collegate ai programmi utente quando questi vengono eseguiti (si faccia riferimento alla Figura 9.3). Alcuni sistemi operativi consentono solo il collegamento statico (*static linking*), in cui le librerie di sistema sono trattate come qualsiasi altro modulo oggetto e combinate dal loader nell'immagine binaria del programma. Il concetto di linking dinamico, invece, è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell'esecuzione, si differisce il collegamento. Questa modalità si usa soprattutto con le librerie di sistema, per esempio la libreria standard del linguaggio C. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre, all'interno dell'eseguibile, di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Ciò porta a un aumento della dimensione del file eseguibile e a un possibile spreco di spazio in memoria centrale. Un secondo vantaggio delle dll è che possono essere condivise tra più processi, in modo che solo un'istanza della dll risieda nella memoria centrale. Per questo motivo, le dll sono anche conosciute come librerie condivise e sono ampiamente utilizzate nei sistemi Windows e Linux.

Quando un programma fa riferimento a una routine presente in una libreria dinamica, il loader individua la dll e, se necessario, la carica in memoria. Gli indirizzi sono poi modificati in modo che facciano riferimento alle funzioni nella libreria dinamica secondo la posizione della dll in memoria.

Le librerie collegate dinamicamente possono essere estese mediante aggiornamenti (per esempio, per correggere bug) o anche sostituite da una nuova versione. In questo caso, tutti i programmi che fanno riferimento alla libreria utilizzeranno automaticamente la nuova versione. Senza linking dinamico tutti i programmi di questo tipo avrebbero bisogno di essere nuovamente linkati per accedere alla nuova libreria. Affinché i programmi non eseguano accidentalmente nuove versioni di librerie incompatibili, le informazioni sulla versione sono incluse sia nel programma sia nella libreria. È possibile caricare in memoria più di una versione della stessa libreria e ogni programma si serve delle informazioni sulla versione per decidere quale copia della libreria utilizzare. Se le modifiche sono di piccola entità, il numero di versione resta invariato; se l'entità delle modifiche diviene rilevante, si aumenta anche il numero di versione. Perciò, solo i programmi compilati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporate nella libreria stessa. I programmi collegati prima dell'installazione della nuova libreria continuano ad avvalersi della vecchia libreria.

A differenza del caricamento dinamico, il linking dinamico e le librerie condivise richiedono generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria. Questo concetto è sviluppato nell'analisi della paginazione, nel Paragrafo 9.3.4.

9.2 Allocazione contigua della memoria

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. In questo paragrafo si tratta il primo metodo di allocazione della memoria, l'allocazione contigua.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utente. Il sistema operativo si può collocare sia nella parte bassa sia nella parte alta della memoria. Questa decisione dipende da molti fattori, per esempio dalla posizione del vettore delle interruzioni. Visto però che molti sistemi operativi (inclusi Linux e Windows) collocano il sistema operativo in memoria alta, discutiamo solo questa situazione.

Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'allocazione contigua della memoria, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo. Prima di discutere ulteriormente questo schema di allocazione della memoria dobbiamo affrontare il problema della protezione della memoria.

9.2.1 Protezione della memoria

Prima di trattare l'allocazione della memoria, dobbiamo soffermarci sul problema della protezione della memoria. Possiamo evitare che un processo acceda alla memoria che non gli appartiene combinando due idee discusse in precedenza. Se abbiamo un sistema con un registro di rilocazione (Paragrafo 9.1.3) e un registro limite (Paragrafo 9.1.1) abbiamo già raggiunto il nostro obiettivo. Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600. Ogni indirizzo logico deve cadere nell'intervallo specificato dal registro limite; la mmu fa corrispondere *dinamicamente* l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione (Figura 9.6), e invia l'indirizzo risultante alla memoria.

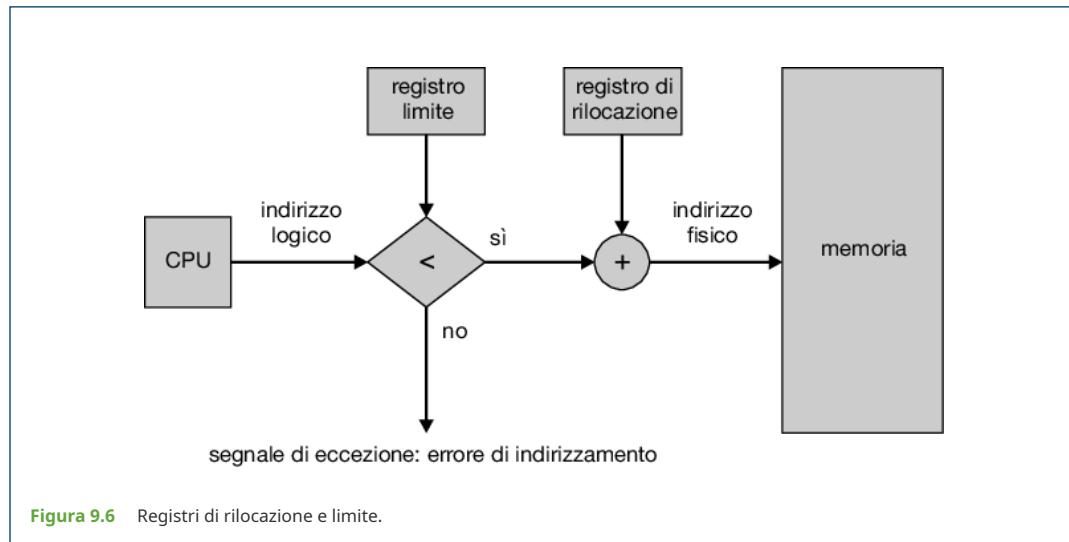


Figura 9.6 Registri di rilocazione e limite.

Quando lo scheduler della cpu seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla cpu con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione.

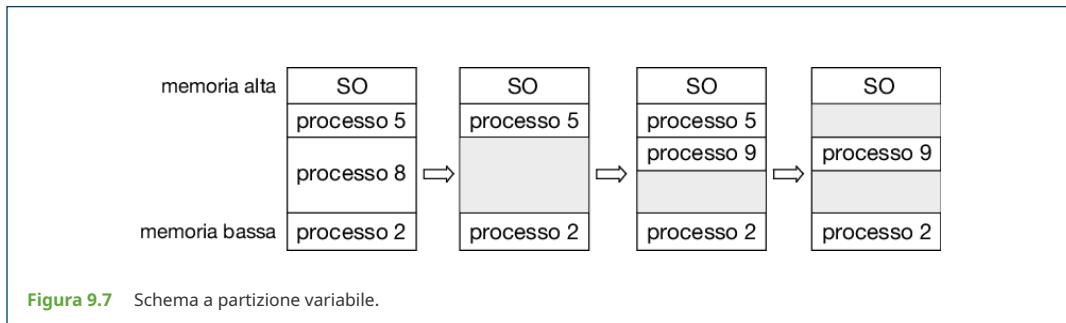
Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi. Se un driver di periferica non è attualmente in uso, ha poco senso mantenerlo in memoria. È meglio, invece, caricarlo in memoria solo quando è necessario e rimuoverlo quando non lo è più, destinando la sua memoria ad altre esigenze.

9.2.2 Allocazione della memoria

Siamo ora pronti per parlare dell'allocazione della memoria. Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in partizioni di dimensione variabile, dove ciascuna partizione può contenere esattamente un processo. In questo schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e

quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un buco (*hole*). Nel lungo periodo, come si vede nel seguito, la memoria contiene una serie di buchi di diverse dimensioni.

La Figura 9.7 illustra questo schema. Inizialmente, la memoria è completamente utilizzata e contiene i processi 5, 8 e 2. Dopo l'uscita del processo 8 si forma un unico buco. Successivamente, entra in memoria il processo 9, per il quale viene allocata memoria. Quindi il processo 5 viene rimosso, producendo due buchi non contigui.



Quando i processi entrano nel sistema, il sistema operativo prende in considerazione i loro requisiti di memoria e la quantità di spazio di memoria disponibile e determina a quali processi allocare memoria. Quando gli viene assegnato dello spazio, un processo viene caricato in memoria e può quindi competere per il controllo della cpu. Quando termina, rilascia la memoria che gli era stata assegnata e il sistema operativo può impiegarla per un altro processo.

Che cosa succede quando non c'è sufficiente memoria per soddisfare le esigenze di un processo in arrivo? Un'opzione semplice è rifiutare il processo e fornire un messaggio di errore appropriato. In alternativa, possiamo inserire il processo in una coda di attesa. Quando, in seguito, la memoria viene rilasciata, il sistema operativo controlla la coda di attesa per determinare se può soddisfare le richieste di memoria di un processo in attesa.

In generale è sempre presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande.

Questa procedura è una particolare istanza del più generale problema di allocazione dinamica della memoria, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- First-fit. Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- Best-fit. Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, a meno che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- Worst-fit. Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, a meno che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio best-fit.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte nessuno dei due è chiaramente migliore dell'altro per quel che riguarda l'utilizzo della memoria ma, in genere, first-fit è più veloce.

9.2.3 Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di frammentazione esterna. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi.

Sia che si adotti l'uno o l'altro, l'impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati il first-fit, in altri dà migliori risultati il best-fit; un altro elemento rilevante è quale sia l'estremità assegnata di un blocco libero (se la parte inutilizzata è quella in alto o quella in basso). A prescindere dal tipo di algoritmo usato, la frammentazione esterna è un problema.

La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. L'analisi statistica dell'algoritmo first-fit, per esempio, rivela che, pur con alcune ottimizzazioni, per n blocchi assegnati, si perdono altri 0,5 n blocchi a causa della

frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria. Questa caratteristica è nota come regola del 50 per cento.

La frammentazione può essere interna oltre che esterna. Si consideri uno schema a partizioni multiple con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un buco di 2 byte. L'overhead necessario per tener traccia di questo buco è nettamente più grande del buco stesso. Il metodo generale per superare questo problema prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta. La differenza tra questi due numeri è la frammentazione interna che consiste nella memoria inutilizzata all'interno di una partizione.

Una soluzione al problema della frammentazione esterna è data dalla compattazione. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è effettuata nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si effettua nella fase d'esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria: tutti i buchi si spostano nell'altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un'altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Questa è la strategia utilizzata nella paginazione, la più comune tecnica di gestione della memoria nei sistemi elaborativi. La paginazione sarà descritta nel paragrafo successivo.

La frammentazione è un problema generale che può verificarsi ogni volta che si opera su blocchi di dati. Questa tematica sarà approfondita ulteriormente nei capitoli sulla gestione della memoria secondaria (dal Capitolo 11 al Capitolo 15).

9.3 Paginazione

La gestione della memoria discussa finora richiedeva che lo spazio di indirizzamento fisico di un processo fosse contiguo. Introduciamo ora la paginazione, uno schema di gestione della memoria che consente allo spazio di indirizzamento fisico di un processo di essere non contiguo. La paginazione evita la frammentazione esterna e la necessità di compattazione, due problemi che affliggono l'allocazione di memoria contigua. Visti i numerosi vantaggi offerti, la paginazione, nelle sue varie forme, viene utilizzata nella maggior parte dei sistemi operativi, da quelli destinati a server di grandi dimensioni a quelli per dispositivi mobili. L'implementazione della paginazione è frutto della cooperazione tra il sistema operativo e l'hardware del computer.

9.3.1 Metodo di base

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione fissa, detti frame, e nel suddividere la memoria logica in blocchi di pari dimensione, detti pagine. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria o dal file system. La memoria ausiliaria è divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria o di blocchi composti da più frame.

numero di pagina	offset di pagina
p	d

Questa idea piuttosto semplice fornisce grandi funzionalità e ha diverse ramificazioni. Per esempio, ora lo spazio degli indirizzi logici è totalmente separato dallo spazio degli indirizzi fisici e dunque un processo può avere uno spazio degli indirizzi logici a 64 bit anche se il sistema ha meno di 2^{64} byte di memoria fisica.

Ogni indirizzo generato dalla cpu è diviso in due parti: un numero di pagina (p), e un offset di pagina (d):

Il numero di pagina serve come indice per la tabella delle pagine relativa al processo (Figura 9.8). La tabella delle pagine contiene l'indirizzo di base di ciascun frame nella memoria fisica e l'offset è la posizione nel frame a cui viene fatto riferimento. Pertanto, l'indirizzo di base del frame viene combinato con l'offset della pagina per definire l'indirizzo di memoria fisica. Il modello di paginazione della memoria è mostrato nella Figura 9.9.

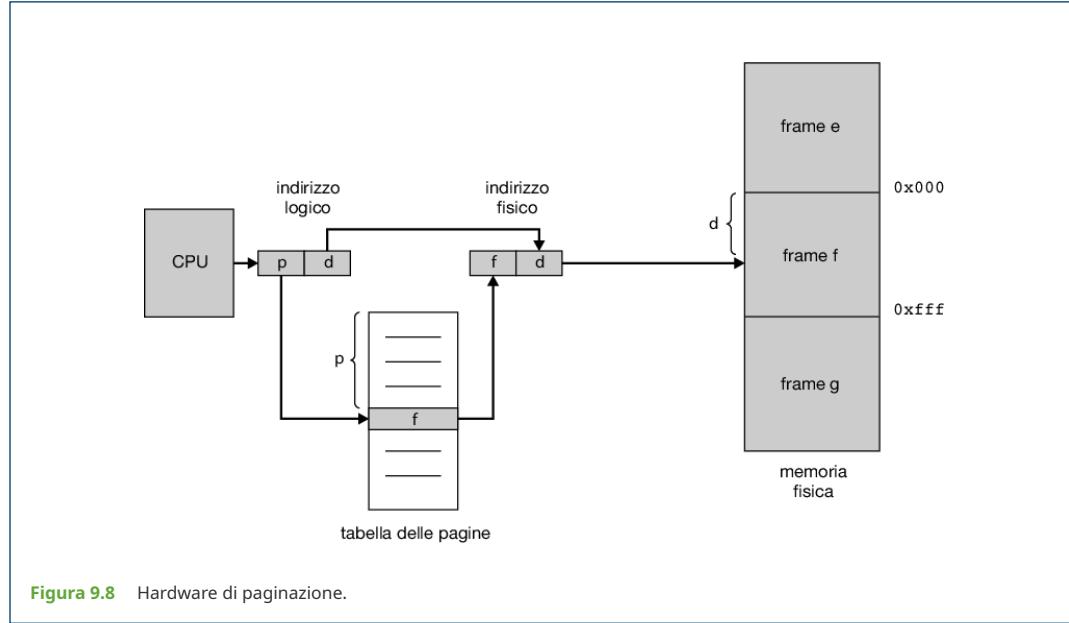


Figura 9.8 Hardware di paginazione.

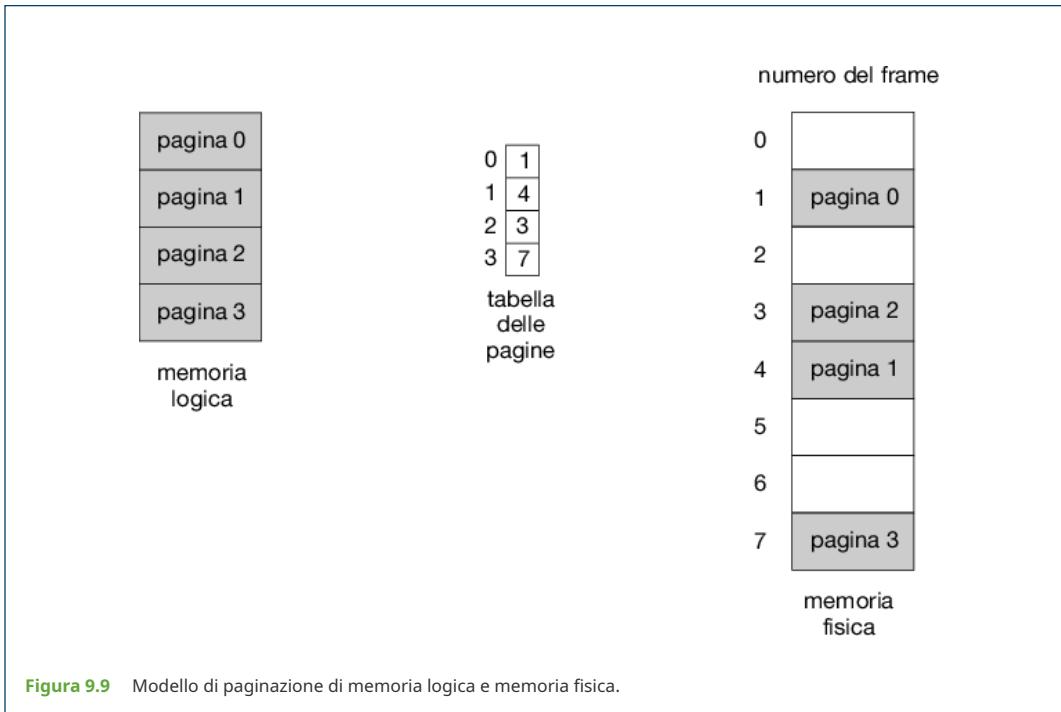


Figura 9.9 Modello di paginazione di memoria logica e memoria fisica.

Descriviamo di seguito i passaggi adottati dalla mmu per tradurre un indirizzo logico generato dalla cpu in un indirizzo fisico.

1. Estrarre il numero di pagina p e utilizzarlo come indice nella tabella delle pagine.
2. Estrarre il numero di frame f corrispondente dalla tabella delle pagine.
3. Sostituire il numero di pagina p nell'indirizzo logico con il numero di frame f .

L'offset d non cambia e pertanto non viene sostituito; il numero di frame e l'offset determinano insieme l'indirizzo fisico.

La dimensione di una pagina, così come quella di un frame, è definita dall'hardware ed è, in genere, una potenza di 2 compresa tra 4 kb e 1 gb, a seconda dell'architettura. La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e offset di pagina. Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una pagina è di 2^n byte, allora gli $m - n$ bit più significativi di un indirizzo logico indicano il numero di pagina, e gli n bit meno significativi indicano l'offset di pagina. L'indirizzo logico ha quindi la forma seguente:

numero di pagina	offset di pagina
p	d
$m - n$	n

dove p è un indice della tabella delle pagine e d è l'offset all'interno della pagina indicata da p .

Come esempio concreto, anche se minimo, si consideri la memoria illustrata nella Figura 9.10; qui, nell'indirizzo logico, $n = 2$ e $m = 4$. Utilizzando pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), vediamo come si fa a corrispondere la memoria vista dal programmatore alla memoria fisica. L'indirizzo logico 0 è la pagina 0 con offset 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 [= (5 × 4) + 0]. All'indirizzo logico 3 (pagina 0, offset 3) corrisponde l'indirizzo fisico 23 [= (5 × 4) + 3]. Per quel che riguarda l'indirizzo logico 4 (pagina 1, offset 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico 24 [= (6 × 4) + 0]. All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

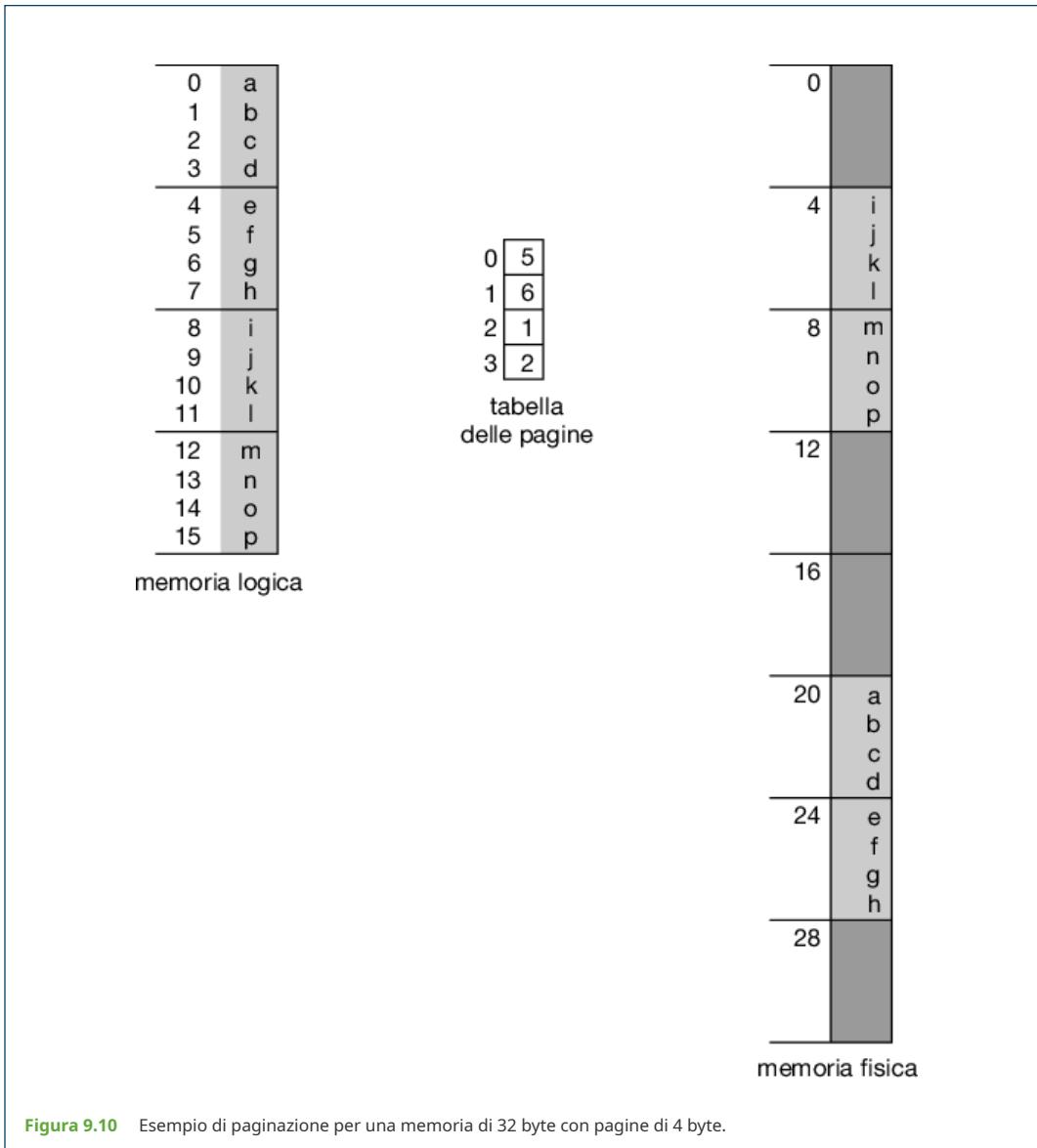


Figura 9.10 Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

Il lettore può aver notato che la paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. L'uso della tabella delle pagine è simile all'uso di una tabella di registri base (o di rilocazione), uno per ciascun frame.

Con la paginazione si evita la frammentazione esterna: *qualsiasi* frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l'*ultimo* frame assegnato può non essere completamente pieno. Se, per esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di $2048 - 1086 = 962$ byte. Il caso peggiore si ha con un processo che necessita di n pagine più un byte: si assegnano $n + 1$ frame, quindi si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della pagina, si deve prevedere una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un overhead che si riduce all'aumentare delle dimensioni delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'i/o su disco è più efficiente (Capitolo 11). Generalmente, nel tempo la dimensione delle pagine è cresciuta col crescere dei processi, dei dati e della memoria centrale; attualmente la dimensione tipica delle pagine è compresa tra 4 kb e 8 kb; in alcuni sistemi può essere anche maggiore. Per esempio, su sistemi x86-64, Windows 10 supporta dimensioni di pagina di 4 kb e 2 mb, mentre Linux supporta una dimensione di pagina predefinita, in genere di 4 kb, e pagine più grandi la cui dimensione dipende dall'architettura, chiamate pagine enormi (*huge page*).

Spesso, su una cpu a 32 bit, ogni voce della tabella delle pagine è lunga 4 byte, ma questa dimensione può variare. Un singola voce di 32 bit può puntare a uno dei 2^{32} frame di pagina fisici. Se la dimensione di un frame è di 4 kb (2^{12}), un sistema con voci della tabella delle pagine di 4 byte può indirizzare 2^{44} byte (o 16 tb) di memoria fisica. Va notato che la dimensione della memoria fisica in un sistema di memoria paginata è differente dalla dimensione logica massima di un processo. Quando analizzeremo ulteriormente la paginazione, introdurremo altre informazioni che devono essere mantenute nelle voci della tabella delle pagine. Queste informazioni riducono il numero di bit disponibili per indirizzare i frame. Un sistema con voci di 32 bit è quindi in grado di indirizzare una quantità di memoria fisica inferiore al valore massimo teoricamente possibile. Una cpu a 32 bit utilizza indirizzi a 32 bit, il che significa che lo spazio di memoria di un processo può essere al massimo di 2^{32} byte (4 gb). La paginazione ci consente quindi di utilizzare una memoria fisica che è decisamente più grande rispetto a quella che potrebbe essere indirizzata dalla lunghezza del puntatore agli indirizzi della cpu.

Quando si deve eseguire un processo, il sistema esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede n pagine, devono essere disponibili almeno n frame che, se ci sono, vengono assegnate al processo. Si carica la prima pagina in uno dei frame assegnati e s'inserisce il numero del frame nella tabella delle pagine relativa al processo in questione. La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine, e così via (Figura 9.11).

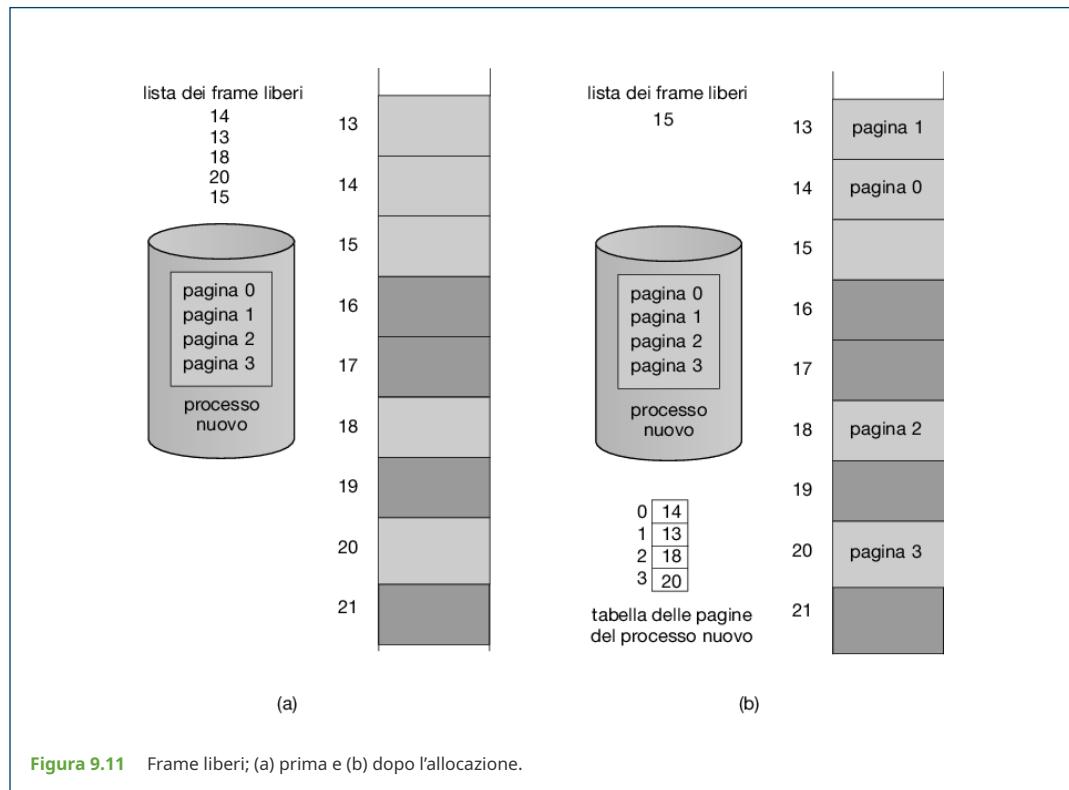


Figura 9.11 Frame liberi; (a) prima e (b) dopo l'allocazione.

Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dal programmatore e l'effettiva memoria fisica: il programmatore vede la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dal programmatore e la memoria fisica è colmata dall'hardware di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti. Questa corrispondenza non è visibile ai programmatore ed è controllata dal sistema operativo. Si noti che un processo utente, per definizione, non può accedere alle zone di memoria che non gli appartengono. Non ha modo di accedere alla memoria oltre quel che è previsto dalla sua tabella delle pagine, e tale tabella contiene soltanto le pagine che appartengono al processo.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei dettagli della allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata tabella dei frame, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utenti operano nello spazio utente, e tutti gli indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una chiamata di sistema (per esempio per eseguire un'operazione di i/o) e fornisce un indirizzo come parametro (per esempio l'indirizzo di un buffer), si deve tradurre questo indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel contatore di programma e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che

il sistema operativo deve associare esplicitamente un indirizzo fisico a un indirizzo logico. La stessa copia è usata anche dal dispatcher della cpu per impostare l'hardware di paginazione quando a un processo sta per essere assegnata la cpu. La paginazione fa quindi aumentare la durata dei cambi di contesto.

OTTENERE LA DIMENSIONE DELLA PAGINA SU SISTEMI LINUX

Su un sistema Linux la dimensione della pagina varia a seconda dell'architettura. Vi sono diversi modi per ottenere le dimensioni della pagina. Un approccio è quello di utilizzare la chiamata di sistema `getpagesize()`. In alternativa è possibile eseguire da riga di comando:

```
getconf PAGESIZE
```

Ciascuna di queste tecniche restituisce la dimensione della pagina espressa in byte.

9.3.2 Supporto hardware alla paginazione

Poiché ogni processo ha la sua tabella delle pagine, un puntatore alla tabella di pagina viene memorizzato insieme ai valori di altri registri (come il puntatore alle istruzioni) nel *process control block* di ciascun processo. Quando lo scheduler della cpu seleziona un processo per l'esecuzione, deve ricaricare i registri utente e i valori appropriati della tabella delle pagine hardware dalla tabella delle pagine utente memorizzata.

L'implementazione hardware della tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice è implementata come un insieme di registri hardware dedicati ad alta velocità, così da rendere la traduzione dell'indirizzo molto efficiente. Tuttavia, questo approccio aumenta il tempo dei cambi di contesto, poiché durante un cambio di contesto ogni registro dovrà essere scambiato.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. Però la maggior parte dei calcolatori contemporanei usa tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima viene invece mantenuta nella memoria principale e un registro di base della tabella delle pagine (*page-table base register*, ptbr) punta alla tabella stessa. Il cambio della tabella delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto.

9.3.2.1 TLB

La memorizzazione della tabella delle pagine nella memoria principale può favorire cambi di contesto più rapidi, ma può anche comportare tempi di accesso alla memoria più lenti. Supponiamo di voler accedere alla posizione i . Per farlo, occorre far riferimento alla tabella delle pagine usando il valore contenuto nel ptbr aumentato dell'offset relativo alla pagina di i , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato all'offset rispetto all'inizio della pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo sono necessari *due* accessi alla memoria per accedere ai dati (uno per l'elemento della tabella delle pagine e uno per il dato effettivo). L'accesso alla memoria è quindi rallentato di un fattore 2, un ritardo considerato intollerabile nella maggior parte delle circostanze.

La soluzione tipica a questo problema consiste nell'impiego del tlb (*translation look-aside buffer*), una speciale, piccola cache hardware. Il tlb è una memoria associativa ad alta velocità in cui ogni elemento consiste di due parti: una chiave (o *tag*) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida e in un hardware moderno è parte della pipeline delle istruzioni: non induce dunque nessuna penalizzazione in termini di prestazioni. Per poter eseguire la ricerca in uno stadio della pipeline, tuttavia, il tlb deve essere di dimensioni ridotte, in genere contenute tra le 32 e le 1.024 voci. Alcune cpu implementano tlb separate per istruzioni e dati, in modo da poter raddoppiare il numero di voci tlb disponibili, poiché le due ricerche vengono effettuate in diversi stadi della pipeline. Questo sviluppo rappresenta un esempio di evoluzione della tecnologia delle cpu: i sistemi si sono evoluti passando dall'assenza di tlb fino ad avere più livelli di tlb, proprio come nel caso delle cache.

Il tlb si usa insieme con le tabelle delle pagine nel modo seguente: il tlb contiene una piccola parte degli elementi della tabella delle pagine; quando la cpu genera un indirizzo logico, si presenta il suo numero di pagina al tlb; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Come abbiamo appena menzionato, queste operazioni vengono eseguite come parte della pipeline delle istruzioni all'interno della cpu, senza penalizzare il sistema rispetto ad altri sistemi che non implementano la paginazione.

Se nel tlb non è presente il numero di pagina, situazione nota come insuccesso del tlb (*tlb miss*), si deve consultare la tabella delle pagine in memoria (i passi da seguire sono illustrati nel Paragrafo 9.3.1). A seconda della cpu, questa operazione può essere effettuata automaticamente a livello hardware oppure per mezzo di un interrupt al sistema operativo. Il numero del frame così ottenuto viene

usato per accedere alla memoria (Figura 9.12). Inoltre, i numeri della pagina e del frame vengono inseriti nel tlb, e al riferimento successivo la ricerca sarà molto più rapida.

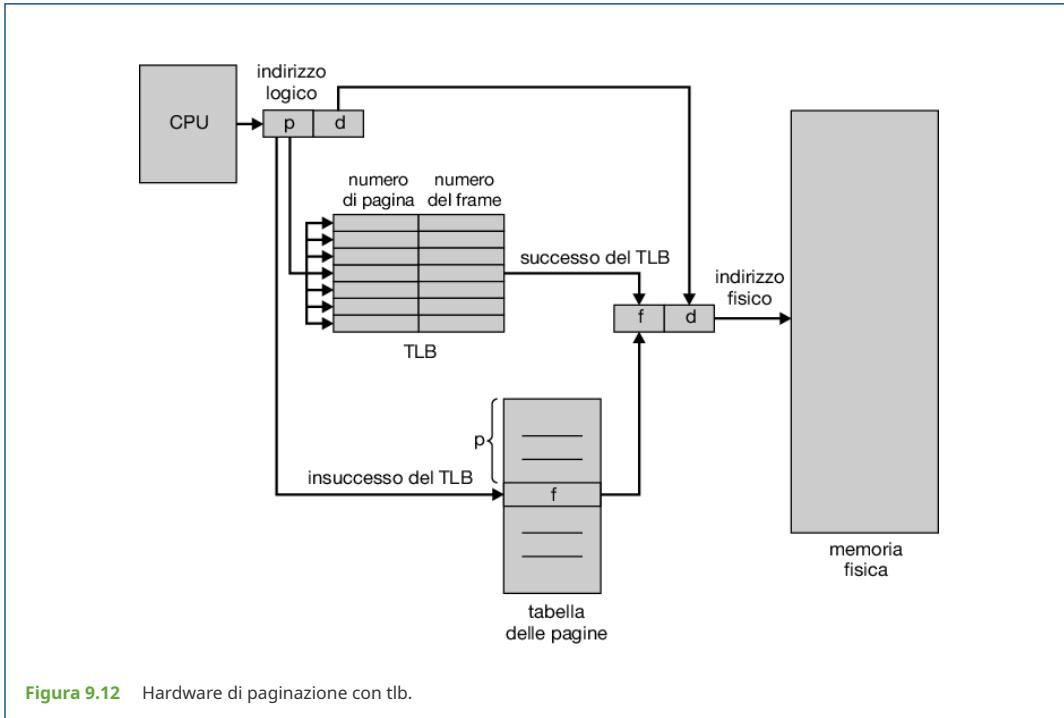


Figura 9.12 Hardware di paginazione con tlb.

Se il tlb è già pieno d'elementi, occorre sceglierne uno per sostituirlo. I criteri di sostituzione variano dalla scelta dell'elemento usato meno recentemente (lru), a una politica round-robin, fino alla scelta casuale. Alcune cpu permettono al sistema operativo di partecipare alla sostituzione lru di elementi, mentre altre gestiscono in autonomia questa operazione. Inoltre alcuni tlb consentono che certi elementi siano vincolati (*wired down*), cioè non si possano rimuovere dal tlb; in genere si vincolano gli elementi per il codice chiave del kernel.

Alcuni tlb memorizzano gli identificatori dello spazio d'indirizzi (*address-space identifier*, asid) in ciascun elemento del tlb. Un asid identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, il tlb si assicura che l'asid per il processo attualmente in esecuzione corrisponda all'asid associato alla pagina virtuale. La mancata corrispondenza dell'asid viene trattata come un tlb miss. Oltre a fornire la protezione dello spazio d'indirizzi, l'asid consente che il tlb contenga nello stesso istante elementi di diversi processi. Se il tlb non permette l'uso di asid distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio a ogni cambio di contesto, si deve cancellare (*flush*) il tlb, in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci vecchi elementi del tlb contenenti indirizzi virtuali validi, ma con indirizzi fisici corrispondenti sbagliati o non validi, lasciati dal precedente processo.

La percentuale di volte che il numero di pagina di interesse si trova nel tlb è detta tasso di successi (*hit ratio*). Un tasso di successi dell'80 per cento significa che il numero di pagina desiderato si trova nel tlb nell'80 per cento dei casi. Se sono necessari 10 nanosecondi per accedere alla memoria, allora un accesso alla memoria mappata nel tlb richiede 10 nanosecondi. Se, invece, il numero non è contenuto nel tlb, occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (10 nanosecondi), quindi accedere al byte desiderato in memoria (10 nanosecondi); in totale sono necessari 20 nanosecondi. Stiamo in questo caso supponendo che una ricerca nella tabella delle pagine richieda un solo accesso alla memoria, ma, come vedremo in seguito, potrebbero talvolta essere necessari più accessi. Per calcolare il tempo effettivo d'accesso alla memoria occorre tener conto della probabilità dei due casi:

$$\text{tempo effettivo d'accesso} = 0,80 \times 10 + 0,20 \times 20 = 12 \text{ nanosecondi}$$

In questo esempio si verifica un rallentamento del 20 per cento nel tempo medio d'accesso alla memoria (da 10 a 12 nanosecondi). Per un tasso di successi del 99 per cento, molto più realistico, si ottiene il seguente risultato:

$$\text{tempo effettivo d'accesso} = 0,99 \times 10 + 0,01 \times 20 = 10,1 \text{ nanosecondi}$$

Con questo tasso di successi, il rallentamento del tempo d'accesso alla memoria scende all'1 per cento.

Come abbiamo osservato in precedenza, le cpu di oggi possono fornire più livelli di tlb. Il calcolo dei tempi di accesso alla memoria nelle cpu moderne diventa quindi molto più complicato di quanto illustrato nell'esempio precedente. Per esempio, la cpu Intel Core i7

ha un tlb L1 da 128 elementi per le istruzioni e un tlb L1 da 64 elementi per i dati. In caso di insuccesso sul tlb L1, sono necessari sei cicli di cpu per cercare la voce sul tlb L2 da 512 elementi. Un insuccesso sul tlb L2 significa che la cpu deve attraversare le voci della tabella delle pagine in memoria per trovare l'indirizzo del frame associato, il che può richiedere centinaia di cicli, oppure generare un'interruzione per il sistema operativo affinché esegua lo stesso lavoro.

Un'analisi completa delle prestazioni in un tale sistema richiederebbe informazioni sul tasso di insuccesso di ogni livello di tlb. Da quanto abbiamo detto possiamo tuttavia dedurre che le caratteristiche hardware possono condizionare in maniera significativa le prestazioni della memoria e che le migliorie al sistema operativo (come la paginazione) possono indurre modifiche hardware e, a loro volta, essere da queste influenzate (come nel caso del tlb). L'impatto del tasso di successi nel tlb è ulteriormente analizzato nel Capitolo 10.

I tlb sono una caratteristica hardware e in quanto tali potrebbero sembrare di scarso interesse per i sistemi operativi e i loro progettisti. I progettisti, tuttavia, hanno bisogno di capire la funzione e le caratteristiche dei tlb, che variano a seconda della piattaforma hardware. Per un funzionamento ottimale, il progetto di un sistema operativo per una data piattaforma deve implementare la paginazione basandosi sul progetto dei tlb della piattaforma. Analogamente, un cambiamento nel progetto del tlb (per esempio, tra generazioni diverse di cpu Intel) può rendere necessaria una modifica nell'implementazione della paginazione dei sistemi operativi che utilizzano questa tecnica.

9.3.3 Protezione

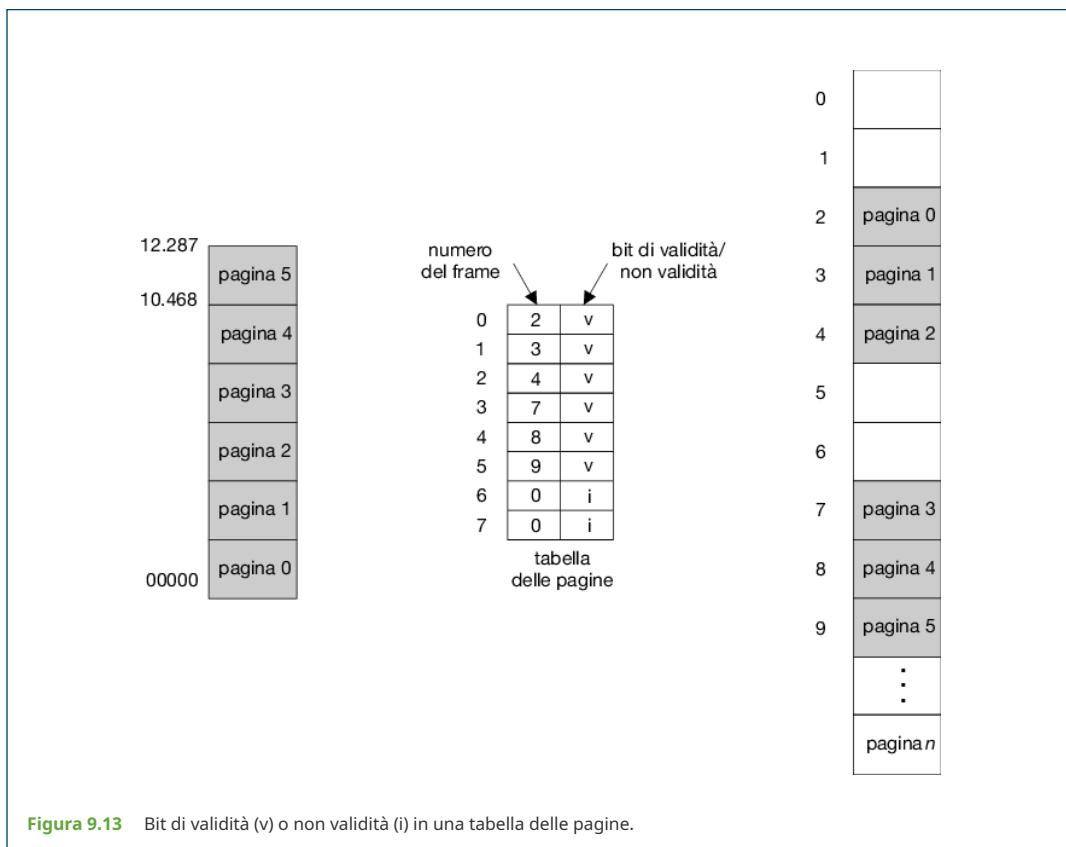
In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero corretto del frame; quindi mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo causerebbe la generazione di un'eccezione hardware per il sistema operativo, si avrebbe cioè una violazione della protezione della memoria.

Questo metodo si può facilmente estendere per fornire un livello di protezione più perfezionato. Si può progettare un hardware che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione. In alternativa, con bit di protezione distinti per ogni tipo d'accesso, si può ottenere una qualsiasi combinazione di tali tipi d'accesso; i tentativi illegali causano la generazione di un'eccezione per il sistema operativo.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto bit di validità. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un'eccezione. Il sistema operativo concede o impedisce l'accesso a una pagina impostando in modo appropriato tale bit.

Per esempio, supponiamo che in un sistema con uno spazio di indirizzi di 14 bit (da 0 a 16.383) si abbia un programma che deve usare soltanto gli indirizzi da 0 a 10.468. Con una dimensione delle pagine di 2 kb si ha la situazione mostrata nella Figura 9.13. Gli indirizzi nelle pagine 0, 1, 2, 3, 4 e 5 sono tradotti normalmente tramite la tabella delle pagine. D'altra parte, ogni tentativo di generare un indirizzo nelle pagine 6 o 7 trova non valido il bit di validità; in questo caso il calcolatore invia un'eccezione al sistema operativo (riferimento di pagina non valido).



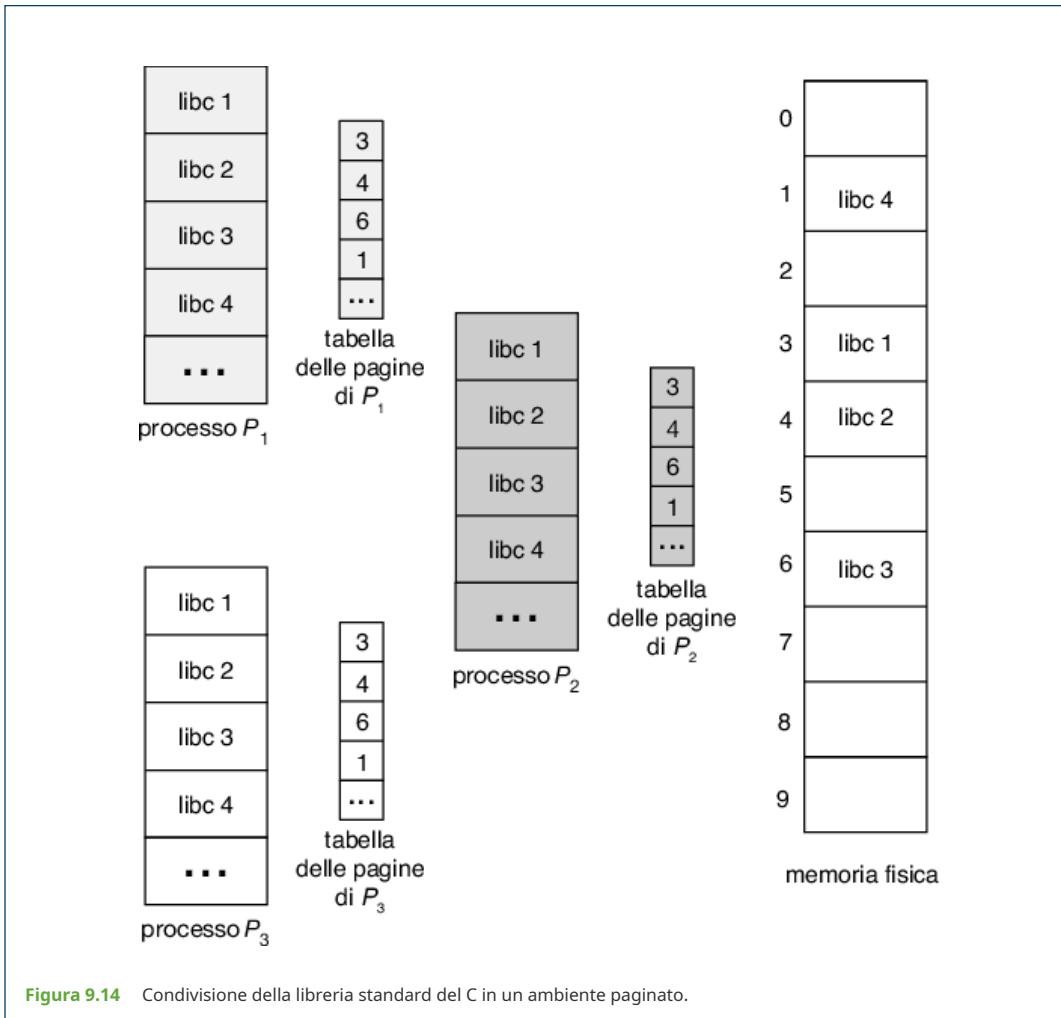
Questo schema ha creato un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288 a 16.383 sono non validi. Tale problema è dovuto alla dimensione delle pagine di 2 kb e corrisponde alla frammentazione interna della paginazione.

Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una gran parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria. Alcune architetture dispongono di registri, detti registri di lunghezza della tabella delle pagine (*page-table length register*, ptlr), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa la generazione di un'eccezione per il sistema operativo.

9.3.4 Pagine condivise

Un vantaggio della paginazione risiede nella possibilità di *condividere* codice comune, il che è particolarmente importante in un ambiente con più processi. Si consideri la libreria standard del C, che fornisce parte dell'interfaccia alla chiamata di sistema in molte versioni di unix e Linux. Su un tipico sistema Linux, la maggior parte dei processi utente richiede la libreria standard del C `libc`. Un'opzione percorribile è che ogni processo carichi la propria copia di `libc` nel proprio spazio di indirizzamento. Se un sistema ha 40 processi utente e la libreria `libc` è di 2 mb, ciò richiederebbe 80 mb di memoria.

Se il codice è rientrante può però essere condiviso, come illustrato nella Figura 9.14, dove si osservano tre processi che condividono le pagine della libreria `libc` (anche se la figura mostra che la libreria `libc` occupa quattro pagine, nella realtà ne occuperebbe di più). Il codice rientrante è un codice non auto-modificante: non cambia mai durante l'esecuzione. Due o più processi possono quindi eseguire lo stesso codice allo stesso tempo. Ogni processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari per la propria esecuzione. I dati per due diversi processi saranno, ovviamente, diversi. Solo una copia della libreria standard del C deve essere conservata nella memoria fisica e la tabella delle pagine di ogni processo utente viene mappata sulla stessa copia fisica di `libc`. Quindi, per supportare 40 processi, abbiamo bisogno di una sola copia della libreria, e lo spazio totale ora richiesto è di 2 mb anziché di 80 mb: un risparmio significativo!



Oltre alle librerie di run-time come `libc`, altri programmi d'uso frequente possono essere condivisi: compilatori, interfacce a finestre, sistemi di database e così via. Le librerie condivise discusse nel Paragrafo 9.1.5 sono in genere implementate con pagine condivise. Per poter essere condiviso, il codice deve essere rientrante. La natura di sola lettura del codice condiviso non deve essere lasciata alla correttezza intrinseca del codice, ma deve essere fatta rispettare dal sistema operativo.

La condivisione della memoria tra processi di un sistema è simile al modo in cui i thread condividono lo spazio d'indirizzi di un task (Capitolo 4). Inoltre con riferimento al Capitolo 3, dove si descrive la memoria condivisa come un metodo di comunicazione tra processi, alcuni sistemi operativi realizzano la memoria condivisa impiegando le pagine condivise.

Oltre a permettere che più processi condividano le stesse pagine fisiche, l'organizzazione della memoria in pagine offre numerosi altri vantaggi; ne vedremo alcuni nel Capitolo 10.

9.4 Struttura della tabella delle pagine

In questo paragrafo si descrivono alcune tra le tecniche più comuni per strutturare la tabella delle pagine: la paginazione gerarchica, la tabella delle pagine di tipo hash e la tabella delle pagine invertita.

9.4.1 Paginazione gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). In un ambiente di questo tipo la stessa tabella delle pagine diventa eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d'indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 kb (2^{12}), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi ($2^{32}/2^{12}$). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 mb di spazio fisico d'indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (Figura 9.15). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 kb. Ciascun indirizzo logico è suddiviso in un numero di pagina di 20 bit e in un offset di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è sua volta suddiviso in un numero di pagina di 10 bit e un offset di pagina di 10 bit. Quindi, l'indirizzo logico è strutturato come segue:

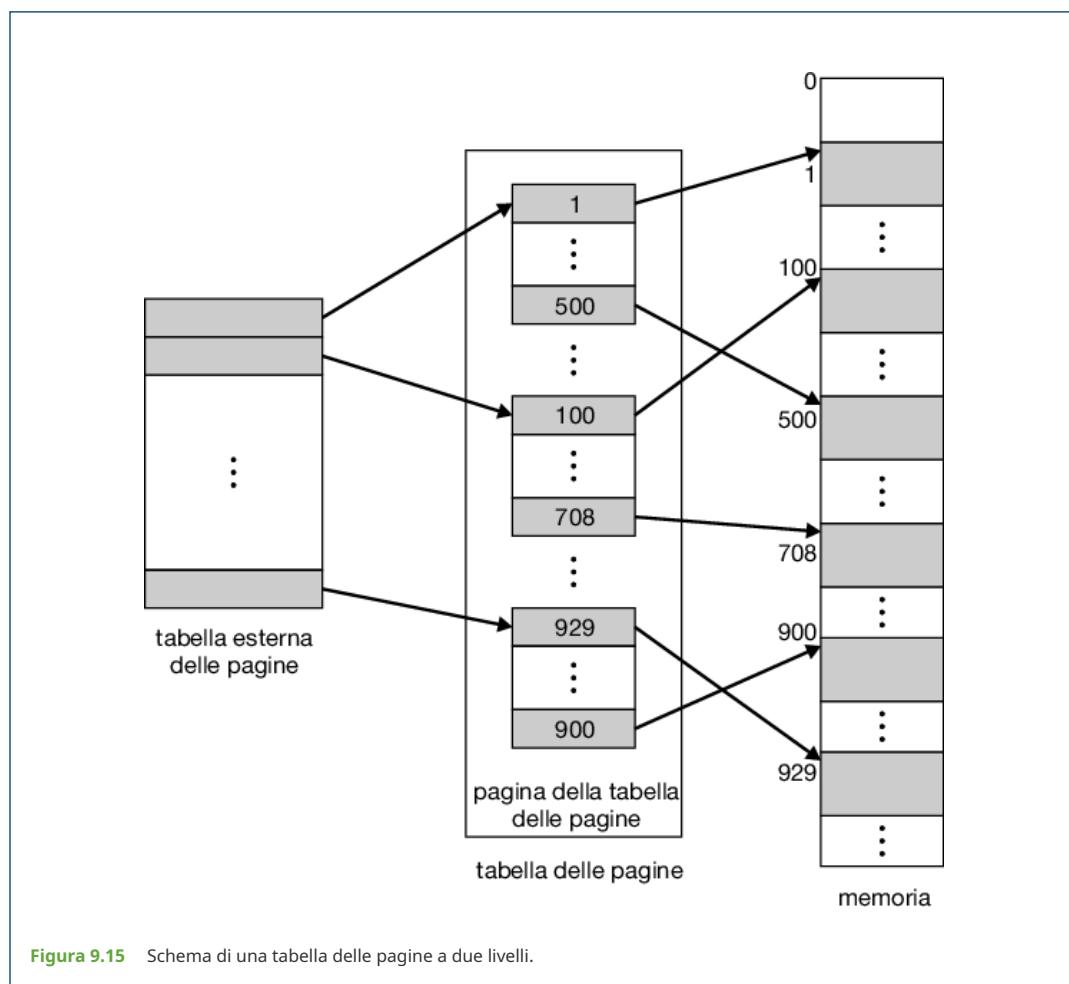
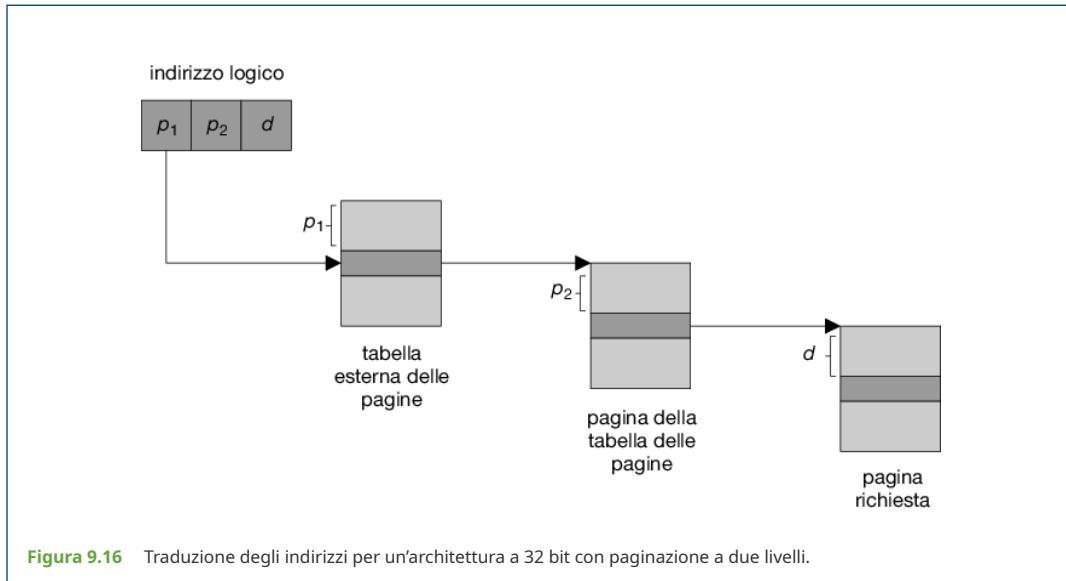


Figura 9.15 Schema di una tabella delle pagine a due livelli.

numero di pagina	offset di pagina
p ₁	p ₂
d	

dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è l'offset all'interno della pagina indicata dalla tabella esterna delle pagine. Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella Figura 9.16. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come tabella delle pagine ad associazione diretta (*forward-mapped page table*).



Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per spiegare questo aspetto, si supponga che la dimensione delle pagine di questo sistema sia di 4 kb (2^{12}). In questo caso, la tabella delle pagine conterrà fino a 2^{52} elementi. Adottando uno schema di paginazione a due livelli, le tabelle interne delle pagine possono occupare convenientemente una pagina, o contenere 2^{10} elementi di 4 byte. Gli indirizzi si presentano come segue:

pagina esterna	pagina interna	offset
p_1	p_2	d
42	10	12

La tabella esterna delle pagine consiste di 2^{42} elementi, o 2^{44} byte. La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcuni processori a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza.

La tabella esterna delle pagine si può suddividere in vari modi. Per esempio, si può paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Si supponga che la tabella esterna delle pagine sia costituita di pagine di dimensione ordinaria (2^{10} elementi, o 2^{12} byte); uno spazio d'indirizzi a 64 bit è ancora enorme:

seconda pagina esterna	pagina esterna	pagina interna	offset
p_1	p_2	p_3	d
32	10	10	12

La tabella esterna delle pagine è ancora di 2^{34} byte (16 gb).

Il passo successivo sarebbe uno schema di paginazione a quattro livelli, in cui si pagina anche la tabella esterna di secondo livello delle pagine, e così via. L'Ultrasparc a 64 bit richiederebbe sette livelli di paginazione – con un numero proibitivo di accessi alla memoria – per tradurre ciascun indirizzo logico. Da questo esempio è possibile capire perché, per le architetture a 64 bit, le page table gerarchiche sono in genere considerate inappropriate.

9.4.2 Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi oltre i 32 bit consiste nell'impiego di una tabella delle pagine di tipo hash, in cui l'argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi: (1) il numero della pagina virtuale; (2) l'indirizzo del frame corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata (Figura 9.17).

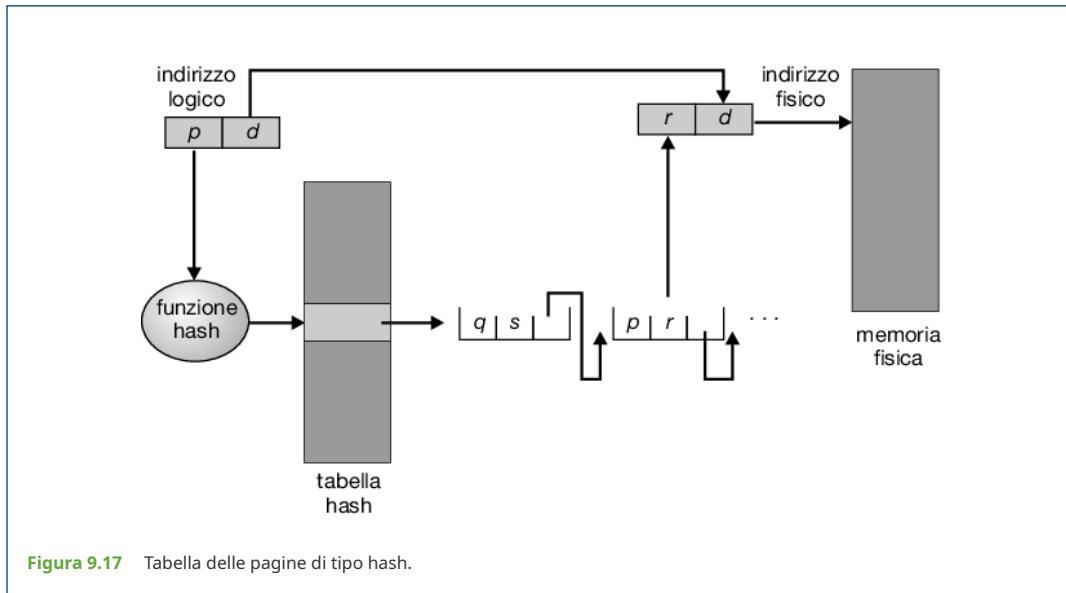


Figura 9.17 Tabella delle pagine di tipo hash.

Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della tabella delle pagine a gruppi (*clustered page table*), simile alla tabella delle pagine di tipo hash; la differenza è che ciascun elemento della tabella hash contiene i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue (per esempio 16). Le tabelle delle pagine a gruppi sono particolarmente utili per gli spazi d'indirizzi sparsi, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi.

9.4.3 Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, ossia vi sono elementi corrispondenti a ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa rappresentazione tabellare è naturale, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è che ciascuna tabella delle pagine può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria solo per sapere com'è impiegata la rimanente memoria fisica.

Per risolvere questo problema si può fare uso della tabella delle pagine invertita. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Nella Figura 9.18 sono mostrate le operazioni di una tabella delle pagine invertita; si confronti questa figura con la Figura 9.8, che illustra il modo di operare per una tabella delle pagine ordinaria. Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio d'indirizzi (Paragrafo 9.3.2) in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi d'indirizzi diversi associati alla memoria fisica; l'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente. Esempi di sistemi che usano le tabelle delle pagine invertite sono l'Ultrasparc a 64 bit e il Powerpc.

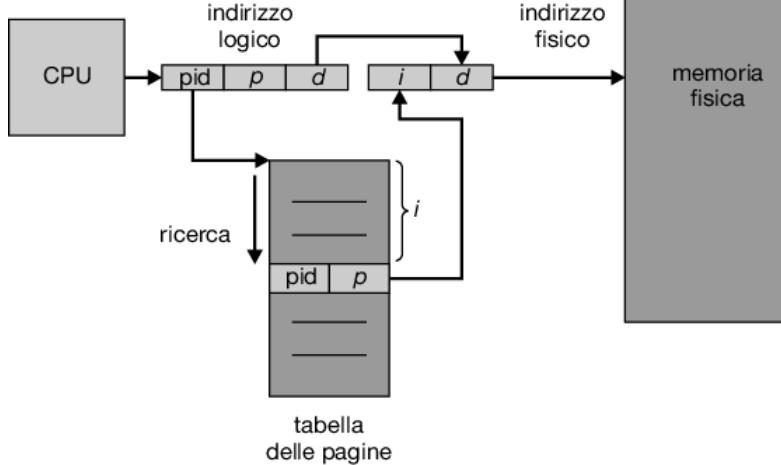


Figura 9.18 Tabella delle pagine invertita.

Per illustrare questo metodo descriviamo una versione semplificata della tabella delle pagine invertita dell'ibm rt. ibm è stata la prima grande azienda a utilizzare le tabelle delle pagine invertite, a cominciare dal System 38, passando per il sistema RS/6000, fino alle attuali cpu ibm Power. Nell'ibm rt ciascun indirizzo virtuale è una tripla del tipo seguente:

<id-processo, numero di pagina, offset>

Ogni elemento della tabella delle pagine invertita è una coppia <id-processo, numero di pagina> dove l'*id-processo* assume il ruolo di identificatore dello spazio d'indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell'indirizzo virtuale, formato da <id-processo, numero di pagina>, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull'elemento *i*, si genera l'indirizzo fisico <*i*, offset>. In caso contrario è stato tentato un accesso a un indirizzo illegale.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza potrebbe essere necessario esaminare tutta la tabella; questa ricerca richiederebbe troppo tempo. Per limitare l'entità del problema si può impiegare una tabella hash (come si descrive nel Paragrafo 9.4.2), che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Ricordiamo che la ricerca si effettua prima nel tlb, quindi si consulta la tabella hash, il che migliore le prestazioni.

Un quesione interessante relativa alle tabelle delle pagine invertite riguarda la memoria condivisa. Con la paginazione standard, ogni processo ha una propria tabella delle pagine, il che consente di mappare più indirizzi virtuali sullo stesso indirizzo fisico. Questo metodo non può essere utilizzato con le tabelle delle pagine invertite. Infatti, poiché esiste una sola pagina virtuale per ogni pagina fisica, una pagina fisica non può avere due (o più) indirizzi virtuali condivisi. Con le tabelle delle pagine invertite, in un dato istante si può dunque avere una sola mappatura di un indirizzo virtuale sull'indirizzo fisico. Un riferimento da parte di un altro processo che condivide la memoria provocherà un errore di pagina e sostituirà la mappatura con un indirizzo virtuale diverso.

9.4.4 Oracle SPARC Solaris

Consideriamo come ultimo esempio una moderna cpu a 64 bit e un sistema operativo strettamente integrati tra loro per fornire una memoria virtuale a basso overhead. Il sistema Solaris in esecuzione sulla cpu sparc è un sistema operativo completamente a 64 bit e come tale deve risolvere il problema della memoria virtuale senza esaurire tutta la sua memoria fisica, mantenendo livelli multipli di tabelle delle pagine. L'approccio di Solaris è piuttosto complesso, ma risolve il problema in modo efficiente utilizzando tabelle delle pagine di tipo hash. Vi sono due tabelle hash, una per il kernel e una per tutti i processi utente. Ogni tabella mappa indirizzi di memoria virtuale nella memoria fisica. Ogni elemento della tabella hash rappresenta un'area contigua di memoria virtuale mappata, il che è più efficiente rispetto ad avere voci separate per ciascuna pagina. Ogni voce ha un indirizzo di base e un intervallo (*span*) che indica il numero di pagine rappresentate da quella voce.

La traduzione da virtuale a fisico impiegherebbe troppo tempo se ogni indirizzo richiedesse la ricerca attraverso una tabella hash, quindi la cpu implementa un tlb che contiene le voci della tabella di traduzione (tte), in modo da offrire ricerche hardware rapide. Una

cache di queste tte risiede in un buffer di memoria di traduzione (tsb), che include una voce per ogni pagina di recente accesso. In caso di riferimento a un indirizzo virtuale, l'hardware interroga il tlb per una traduzione. Se non vengono trovate traduzioni, l'hardware scorre il buffer tsb in cerca della tte che corrisponde all'indirizzo virtuale che ha causato la ricerca. Questa funzionalità, chiamata tlb walk, è presente su molte cpu moderne. Se viene trovata una corrispondenza nel tsb, la cpu copia la voce tsb nel tlb, e la traduzione viene completata. Se invece non viene trovata alcuna corrispondenza, viene interrotto il kernel per effettuare una ricerca nella tabella hash. Il kernel crea quindi una tte dalla tabella hash appropriata e la memorizza nel tsb affinché l'unità di gestione della memoria della cpu possa effettuare il caricamento automatico nel tlb. Infine, il gestore di interrupt restituisce il controllo alla mmu, che completa la conversione dell'indirizzo e recupera il byte o la parola richiesta dalla memoria principale.

9.5 Avvicendamento dei processi (swapping)

Le istruzioni di un processo e i dati su cui operano per essere eseguiti devono essere in memoria. Tuttavia, un processo, o una sua parte, può essere rimosso temporaneamente dalla memoria centrale (mediante un'operazione detta di *swap out*, scaricamento), spostato in una memoria ausiliaria (*backing store*) e in seguito riportato in memoria (*swap in*, caricamento) per continuare la sua esecuzione (Figura 9.19). Grazie a questo procedimento, detto avvicendamento dei processi o swapping, lo spazio totale degli indirizzi fisici di tutti i processi può eccedere la reale dimensione della memoria fisica del sistema, aumentando così il grado di multiprogrammazione possibile.

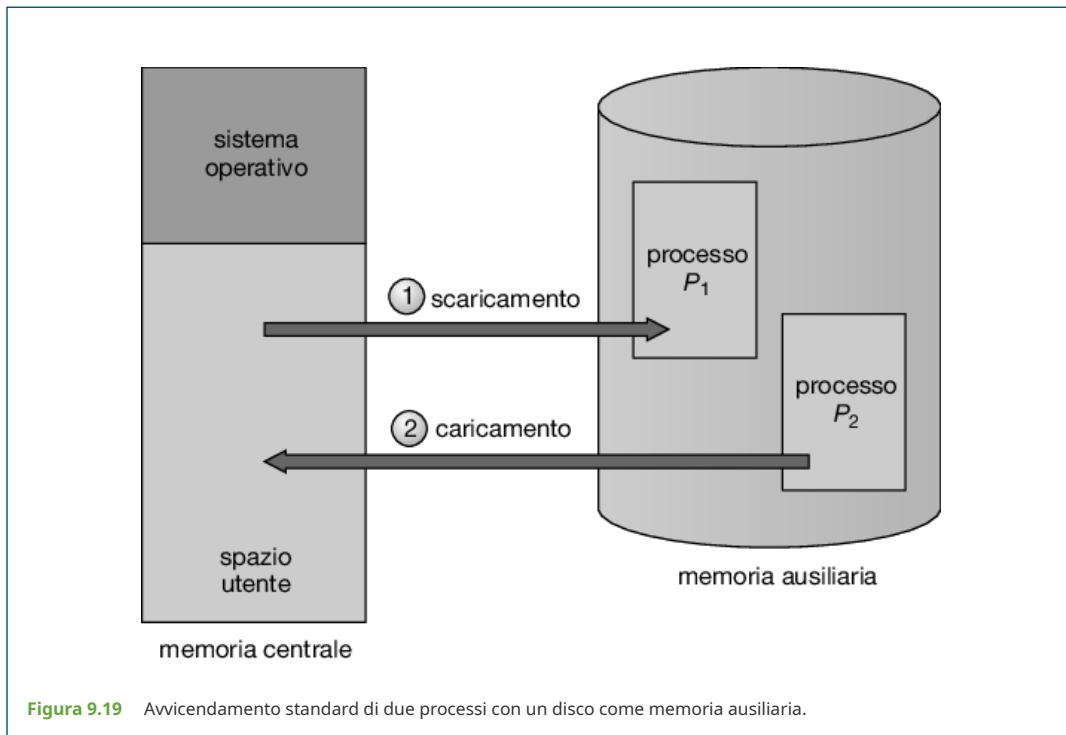


Figura 9.19 Avvicendamento standard di due processi con un disco come memoria ausiliaria.

9.5.1 Avvicendamento standard

L'avvicendamento standard riguarda lo spostamento di interi processi tra la memoria centrale e una memoria ausiliaria solitamente costituita da un veloce dispositivo di memorizzazione secondaria. Tale memoria deve essere abbastanza ampia da contenere tutte le porzioni dei processi che devono essere memorizzate e recuperate, e deve fornire accesso diretto alle immagini dei processi memorizzati. Quando un processo, o una sua parte, viene spostato nella memoria ausiliaria, devono essere scritte in tale memoria anche le strutture dati associate al processo. In un processo multithread devono essere spostate anche tutte le strutture dati relative a ogni thread. Il sistema operativo deve inoltre conservare i metadati relativi ai processi che sono stati spostati, in modo da poterli ripristinare quando i processi vengono reinseriti nella memoria.

Il vantaggio dell'avvicendamento standard è che consente di sovrascrivere la memoria fisica, in modo che il sistema possa ospitare più processi rispetto alla quantità di memoria fisica effettivamente a disposizione. I processi inattivi o raramente attivi sono buoni candidati per l'avvicendamento e la memoria allocata a questi processi inattivi può quindi essere destinata ai processi attivi. Se un processo inattivo che è stato rimosso dalla memoria centrale diventa di nuovo attivo, deve essere riportato in memoria e ripristinato. Questo procedimento è illustrato nella Figura 9.19.

9.5.2 Avvicendamento con paginazione

L'avvicendamento standard è stato utilizzato nei sistemi Unix tradizionali, ma in genere non è più utilizzato nei sistemi operativi contemporanei, poiché la quantità di tempo necessaria per spostare interi processi tra la memoria centrale e la memoria ausiliaria è proibitiva (un'eccezione è costituita da Solaris, che usa ancora l'avvicendamento standard, anche se solo in circostanze estreme in cui la memoria disponibile è estremamente bassa).

La maggior parte dei sistemi, inclusi Linux e Windows, usa ora una variante dell'avvicendamento standard in cui è possibile spostare solo alcune pagine di un processo, piuttosto che l'intero processo. Questa strategia consente tuttavia di sovrascrivere la memoria

fisica, ma non comporta il costo dello spostamento di interi processi, poiché presumibilmente solo un numero limitato di pagine sarà coinvolto nello scambio. Al giorno d'oggi, generalmente, si utilizza il termine avvicendamento (o swapping) per riferirsi all'avvicendamento standard, mentre quando si parla di paginazione (o paging) si fa riferimento all'avvicendamento con paginazione. Un'operazione di *page out* (scaricamento della pagina) sposta una pagina dalla memoria centrale alla memoria ausiliaria, mentre il processo inverso è noto come *page in* (caricamento della pagina). L'avvicendamento con paginazione è illustrato nella Figura 9.20, dove su un sottoinsieme delle pagine dei processi A e B viene effettuata un'operazione, rispettivamente, di page out e page in. Come vedremo nel Capitolo 10, l'avvicendamento con paginazione funziona bene in abbinamento alla memoria virtuale.

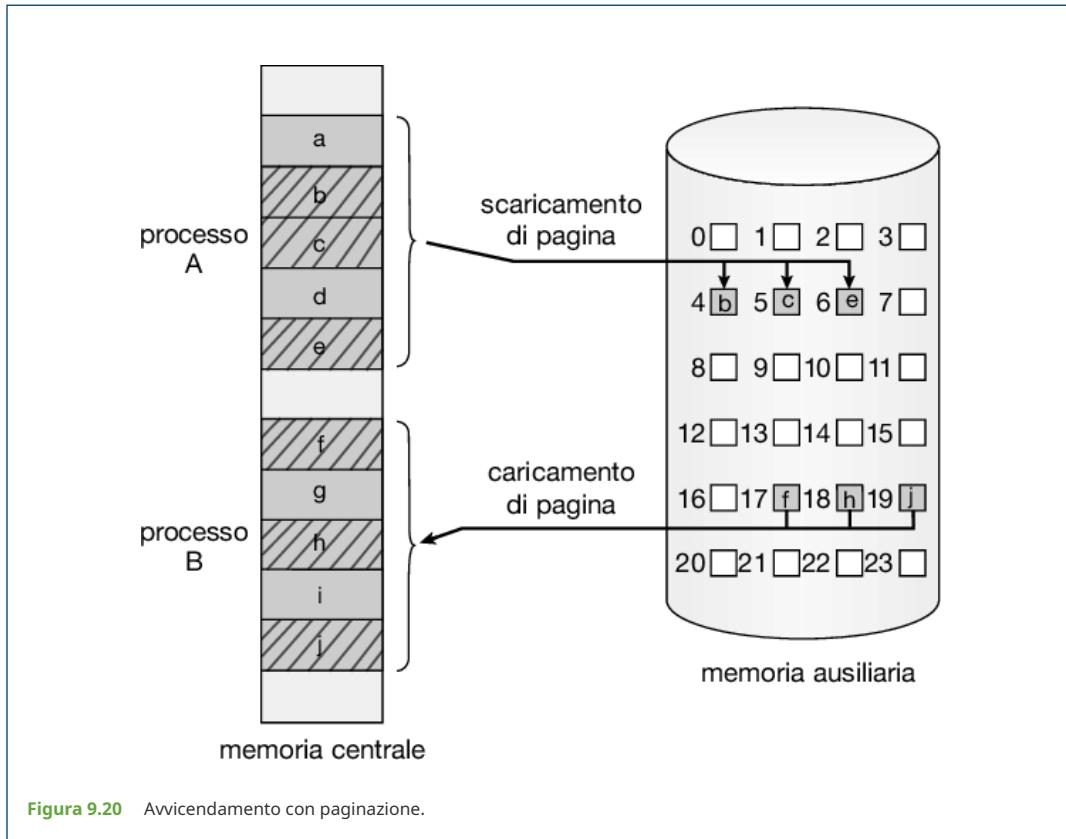


Figura 9.20 Avvicendamento con paginazione.

9.5.3 Avvicendamento di processi nei sistemi mobili

La maggior parte dei sistemi operativi per pc e server supporta l'avvicendamento con paginazione. Al contrario, i sistemi mobili in genere non supportano alcuna forma di avvicendamento. I dispositivi mobili utilizzano infatti solitamente la memoria flash per la memorizzazione non volatile, anziché i più capienti dischi rigidi: il vincolo di spazio che ne risulta è uno dei motivi per cui i progettisti di sistemi operativi mobili evitano l'avvicendamento. Altri motivi includono il numero limitato di scritture che la memoria flash può tollerare prima di diventare inaffidabile e la scarsa velocità di trasferimento tra la memoria principale e la memoria flash su quel tipo di dispositivo.

Invece di usare l'avvicendamento dei processi, se la memoria disponibile scende al di sotto di una certa soglia, ios di Apple *chiede* alle applicazioni di rinunciare volontariamente alla memoria allocata. I dati di sola lettura (come il codice) vengono rimossi dal sistema e successivamente ricaricati dalla memoria flash, se necessario. I dati che sono stati modificati (per esempio lo stack) non vengono rimossi. Tuttavia, tutte le applicazioni che non riescono a liberare memoria a sufficienza possono essere terminate dal sistema operativo.

Android non supporta l'avvicendamento e adotta una strategia simile a quella utilizzata da ios. Anche Android può terminare un processo qualora la memoria libera disponibile non sia sufficiente. Tuttavia, prima di terminarlo, scrive lo stato dell'applicazione (*application state*) nella memoria flash, in modo che il processo possa essere rapidamente riavviato.

A causa di queste limitazioni, gli sviluppatori per dispositivi mobili devono allocare e rilasciare memoria con molta cura, in modo da garantire che le loro applicazioni non utilizzino troppa memoria e non soffrano di *memory leak* ("perdite di memoria").

Anche se l'avvicendamento di pagine è più efficiente dell'avvicendamento di interi processi, quando un sistema è sottoposto a qualsiasi forma di avvicendamento è spesso segno che ci sono più processi attivi rispetto alla memoria fisica disponibile. Esistono generalmente due approcci per gestire questa situazione: (1) terminare alcuni processi o (2) acquistare più memoria fisica!

9.6 Esempio: le architetture Intel a 32 e 64 bit

L'architettura Intel ha dominato il mondo dei personal computer per diversi anni. Il processore Intel 8086, a 16 bit, apparve alla fine degli anni '70 e fu presto seguito da un altro chip a 16 bit, l'Intel 8088, noto per essere stato il chip utilizzato nel pc ibm originale. Sia il chip 8086 che il chip 8088 erano basati su una architettura segmentata. Più tardi Intel iniziò la produzione di una serie di chip a 32 bit, ia-32, che includeva la famiglia di processori Pentium. L'architettura ia-32 supportava paginazione e segmentazione. Più di recente Intel ha prodotto una serie di chip 64 bit basati sull'architettura x86-64. Attualmente tutti i più popolari sistemi operativi per pc, tra cui Windows, macos e Linux (anche se Linux, ovviamente, gira su diverse altre architetture), vengono eseguiti su chip Intel. Va notato tuttavia che la posizione dominante di Intel non si è propagata ai sistemi mobili, su cui attualmente gode di notevole successo l'architettura arm (si veda il Paragrafo 9.7).

In questo paragrafo esaminiamo la traduzione degli indirizzi nelle architetture ia-32 e x86-64. Prima di procedere è importante osservare che, poiché Intel ha rilasciato diverse versioni e diverse varianti delle sue architetture nel corso degli anni, non possiamo fornire una descrizione completa della struttura di gestione della memoria per tutti i suoi chip. Non è nemmeno nostra intenzione fornire tutti i dettagli della cpu, perché questi argomenti vengono trattati nei libri di architettura degli elaboratori. Quello che faremo, piuttosto, è di presentare i principali concetti della gestione della memoria di queste cpu Intel.

9.6.1 Architettura IA-32

La gestione della memoria nei sistemi ia-32 è suddivisa in due componenti: segmentazione e paginazione. La cpu genera indirizzi logici che vengono passati all'unità di segmentazione. L'unità di segmentazione produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare viene quindi passato all'unità di paginazione, che genera l'indirizzo fisico nella memoria principale. Dunque, l'unità di segmentazione e l'unità di paginazione formano insieme l'equivalente dell'unità di gestione della memoria (mmu). Questo schema è mostrato nella Figura 9.21.



Figura 9.21 Traduzione degli indirizzi logici in indirizzi fisici in IA-32.

9.6.1.1 Segmentazione in IA-32

Nell'architettura ia-32 un segmento può raggiungere la dimensione massima di 4 gb; il numero massimo di segmenti per processo è pari a 16 k. Lo spazio degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8 k segmenti riservati al processo; la seconda contiene fino a 8 k segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella tabella locale dei descrittori (*local descriptor table*, ldt), quelle relative alla seconda partizione sono memorizzate nella tabella globale dei descrittori (*global descriptor table*, gdt). Ciascun elemento nella ldt e nella gdt è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite.

Un indirizzo logico è una coppia (*selettore, offset*), dove il selettore è un numero di 16 bit:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

in cui *s* indica il numero del segmento, *g* indica se il segmento si trova nella gdt o nella ldt e *p* contiene informazioni relative alla protezione. L'*offset* è un numero di 32 bit che indica la posizione del byte (o della parola) all'interno del segmento in questione.

La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di micropogramma di 8 byte per i corrispondenti descrittori prelevati dalla ldt o dalla gdt. Questa cache evita alla macchina di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

Un indirizzo lineare di ia-32 è lungo 32 bit e si genera come segue. Il registro di segmento punta all'elemento appropriato all'interno della ldt o della gdt; le informazioni relative alla base e al limite di tale segmento si usano per generare un indirizzo lineare. Innanzitutto si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa la generazione di un'eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dell'offset al valore della base, ottenendo un indirizzo lineare di 32 bit. La Figura 9.22 illustra tale processo. Nel paragrafo successivo si considera come l'unità di paginazione trasforma questo indirizzo lineare in un indirizzo fisico.

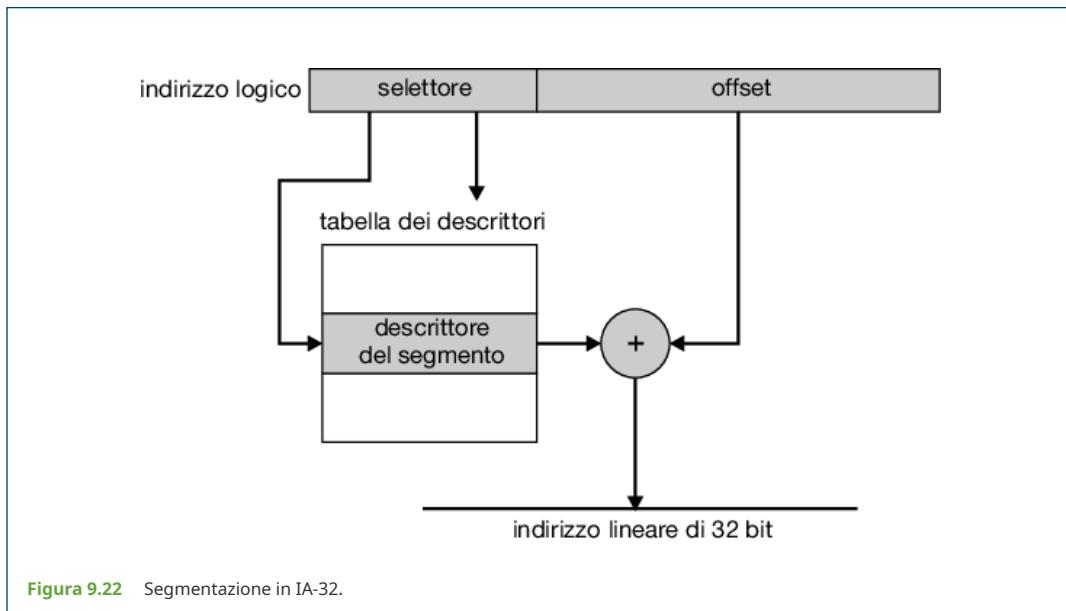
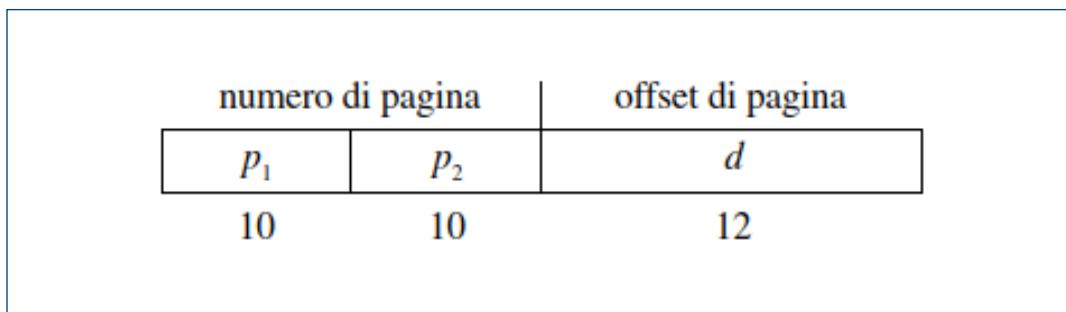


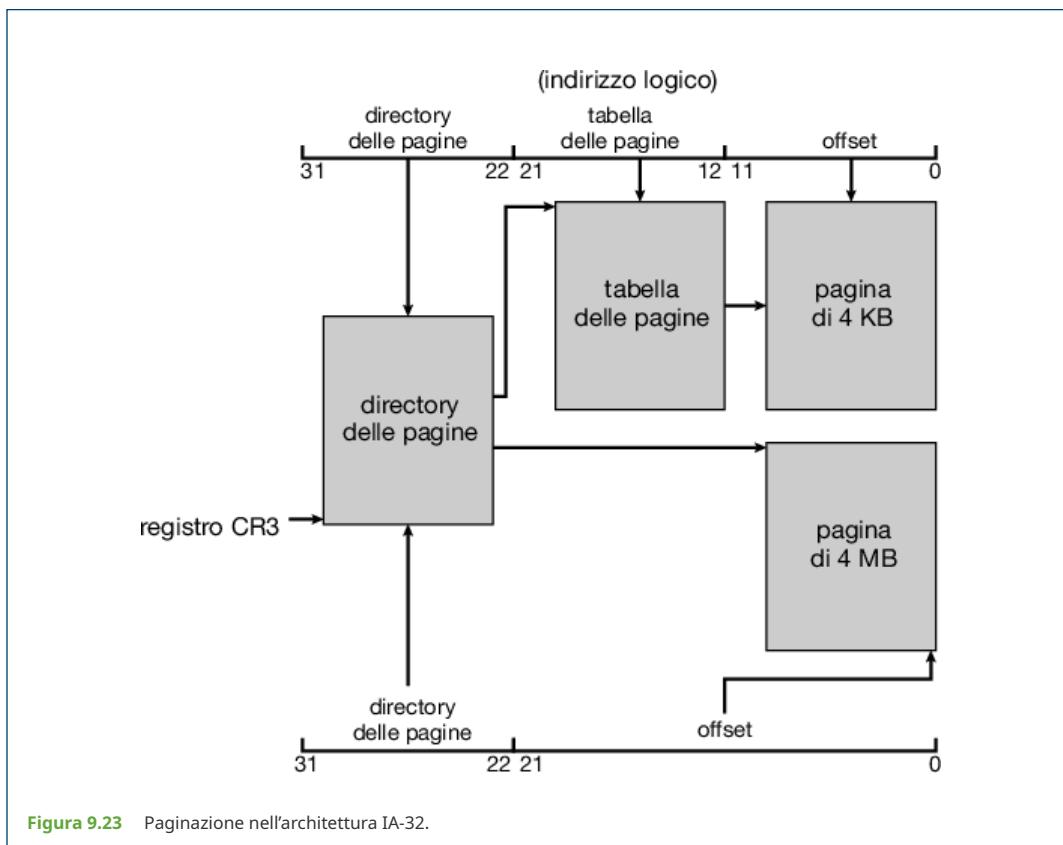
Figura 9.22 Segmentazione in IA-32.

9.6.1.2 Paginazione in IA-32

L'architettura ia-32 prevede che le pagine abbiano una misura di 4 kb oppure di 4 mb. Per le pagine di 4 kb, in ia-32 vige uno schema di paginazione a due livelli che prevede la seguente scomposizione degli indirizzi lineari a 32 bit:



Lo schema di traduzione degli indirizzi per questa architettura, simile a quello rappresentato nella Figura 9.16, è mostrato in dettaglio nella Figura 9.23. I dieci bit più significativi puntano a un elemento nella tabella delle pagine più esterna, detta in ia-32 directory delle pagine. (Il registro cr3 punta alla directory delle pagine del processo corrente.) Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono l'offset da applicare all'interno della pagina di 4 kb cui si fa riferimento nella tabella delle pagine.



Un elemento appartenente alla directory delle pagine è il flag `Page_Size`; se impostato, indica che il frame non ha la dimensione standard di 4 mb, ma misura invece 4 kb. In questo caso, la directory di pagina punta direttamente al frame di 4 mb, scavalcando la tabella delle pagine interna; i 22 bit meno significativi nell'indirizzo lineare indicano l'offset nella pagina di 4 mb.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine in ia-32 possono essere trasferite sul disco. In questo caso, si ricorre a un bit *invalid* in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Se è su disco, il sistema operativo può usare i 31 bit rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

Non appena gli sviluppatori di software hanno iniziato a soffrire della limitazione della memoria a 4 gb imposta dall'architettura a 32 bit, Intel ha introdotto l'estensione di indirizzo della pagina (pae, *page address extension*), che consente ai processori a 32 bit di accedere a uno spazio di indirizzamento fisico più grande di 4 gb. La differenza fondamentale introdotta dal supporto pae era il passaggio della paginazione da una schema a due livelli (come mostrato nella Figura 9.23) a uno schema a tre livelli, in cui i primi due bit fanno riferimento a una tabella di puntatori alle directory di pagina. La Figura 9.24 illustra un sistema pae con pagine di 4 kb. pae supporta anche pagine di 2 mb.

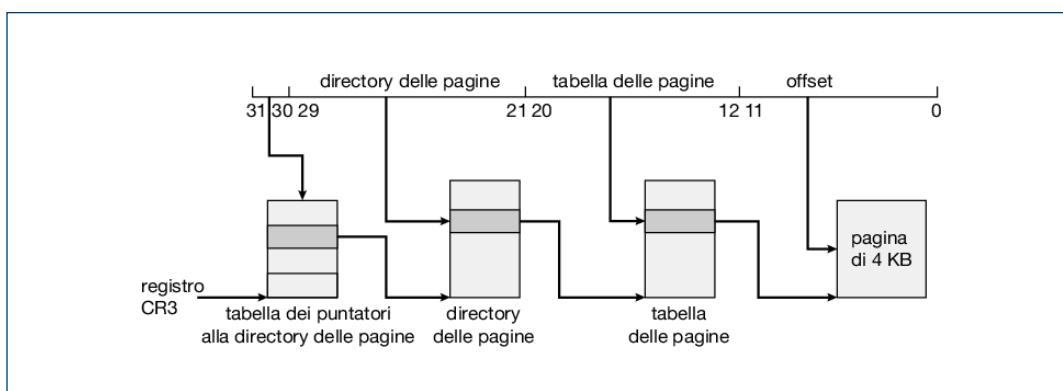


Figura 9.24 Estensione degli indirizzi di pagina.

Con pae è stata inoltre aumentata la dimensione degli elementi della directory delle pagine e della tabella delle pagine, che passa da 32 a 64 bit, permettendo di estendere l'indirizzo di base delle tabelle delle pagine e dei frame da 20 a 24 bit. In combinazione con i 12 bit di offset, l'aggiunta del supporto pae a ia-32 ha aumentato lo spazio di indirizzamento a 36 bit, garantendo il supporto di un massimo di 64 gb di memoria fisica. È importante notare che per utilizzare pae è necessario il supporto del sistema operativo. Linux e macos supportano pae. Tuttavia, le versioni a 32 bit dei sistemi operativi Windows per desktop supportano soltanto 4 gb di memoria fisica, anche se pae è abilitato.

9.6.2 Architettura x86-64

Lo sviluppo di architetture Intel a 64 bit ha avuto una storia curiosa. La prima di queste architetture era ia-64 (in seguito denominata Itanium), ma questa architettura non ha avuto un'ampia diffusione. Nel frattempo, un altro produttore di chip – amd – ha iniziato a sviluppare un'architettura a 64 bit nota come x86-64, basata sull'estensione del set di istruzioni ia-32 esistente. L'architettura x86-64 supportava spazi di indirizzamento logico e fisico molto più grandi e introduceva diverse altre novità architettoniche. Storicamente amd aveva spesso sviluppato chip basati sull'architettura Intel, ma in questo caso i ruoli si sono invertiti e Intel ha adottato l'architettura x86-64 di amd. Nel discutere questa architettura, piuttosto che utilizzare le denominazioni commerciali amd64 e Intel 64, useremo il termine più generale x86-64.

Il supporto a uno spazio di indirizzamento a 64 bit permette di indirizzare la straordinaria quantità di 2^{64} byte di memoria – un numero superiore a 16 miliardi di miliardi (o 16 exabyte). Tuttavia, anche se i sistemi a 64 bit possono potenzialmente indirizzare una tale quantità di memoria, nella pratica vengono utilizzati nei progetti attuali assai meno di 64 bit per la rappresentazione di un indirizzo. L'architettura x86-64 utilizza attualmente un indirizzo virtuale di 48 bit con supporto ai formati di pagina di 4 kb, 2 mb o 1 gb utilizzando una paginazione a quattro livelli. La rappresentazione dell'indirizzo lineare è mostrata nella Figura 9.25. Poiché questo schema di indirizzamento può usare pae, gli indirizzi virtuali sono di 48 bit, ma supportano indirizzi fisici a 52 bit (4096 terabyte).

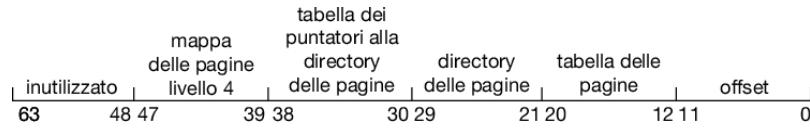


Figura 9.25 Indirizzo lineare in x86-64.

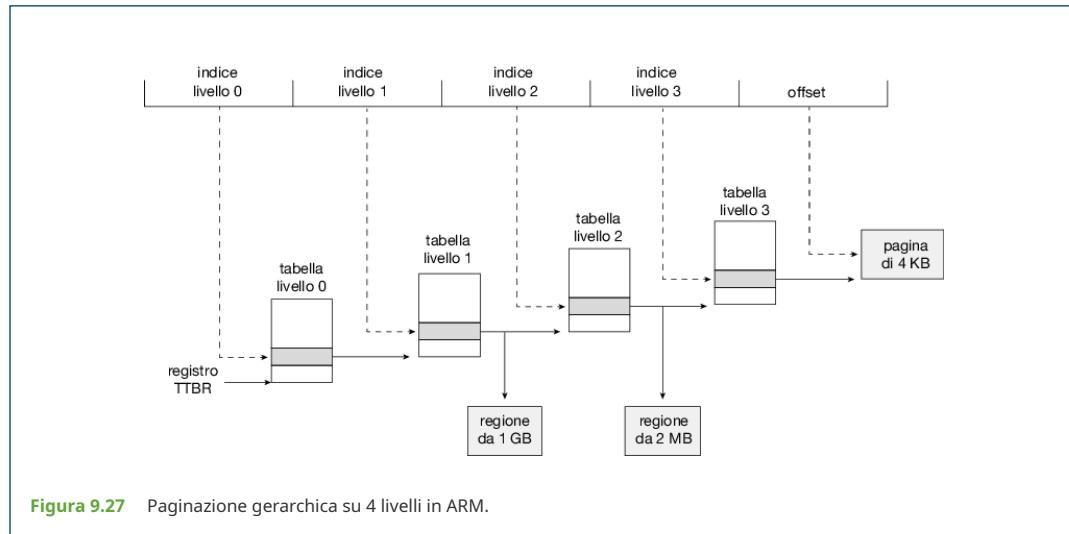
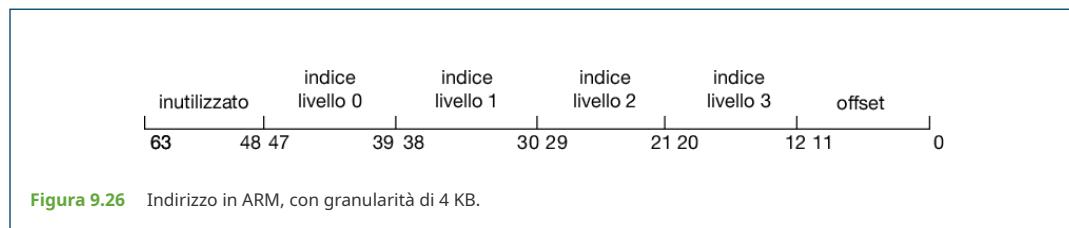
9.7 Esempio: architettura ARMv8

Anche se i chip Intel hanno dominato il mercato dei personal computer per oltre 30 anni, i dispositivi mobili come smartphone e tablet montano spesso processori arm a 32 bit. È interessante notare che mentre Intel progetta e produce i chip, arm li progetta soltanto, concedendo poi in licenza i suoi progetti ai produttori di chip. Apple ha adottato arm per i suoi dispositivi mobili iPhone e iPad e anche diversi smartphone basati su Android utilizzano processori arm. Oltre ai dispositivi mobili, arm progetta architetture per sistemi embedded real-time. Grazie alla grande varietà di dispositivi che utilizzano un'architettura arm, sono stati prodotti oltre 100 miliardi di processori arm, rendendo così questa architettura la più utilizzata in termini della quantità di chip prodotti. In questo paragrafo descriviamo l'architettura armv8 a 64 bit.

armv8 supporta tre diverse granularità di traduzione (*translation granule*): 4 kb, 16 kb e 64 kb. Per ogni diversa granularità sono fornite diverse dimensioni di pagina, oltre a sezioni più grandi di memoria contigua, note come regioni. Di seguito sono riportate le dimensioni di pagina e di regione per le diverse granularità di traduzione:

Granularità	Dimensione della pagina	Dimensione della regione
4 kb	4 kb	2 mb, 1 gb
16 kb	16 kb	32 mb
64 kb	64 kb	512 mb

Con granularità di 4 kb e 16 kb è possibile utilizzare fino a quattro livelli di paginazione, mentre con granularità di 64 kb sono possibili fino a tre livelli di paginazione. La Figura 9.26 illustra la struttura degli indirizzi armv8 con una granularità di 4 kb, con un massimo di quattro livelli di paginazione. Si noti che, sebbene armv8 sia un'architettura a 64 bit, vengono attualmente utilizzati solo 48 bit. La struttura di paginazione gerarchica a quattro livelli per una granularità di traduzione di 4 kb è illustrata nella Figura 9.27, dove il registro ttbr è il registro di base della tabella di traduzione e punta alla tabella del livello 0 per il thread corrente.



Se sono utilizzati tutti e quattro i livelli, l'offset (bit 0-11 nella Figura 9.26) si riferisce a una pagina di 4 kb. Tuttavia, si noti che gli elementi della tabella nei livelli 1 e 2 possono fare riferimento a un'altra tabella o a una regione da 1 gb (per la tabella livello 1) o da 2

mb (per la tabella livello 2). Per esempio, se la tabella di livello 1 fa riferimento a una regione da 1 gb anziché a una tabella di livello 2, i 30 bit meno significativi (bit 0-29 nella Figura 9.26) vengono utilizzati come offset per questa regione. Allo stesso modo, se la tabella di livello 2 fa riferimento a una regione da 2 mb anziché a una tabella di livello 3, i 21 bit meno significativi (bit 0-20 nella Figura 9.26) fanno riferimento all'offset all'interno di questa regione da 2 mb.

L'architettura arm supporta inoltre due livelli di tlb. A livello interno vi sono due micro tlb separati, uno per i dati e uno per le istruzioni. Il micro tlb supporta gli asid. A livello esterno vi è un unico tlb principale. La traduzione di un indirizzo inizia a livello micro tlb: in caso di insuccesso viene controllato il tlb principale. In caso di ulteriore insuccesso, ci si rivolge, via hardware, alla tabella delle pagine.

ELABORAZIONE A 64-BIT

La storia ci insegna che anche quando la capacità di memoria, la velocità della cpu e altre caratteristiche simili sembrano sufficienti a soddisfare la domanda futura, il progresso tecnologico riesce alla fine ad assorbire tutte le risorse disponibili. Ci si ritrova dunque ad aver bisogno di più memoria o di una maggior capacità di elaborazione e spesso ciò avviene prima del previsto. Che cosa ci porteranno le future tecnologie per far sembrare troppo piccoli gli indirizzi di 64 bit?

9.8 Sommario

- La memoria centrale è fondamentale per il funzionamento di un moderno sistema elaborativo e consiste in una lungo vettore di byte, ciascuno con il proprio indirizzo.
- Un modo per allocare uno spazio di indirizzamento a ciascun processo è attraverso l'uso dei registri base e limite. Il registro base contiene il più piccolo indirizzo di memoria fisica consentito e il limite specifica la dimensione dell'intervallo.
- La mappatura dei riferimenti a indirizzi simbolici sugli indirizzi fisici effettivi può essere realizzata durante (1) la compilazione, (2) il caricamento, o (3) al tempo d'esecuzione.
- Un indirizzo generato dalla cpu è noto come indirizzo logico. L'unità di gestione della memoria (mmu) lo traduce in un indirizzo fisico in memoria.
- Un approccio per allocare memoria è assegnare partizioni di memoria contigua di dimensione variabile. Queste partizioni possono essere allocate sulla base di tre possibili strategie: (1) first-fit, (2) best-fit e (3) worst-fit.
- I moderni sistemi operativi utilizzano la paginazione per gestire la memoria: la memoria fisica viene suddivisa in blocchi di dimensioni fisse chiamati frame e la memoria logica in blocchi della stessa dimensione chiamati pagine.
- Quando si utilizza la paginazione, un indirizzo logico è diviso in due parti: un numero di pagina e un'offset di pagina. Il numero di pagina funge da indice nella tabella delle pagine di un processo, la quale contiene il frame nella memoria fisica che contiene la pagina. L'offset è la posizione specifica nel frame a cui si fa riferimento.
- Il tlb (translation look-aside buffer) è una cache hardware per la tabella delle pagine. Ogni elemento del tlb contiene un numero di pagina e il frame corrispondente.
- L'utilizzo di un tlb nella conversione degli indirizzi nei sistemi con paginazione comporta l'ottenimento del numero di pagina dall'indirizzo logico e il controllo della presenza del frame associato alla pagina nel tlb. Se il frame è presente può essere ottenuto dal tlb, in caso contrario deve essere recuperato dalla tabella delle pagine.
- La paginazione gerarchica implica una suddivisione di un indirizzo logico in cui ciascuna parte dell'indirizzo fa riferimento a un livello distinto di tabelle delle pagine. Il numero dei livelli gerarchici può crescere all'aumentare della dimensione degli indirizzi oltre i 32 bit. Due strategie che risolvono questo problema sono le tabelle delle pagine di tipo hash e le tabelle delle pagine invertite.
- L'avvicendamento consente al sistema di spostare su disco le pagine appartenenti a un processo per aumentare il livello di multiprogrammazione.
- L'architettura Intel a 32 bit ha due livelli di tabelle delle pagine e supporta pagine di 4 kb o 4 mb. Questa architettura supporta anche l'estensione di indirizzo della pagina, che consente ai processori a 32 bit di accedere a uno spazio di indirizzamento fisico superiore a 4 gb. Le architetture x86-64 e armv8 sono architetture a 64 bit che utilizzano la paginazione gerarchica.

Esercizi di ripasso

9.1 Citate due differenze tra indirizzi logici e fisici.

9.2 Perché la dimensione delle pagine è sempre una potenza di due?

9.3 Considerate un sistema nel quale un programma possa essere separato in due parti: codice e dati. Il processore sa se necessita di un'istruzione (prelievo di istruzione) o di un dato (prelievo o memorizzazione di dati). Perciò, vengono fornite due coppie di registri base e limite: una per le istruzioni e una per i dati. La coppia di registri base e limite per le istruzioni è automaticamente a sola lettura, di modo che i programmi possano essere condivisi tra i diversi utenti. Discutete i vantaggi e gli svantaggi di questo schema.

9.4 Considerate uno spazio degli indirizzi logici di 64 pagine, ciascuna delle quali di 1024 parole, mappato su una memoria fisica di 32 frame.

a. Quanti bit ci sono nell'indirizzo logico?

b. Quanti bit ci sono nell'indirizzo fisico?

9.5 Quale effetto si verifica se si permette a due voci di una tabella delle pagine di puntare allo stesso frame di pagina della memoria? Spiegate come questo effetto potrebbe essere utilizzato per diminuire il tempo necessario per copiare una grande quantità di memoria da uno spazio a un altro. Nel caso in cui vengano aggiornati alcuni byte della prima pagina, quale effetto si avrebbe sulla seconda pagina?

9.6 Date sei partizioni di memoria, pari a 300 kb, 600 kb, 350 kb, 200 kb, 750 kb e 125 kb, nell'ordine, e cinque processi di 115 kb, 500 kb, 358 kb, 200 kb e 375 kb, nell'ordine, come verrebbero allocati in memoria tali processi dagli algoritmi first-fit, best-fit e worst-fit?

9.7 Assumendo che la dimensione della pagina sia di 1 kb, quali sono i numeri di pagina e gli scostamenti per i seguenti indirizzi (indicati in numeri decimali):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

9.8 Il sistema operativo btv ha indirizzi virtuali a 21 bit, anche se su alcuni dispositivi embedded ha indirizzi fisici di solo 16 bit. La dimensione di pagina è di 2 kb. Quante voci ci sono in ognuna delle seguenti strutture dati?

a. Una tabella delle pagine convenzionale, a singolo livello.

b. Una tabella delle pagine invertita.

9.9 Considerate uno spazio di indirizzo logico di 256 pagine, con una dimensione di pagina di 4 kb, mappato su una memoria fisica di 64 frame.

a. Quanti bit sono necessari all'indirizzo logico?

b. Quanti bit sono necessari all'indirizzo fisico?

9.10 Prendete in considerazione un sistema con un indirizzo logico di 32 bit e una dimensione di pagina di 4 kb. Il sistema supporta fino a 512 mb di memoria fisica. Quante voci ci sono in:

a. una tabella delle pagine convenzionale a singolo livello;

b. una tabella delle pagine invertita.

Esercizi

9.11 Spiegate la differenza tra frammentazione interna e frammentazione esterna.

9.12 Considerate il seguente ciclo di produzione di codice binario eseguibile. Si usa un compilatore per generare il codice oggetto dei singoli moduli, e un editor per gestire la fase di link, ossia per combinare diversi moduli oggetto in un unico codice binario eseguibile. Come può l'editor modificare l'associazione di istruzioni e dati, agli indirizzi di memoria? Quali informazioni devono passare dal compilatore all'editor per facilitare l'editor nell'esecuzione di tale associazione?

9.13 Date sei partizioni di memoria, pari a 100 mb, 170 mb, 40 mb, 205 mb, 300 mb e 185 mb, nell'ordine, e cinque processi di 200 mb, 15 mb, 185 mb, 75 mb, 175 mb e 80 mb, nell'ordine, come verrebbero allocati in memoria tali processi dagli algoritmi first-fit, best-fit e worst-fit? Indicate, eventualmente, quali richieste non possano essere soddisfatte. Commentate l'efficienza con cui ciascuno degli algoritmi gestisce la memoria.

9.14 La maggioranza dei sistemi consente ai programmi di aumentare durante l'esecuzione la memoria allocata al proprio spazio di indirizzi. I dati posti nei segmenti heap dei programmi ne rappresentano un esempio. Di che cosa c'è bisogno per agevolare l'allocazione dinamica della memoria, in ognuno dei seguenti casi?

- a. Allocazione contigua della memoria.
- b. Segmentazione pura.
- c. Paginazione pura.

9.15 Confrontate i modelli della segmentazione pura, paginazione pura e allocazione contigua in riferimento alle seguenti tematiche:

- a. frammentazione esterna;
- b. frammentazione interna;
- c. capacità di condividere codice tra i processi.

9.16 Nei sistemi che si avvalgono della paginazione i processi non possono accedere alla memoria che non possiedono. Perché? In che modo potrebbe il sistema operativo concedere l'accesso a tale memoria estranea? Argomentate perché dovrebbe o non dovrebbe farlo.

9.17 Spiegate perché i sistemi operativi mobili come ios e Android non supportano l'avvicendamento dei processi (swapping).

9.18 Anche se Android non supporta lo swapping sul suo disco di avvio, è possibile impostare uno spazio di swap usando una scheda di memoria non volatile (una scheda sd). Per quale ragione Android non consente lo swapping sul proprio disco di avvio per poi permetterlo su un disco secondario?

9.19 Spiegate perché gli identificatori dello spazio d'indirizzi (asid) vengono utilizzati nei tlb.

9.20 Il codice eseguibile di un programma, in molti sistemi, ha la seguente struttura tipica. Il codice è memorizzato a partire da un piccolo indirizzo virtuale fisso, come, per esempio, 0. Al code segment fa seguito il segmento dei dati, utilizzato per memorizzare le variabili del programma. Quando il programma dà avvio all'esecuzione, lo stack è collocato all'altro estremo dello spazio degli indirizzi virtuali, e ha, dunque, la possibilità di espandersi verso gli indirizzi virtuali inferiori. Quale rilevanza assume la struttura ora descritta nei seguenti schemi di allocazione?

- a. Allocazione contigua della memoria
- b. Paginazione

9.21 Assumendo che la dimensione della pagina sia di 1 kb, quali sono i numeri di pagina e gli scostamenti per i seguenti indirizzi (indicati in numeri decimali):

- a. 21205
- b. 164250
- c. 121357
- d. 16479315
- e. 27253187

9.22 Il sistema operativo MPV è progettato per sistemi embedded e ha un indirizzo virtuale a 24 bit, un indirizzo fisico a 20 bit e una dimensione di pagina di 4 kb. Quante voci ci sono in ciascuna delle seguenti tabelle:

- a. Tabella delle pagine convenzionale a un livello.
- b. Tabella delle pagine invertita.

Qual è la quantità massima di memoria fisica nel sistema operativo MPV?

9.23 Considerate uno spazio di indirizzo logico di 2048 pagine, con una dimensione di pagina di 4 kb, mappato su una memoria fisica di 512 frame.

- a. Quanti bit sono necessari per l'indirizzo logico?

b. Quanti bit sono necessari per l'indirizzo fisico?

9.24 Prendete in considerazione un sistema con un indirizzo logico di 32 bit e una dimensione di pagina di 8 kb. Il sistema supporta fino a 1 GB di memoria fisica. Quante voci ci sono in:

- a. una tabella delle pagine convenzionale a un livello;
- b. una tabella delle pagine invertita.

9.25 Considerate un sistema di paginazione con la tabella delle pagine conservata in memoria.

- a. Se un riferimento alla memoria necessita di 50 nanosecondi, di quanto necessiterà un riferimento alla memoria paginata?
- b. Se si aggiungono TLB, e il 75 percento di tutti i riferimenti si trova in questi ultimi, quale sarà il tempo effettivo di riferimento alla memoria? (Ipotizzate che la ricerca di un elemento, se presente nei TLB, richieda un tempo di 2 nanosecondi.)

9.26 Qual è lo scopo di paginare le tabelle delle pagine?

9.27 Considerate lo schema di traduzione degli indirizzi di ia-32 mostrato nella Figura 9.22.

- a. Descrivete tutti i passi eseguiti da ia-32 nel tradurre un indirizzo logico in un indirizzo fisico.
- b. Esponete i vantaggi offerti a un sistema operativo da un'architettura dotata di un così complesso sistema di traduzione degli indirizzi.
- c. Dite se ci sono svantaggi in questo sistema di traduzione degli indirizzi; se sì, dite quali sono; altrimenti spiegate perché non è impiegato da ogni costruttore.

9.28 Supponete che un sistema abbia un indirizzo virtuale di 32 bit con una dimensione della pagina di 4 kb. Scrivete un programma in C al quale venga passato un indirizzo virtuale (in decimale) dalla riga di comando e che fornisca in output il numero di pagina e l'offset per l'indirizzo dato. Per esempio, se il programma fosse invocato come segue:

```
./addresses 19986
```

il risultato sarebbe:

```
L'indirizzo 19986 contiene:  
numero di pagina = 4  
offset = 3602
```

Scrivere questo programma richiederà di utilizzare tipi di dati appropriati per memorizzare 32 bit. Suggeriamo di utilizzare tipi di dati `unsigned`.

Allocazione di memoria contigua

Nel Paragrafo 9.2 abbiamo presentato diversi algoritmi per l'allocazione di memoria contigua. Questo progetto comporterà la gestione di una regione contigua di memoria di dimensione MAX dove gli indirizzi possono variare nel campo $0 \dots MAX - 1$. Il programma deve rispondere alle seguenti quattro richieste diverse.

1. Richiesta di un blocco di memoria contiguo.
2. Rilascio di un blocco contiguo di memoria.
3. Compattazione dei buchi di memoria inutilizzati in un unico blocco.
4. Segnalazione delle regioni di memoria libera e allocata.

Il vostro programma riceverà la quantità iniziale di memoria all'avvio. Per esempio, il seguente comando lancia il programma con 1 mb (1.048.576 byte) di memoria:

```
./allocator 1048576
```

Una volta avviato il programma, questo presenterà all'utente il seguente prompt:

```
allocator>
```

Risponderà quindi ai seguenti comandi: `RQ` (richiesta), `RL` (rilascio), `C` (compattazione), `STAT` (rapporto di stato) e `X` (uscita). Una richiesta di 40.000 byte verrà effettuata come segue:

```
allocator>RQ P0 40000 W
```

Il primo parametro del comando `RQ` è il nuovo processo che richiede la memoria, seguito dalla quantità di memoria richiesta e infine dalla strategia. (In questa situazione, “`W`” si riferisce al worst fit).

Analogamente, un rilascio verrà effettuato in questo modo:

```
allocator>RL P0
```

Questo comando rilascerà la memoria che è stata allocata al processo P0.

Il comando per la compattazione è:

```
allocator>C
```

Questo comando comprime i buchi di memoria inutilizzati in una regione.

Infine, viene utilizzato il comando `STAT` per riportare lo stato della memoria:

```
allocator>STAT
```

Dato questo comando, il vostro programma riporterà le regioni di memoria che sono assegnate e le regioni non utilizzate. Per esempio, una possibile situazione di allocazione di memoria potrebbe essere la seguente:

```
Addresses [0: 315000] Process P1  
Addresses [315001: 512500] Process P3  
Addresses [512501: 625575] Unused  
Addresses [625575: 725100] Process P6  
Addresses [725001] . . .
```

Allocazione della memoria

Il vostro programma assegnerà la memoria utilizzando uno dei tre approcci evidenziati nel Paragrafo 9.2.2, a seconda del flag che viene passato al comando `RQ`. I flag sono 5847 i seguenti:

- F – first fit
- B – best fit
- W – worst fit

Ciò richiederà che il vostro programma tenga traccia dei diversi buchi che rappresentano memoria disponibile. Quando arriva una richiesta di memoria, verrà allocata la memoria da uno dei buchi disponibili in base alla strategia di allocazione. Se la memoria è insufficiente per soddisfare una richiesta, verrà emesso un messaggio di errore e la richiesta verrà rifiutata.

Il vostro programma dovrà anche tenere traccia di quale regione della memoria sia stata assegnata a quale processo. Ciò è necessario per supportare il comando `STAT` ed è anche necessario quando la memoria viene rilasciata tramite il comando `RL`, dato che

l'identificativo del processo che rilascia la memoria viene passato a questo comando. Se una partizione rilasciata è adiacente a un buco esistente, assicuratevi di unire i due buchi in un buco singolo.

Compattazione

Se l'utente inserisce il comando `c`, il programma comprimerà il set di buchi in un unico buco più grande. Per esempio, se avete quattro buchi separati di dimensione 550 kb, 375 kb, 1.900 kb e 4.500 kb, il vostro programma combinerà questi quattro buchi in un grande buco di dimensioni 7.325 kb.

Esistono diverse strategie per implementare la compattazione, una delle quali è suggerita nel Paragrafo 9.2.3. Assicuratevi di aggiornare l'indirizzo iniziale di qualsiasi processo che sia stato coinvolto nella compattazione.

PARTE I Generalità

Un *sistema operativo* è un programma che agisce come intermediario tra l'utente e l'hardware di un calcolatore. Scopo di un sistema operativo è fornire un ambiente nel quale un utente possa eseguire programmi in modo *conveniente* ed *efficiente*.

Un sistema operativo è il software che gestisce l'hardware di un calcolatore. L'hardware deve fornire meccanismi idonei che assicurino il corretto funzionamento dell'elaboratore e lo preservino da eventuali interferenze improprie da parte dei programmi utenti.

La struttura interna dei sistemi operativi è soggetta a notevole variabilità ed è adattabile a criteri di organizzazione estremamente differenti. La progettazione di un nuovo sistema operativo è un compito impegnativo che richiede, in via preliminare, una chiara definizione degli obiettivi del sistema. In base a tali obiettivi si selezionano le possibili strategie e si individuano i relativi algoritmi.

I sistemi operativi sono programmi complessi e di vaste dimensioni e vanno pertanto realizzati un pezzo per volta, per moduli. Ciascun modulo dovrebbe costituire una parte del sistema chiaramente identificata: è necessario definirne scrupolosamente sia le funzionalità sia i dati in ingresso e in uscita.

CAPITOLO 10

Memoria virtuale

Nel Capitolo 9 sono state esaminate varie strategie di gestione della memoria impiegate nei calcolatori. Hanno tutte lo stesso scopo: tenere contemporaneamente più processi in memoria per permettere la multiprogrammazione; tuttavia esse tendono a richiedere che l'intero processo si trovi in memoria prima di essere eseguito.

La memoria virtuale è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatori dai problemi di limitazione della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e di realizzare memorie condivise, e fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è però difficile da realizzare e, s'è usata scorrettamente, può ridurre di molto le prestazioni del sistema. In questo capitolo si esamina la memoria virtuale, come viene implementata, la sua complessità e i suoi benefici.

10.1 Introduzione

Gli algoritmi di gestione della memoria delineati nel Capitolo 9 sono necessari a causa di un requisito fondamentale: le istruzioni da eseguire si devono trovare all'interno della memoria fisica. Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo in memoria fisica. Il caricamento dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatori.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica. In effetti, da un esame dei programmi reali risulta che in molti casi non è necessario avere in memoria l'intero programma; si considerino per esempio le seguenti situazioni.

- Spesso i programmi dispongono di codice per la gestione di condizioni d'errore insolite. Poiché questi errori sono rari, se non inesistenti, anche il relativo codice non si esegue quasi mai.
- Spesso ad array, liste e tabelle si assegna più memoria di quanta sia effettivamente necessaria. Un array si può dichiarare di 100 per 100 elementi, anche se raramente contiene più di 10 per 10 elementi.
- Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado.

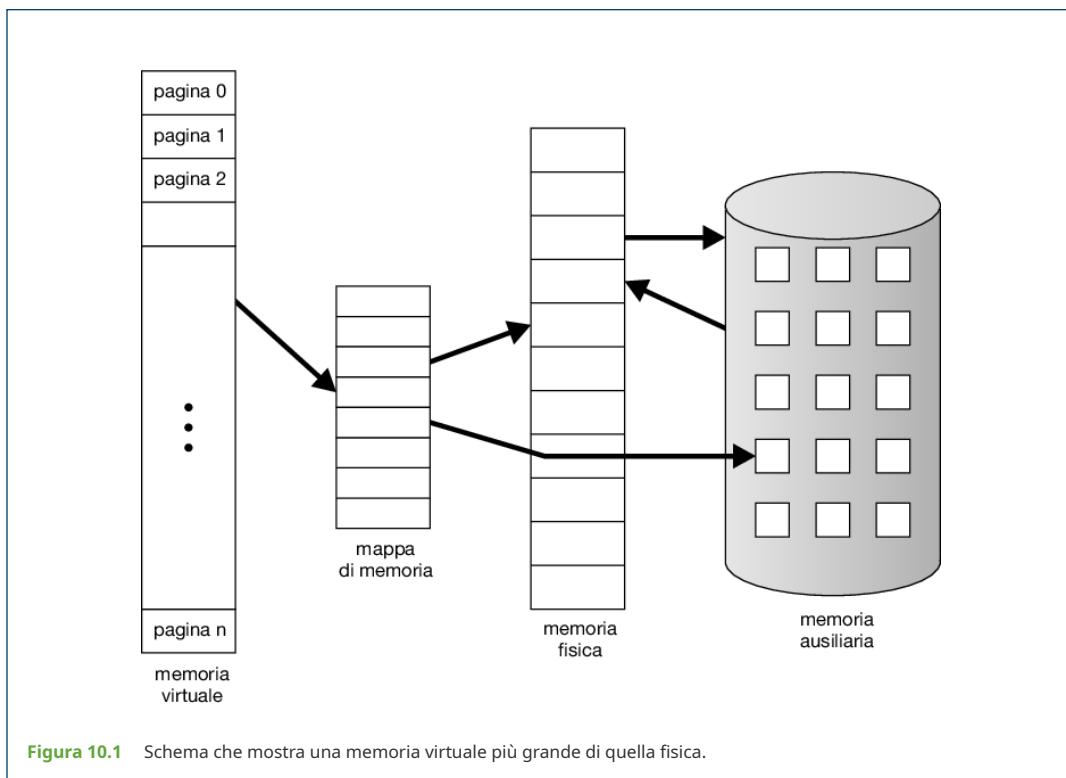
Anche nei casi in cui è necessario disporre di tutto il programma, è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria porterebbe molti benefici.

- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno spazio degli indirizzi virtuali molto grande, semplificando così il compito della programmazione.
- Poiché ogni programma utente può impiegare meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della cpu senza aumentare il tempo di risposta o di completamento.
- Per caricare (o avvicendare) ogni programma utente in memoria sono necessarie meno operazioni di i/o, quindi ogni programma utente è eseguito più rapidamente.

La possibilità di eseguire un programma che non si trovi completamente in memoria apporterebbe quindi vantaggi sia al sistema sia all'utente.

La memoria virtuale si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmati una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola, com'è illustrato nella Figura 10.1. La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile, ma può concentrarsi sul problema da risolvere con il programma.



L'espressione spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Tipicamente, da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico – per esempio, l'indirizzo 0 – e si estende in uno spazio di memoria contigua, come evidenziato dalla Figura 10.2. Come si ricorderà dal Capitolo 9, è tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (mmu) associare in memoria le pagine logiche alle pagine fisiche.

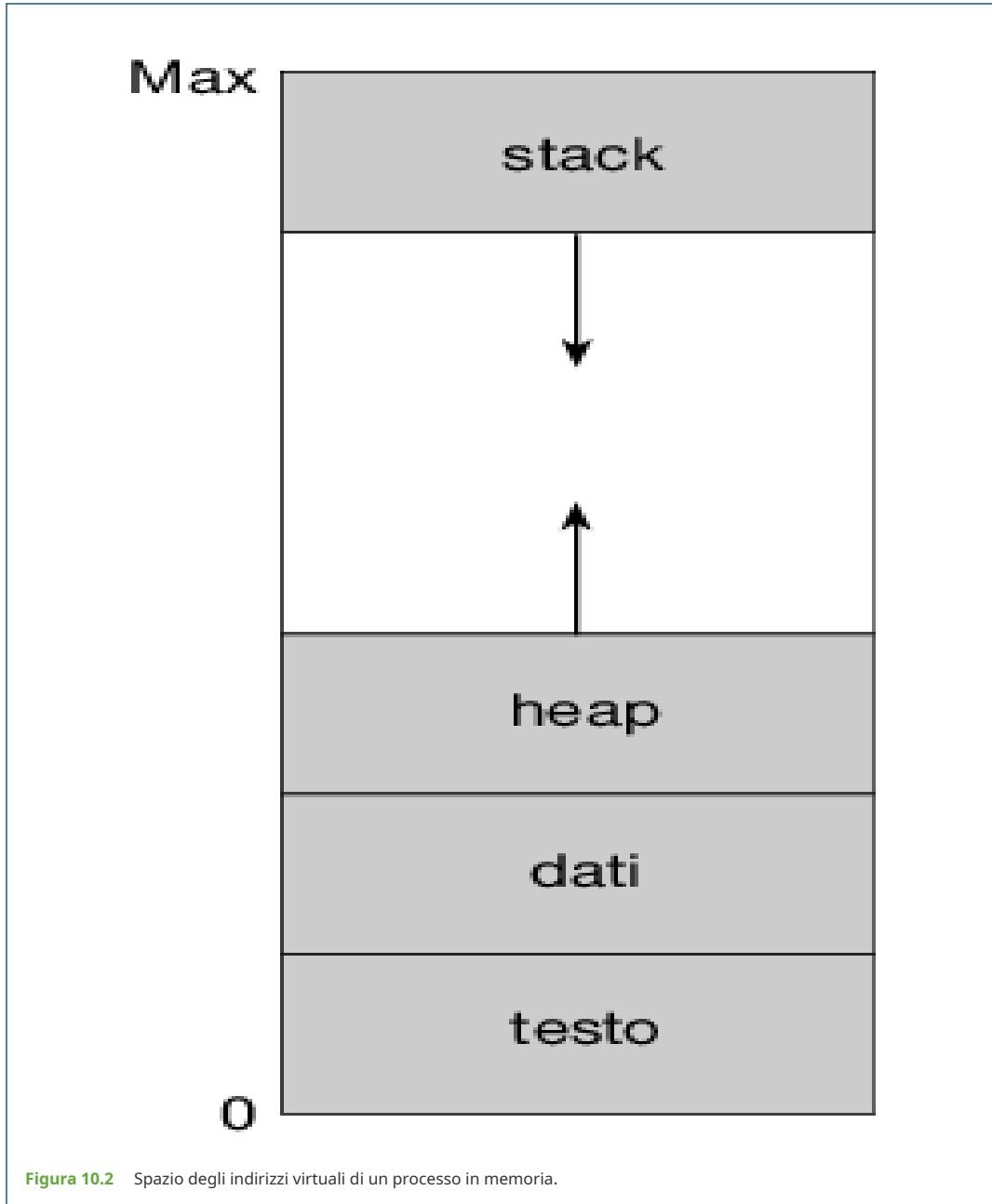


Figura 10.2 Spazio degli indirizzi virtuali di un processo in memoria.

Si noti come, nella Figura 10.2, allo heap sia lasciato sufficiente spazio per crescere verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo allo stack di svilupparsi verso il basso nella memoria, quando vengono effettuate ripetute chiamate di funzione. L'ampio spazio vuoto (o buco) che separa lo heap dallo stack è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche reali solo nel caso che lo heap o lo stack crescano. Uno spazio degli indirizzi virtuali che contiene buchi si definisce sparso. Un simile spazio degli indirizzi è utile, poiché i buchi possono essere riempiti grazie all'espansione dei segmenti heap o stack, oppure se vogliamo collegare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre il vantaggio di condividere i file e la memoria fra due o più processi, mediante la condivisione delle pagine (Paragrafo 9.3.4). Ciò comporta i seguenti vantaggi.

- Le librerie di sistema sono condivisibili da diversi processi associando (“mappando”) l’oggetto di memoria condiviso a uno spazio degli indirizzi virtuali. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi (Figura 10.3). In genere le librerie si associano allo spazio di ogni processo a loro collegato, in modalità di sola lettura.

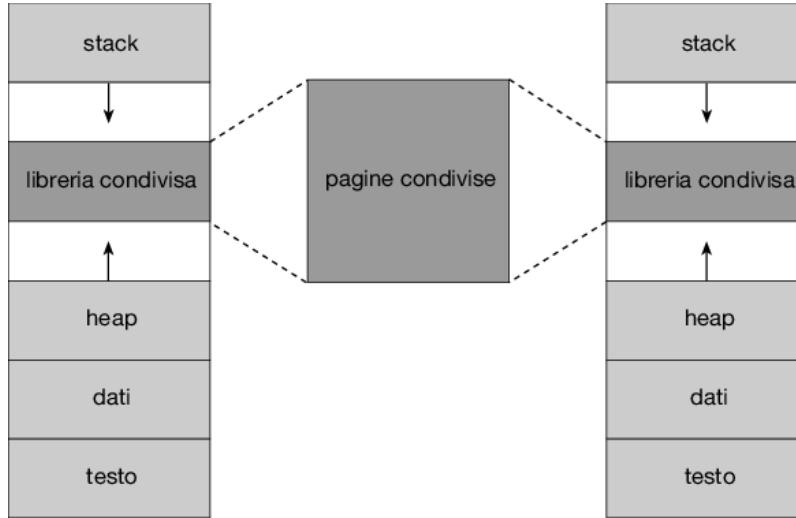


Figura 10.3 Condivisione delle librerie tramite la memoria virtuale.

- In maniera analoga, la memoria può essere condivisa tra processi distinti. Come si rammenterà dal Capitolo 3, due o più processi possono comunicare condividendo memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise, come illustrato sempre dalla Figura 10.3.
- Le pagine possono essere condivise durante la creazione di un processo mediante la chiamata di sistema `fork()`, così da velocizzare la generazione dei processi.

Approfondiremo questi e altri vantaggi offerti dalla memoria virtuale nel corso di questo capitolo. In primo luogo, però, ci soffermeremo sulla memoria virtuale realizzata attraverso la paginazione su richiesta.

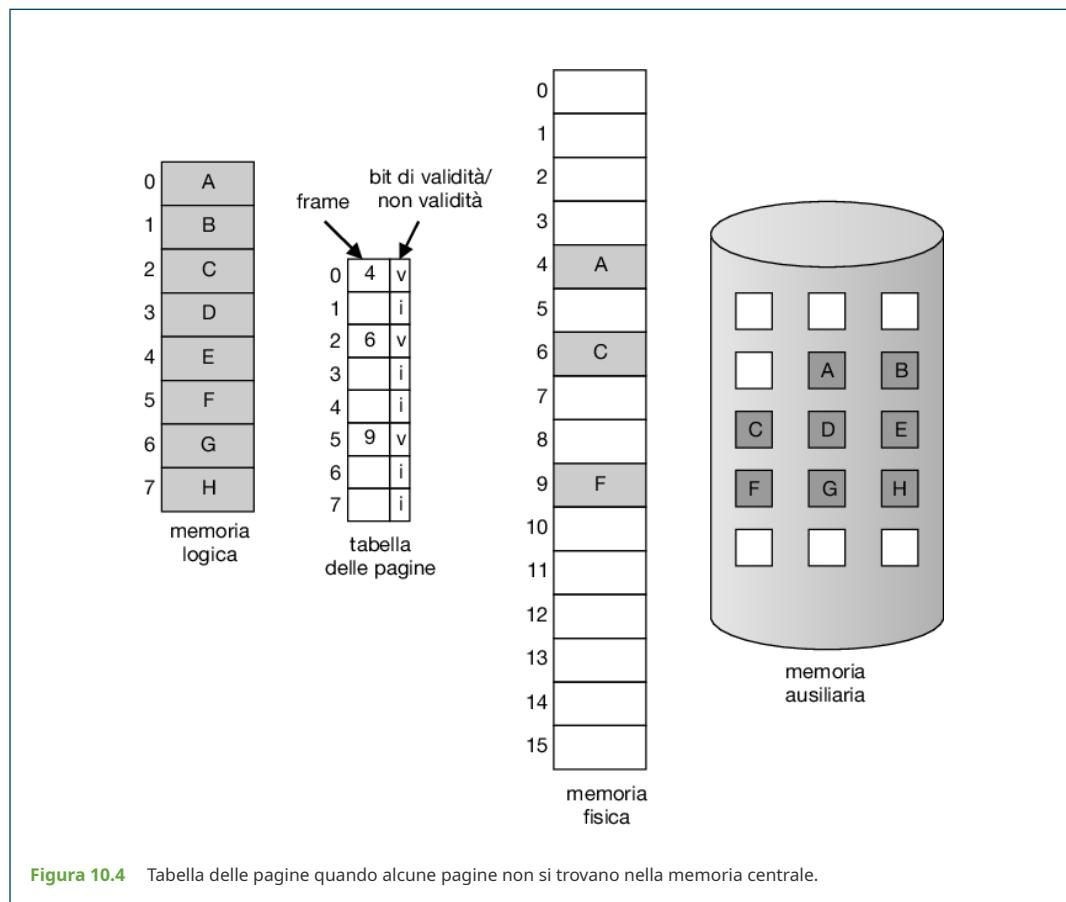
10.2 Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente.

Una strategia alternativa consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica, detta paginazione su richiesta, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica. Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria; si veda il Paragrafo 9.5.2. I processi risiedono in memoria secondaria (generalmente su disco o su un altro supporto di memoria non volatile). La paginazione su richiesta mostra uno dei principali vantaggi della memoria virtuale: caricando solo le parti necessarie dei programmi la memoria viene utilizzata in modo più efficiente.

10.2.1 Concetti fondamentali

Il concetto generale alla base della paginazione su richiesta, come è stato detto, è di caricare una pagina in memoria solo quando è necessaria. Di conseguenza, mentre un processo è in esecuzione, alcune pagine saranno in memoria e altre si troveranno nella memoria secondaria. È dunque necessaria una forma di supporto hardware per distinguere tra i due casi. A tale scopo si può utilizzare lo schema basato sul bit di validità, descritto nel Paragrafo 9.3.3. In questo caso, però, il bit impostato come "valido" significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come "non valido" indica che la pagina non è valida (cioè non appartiene allo spazio d'indirizzi logici del processo) oppure è valida, ma è attualmente nella memoria secondaria. L'elemento della tabella delle pagine corrispondente a una pagina caricata in memoria s'impone come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido. Tale situazione è illustrata nella Figura 10.4. Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi.



Che cosa succede se il processo tenta l'accesso a una pagina che non era stata caricata in memoria? L'accesso a una pagina contrassegnata come non valida causa un evento o eccezione di page fault (*pagina mancante*). L'hardware di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e genera una trap per il sistema operativo; tale eccezione è dovuta a un “insuccesso” del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell'eccezione di page fault è lineare (Figura 10.5).

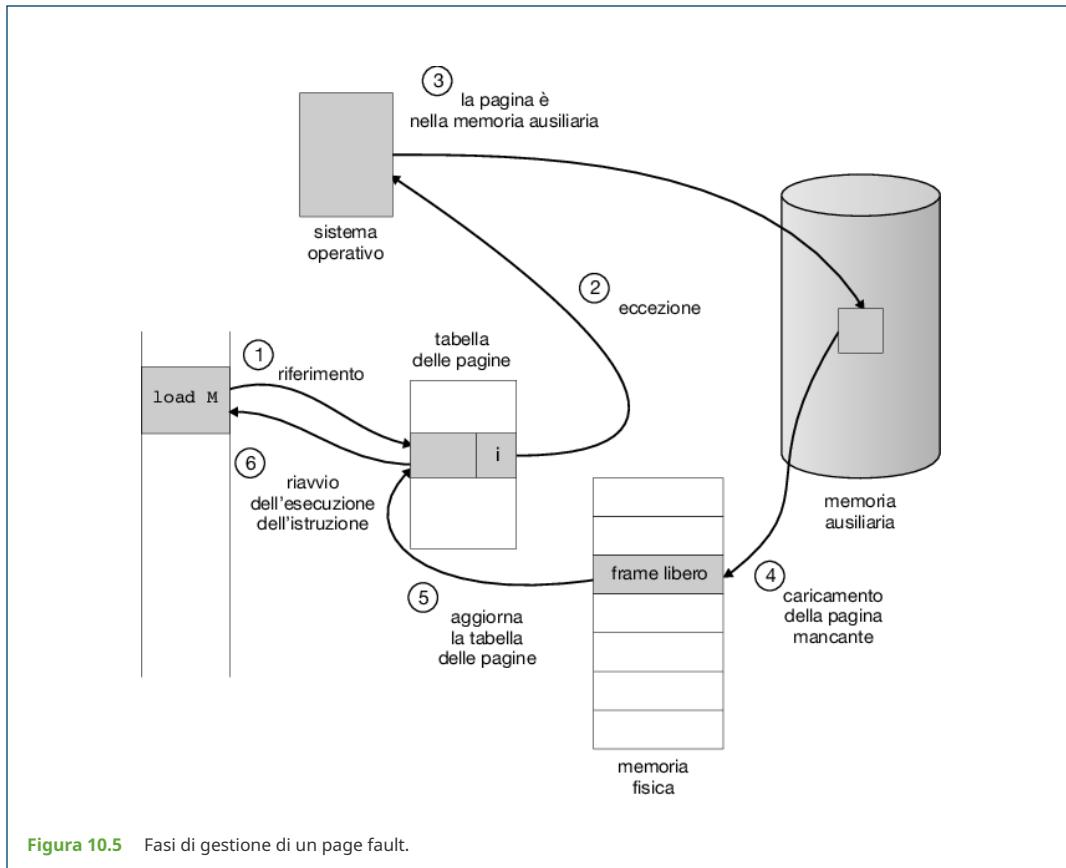


Figura 10.5 Fasi di gestione di un page fault.

- Si controlla una tabella interna per questo processo (in genere tale tabella è conservata insieme al blocco di controllo del processo) allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.
- Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua il caricamento.
- Si individua un frame libero, per esempio prelevandone uno dalla lista dei frame liberi.
- Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
- Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
- Si riavvia l'istruzione interrotta dall'eccezione. A questo punto il processo può accedere alla pagina come se questa fosse stata sempre presente in memoria.

Come caso estremo, è possibile avviare l'esecuzione di un processo *senza* pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo genera immediatamente un page fault. Una volta portata la pagina in memoria, il processo continua l'esecuzione, generando page fault fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una paginazione su richiesta pura, vale a dire che una pagina non si trasferisce mai in memoria se non viene richiesta.

In teoria alcuni programmi possono accedere a diverse nuove pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e molte per i dati), eventualmente causando più page fault per ogni istruzione. In un caso simile le prestazioni del sistema sarebbero inaccettabili. Fortunatamente l'analisi dei programmi in esecuzione mostra che questo comportamento è estremamente improbabile. I programmi tendono ad avere una località dei riferimenti, descritta nel Paragrafo 10.6.1, quindi le prestazioni della paginazione su richiesta risultano ragionevoli.

L'hardware di supporto alla paginazione su richiesta è lo stesso che è richiesto per la paginazione e l'avvicendamento dei processi in memoria:

- tabella delle pagine. Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- memoria secondaria: Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo di swap; la sezione del disco usata a questo scopo si chiama area di avvicendamento (*swap space*). L'allocazione di quest'area è trattata nel Capitolo 11.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione dopo un page fault. Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) al momento del page fault, occorrerà riavviare il processo *esattamente* dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questo requisito è facile da soddisfare. Un page fault si può verificare per qualsiasi riferimento alla memoria. Se si verifica durante la fase di fetch (prelivo) di un'istruzione, l'esecuzione si può riavviare effettuando nuovamente il fetch. Se si verifica durante il fetch di un operando, bisogna effettuare nuovamente fetch e decode dell'istruzione, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione a tre indirizzi, come per esempio la somma (*ADD*) del contenuto di *A* al contenuto di *B*, con risultato posto in *C*. I passi necessari per eseguire l'istruzione sono i seguenti:

1. fetch e decodifica dell'istruzione (*ADD*);
2. prelivo del contenuto di *A*;
3. prelivo del contenuto di *B*;
4. addizione del contenuto di *A* al contenuto di *B*;
5. memorizzazione della somma in *C*.

Se il page fault avviene al momento della memorizzazione in *C*, poiché *C* si trova in una pagina che non è in memoria, occorre prelevare la pagina desiderata, caricarla in memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'istruzione richiede una nuova operazione di fetch, con nuova decodifica e nuovo prelivo dei due operandi; infine occorre ripetere l'addizione. In ogni modo il lavoro da ripetere non è molto, meno di un'istruzione completa, e la ripetizione è necessaria solo nel caso si verifichi un page fault.

La difficoltà maggiore si presenta quando un'istruzione può modificare parecchie locazioni diverse. Si consideri, per esempio, l'istruzione *MVC* (*move character*) del sistema ibm 360/370: quest'istruzione può spostare una sequenza di byte (fino a 256) da una locazione a un'altra (con possibilità di sovrapposizione). Se una delle sequenze (quella d'origine o quella di destinazione) esce dal confine di una pagina, si può verificare un page fault quando lo spostamento è stato effettuato solo in parte. Inoltre, se le sequenze d'origine e di destinazione si sovrappongono, è possibile che la sequenza d'origine sia stata modificata, in tal caso non è possibile limitarsi a riavviare l'istruzione.

Il problema si può risolvere in due modi. In una delle due soluzioni il microcodice computa e tenta di accedere alle estremità delle due sequenze di byte. Un'eventuale page fault si può verificare solo in questa fase, prima che si apporti qualsiasi modifica. A questo punto si può compiere lo spostamento senza rischio di page fault perché tutte le pagine interessate si trovano in memoria. L'altra soluzione si serve di registri temporanei per conservare i valori delle locazioni sovrascritte. Nel caso di un page fault, si riscrivono tutti i vecchi valori in memoria prima che sia generata la trap. Questa operazione riporta la memoria allo stato in cui si trovava prima che l'istruzione fosse avviata, perciò si può ripetere la sua esecuzione.

Sebbene non si tratti certo dell'unico problema da affrontare per estendere un'architettura esistente con la funzionalità della paginazione su richiesta, illustra alcune delle difficoltà da superare. Il sistema di paginazione si colloca tra la cpu e la memoria di un calcolatore e deve essere completamente trasparente al processo utente. L'opinione comune che la paginazione si possa aggiungere a qualsiasi sistema è vera per gli ambienti senza paginazione su richiesta, nei quali un'eccezione di page fault rappresenta un errore fatale, ma è falsa nei casi in cui un'eccezione di page fault implica solo la necessità di caricare in memoria un'altra pagina e quindi riavviare il processo.

10.2.2 Lista dei frame liberi

Quando si verifica un page fault, il sistema operativo deve spostare la pagina desiderata dalla memoria secondaria alla memoria principale. Per risolvere i page fault la maggior parte dei sistemi operativi mantiene una lista dei frame liberi, ovvero un insieme di frame disponibili e utilizzabili per soddisfare le richieste, come mostrato nella Figura 10.6 (i frame liberi devono essere allocati anche quando lo stack o l'heap di un processo si espandono). I sistemi operativi allocano generalmente i frame liberi usando una tecnica nota come zero-fill-on-demand ("riempimento con zeri su richiesta"): i frame vengono "azzerati" su richiesta prima di essere allocati, cancellando così il loro precedente contenuto (si considerino le potenziali implicazioni sulla sicurezza della non eliminazione del contenuto di un frame prima di riassegnarlo).



Figura 10.6 Lista dei frame liberi.

All'avvio di un sistema tutta la memoria disponibile viene inserita nella lista dei frame liberi. Man mano che vengono richiesti frame liberi (per esempio, tramite paginazione su richiesta), la dimensione della lista dei frame liberi si riduce. A un certo punto la lista

diventa vuota, oppure la sua dimensione scende al di sotto di una certa soglia fissata: quando ciò si verifica la lista deve essere ripopolata. Tratteremo le strategie per risolvere entrambe queste situazioni nel Paragrafo 10.4.

10.2.3 Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il tempo d'accesso effettivo per una memoria con paginazione su richiesta. Il tempo d'accesso alla memoria, che si denota *ma*, è di 10 nanosecondi. Finché non si verifichino page fault, il tempo d'accesso effettivo è uguale al tempo d'accesso alla memoria. Se però si verifica un page fault, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola della memoria desiderata.

Supponendo che p sia la probabilità che si verifichi un page fault ($0 \leq p \leq 1$), è probabile che p sia molto vicina allo zero, cioè che ci siano solo pochi fault. Il tempo d'accesso effettivo è dato dalla seguente espressione:

$$\text{tempo d'accesso effettivo} = (1 - p) \times ma + p \times \text{tempo di gestione del page fault}$$

Per calcolare il tempo d'accesso effettivo occorre conoscere il tempo necessario alla gestione di un page fault. In tal caso si deve eseguire la seguente sequenza:

1. trap per il sistema operativo;
2. salvataggio dei registri utente e dello stato del processo;
3. verifica che l'interruzione sia dovuta o meno a un page fault;
4. controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco;
5. lettura dal disco e trasferimento in un frame libero:
 - a) attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita;
 - b) attesa del tempo di posizionamento e latenza del dispositivo;
 - c) inizio del trasferimento della pagina in un frame libero;
6. durante l'attesa, allocazione della cpu a un altro processo utente (scheduling della cpu, facoltativo);
7. ricezione di un'interruzione dal controllore del disco (i/o completato);
8. salvataggio dei registri e dello stato dell'altro processo utente (se è stato eseguito il passo 6);
9. verifica della provenienza dell'interruzione dal disco;
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. attesa che la cpu sia nuovamente assegnata a questo processo;
12. ripristino dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta.

Non sempre sono necessari tutti i passi sopra elencati. Nel passo 6, per esempio, si ipotizza che la cpu sia assegnata a un altro processo durante un'operazione di i/o. Tale possibilità permette la multiprogrammazione per mantenere occupata la cpu, ma una volta completato il trasferimento di i/o implica un dispendio di tempo per riprendere la procedura di servizio dell'eccezione di page fault.

In ogni caso, il tempo di servizio dell'eccezione di page fault ha tre componenti principali:

1. servizio della eccezione di page fault;
2. lettura della pagina da disco;
3. riavvio del processo.

La prima e la terza operazione si possono ridurre, per mezzo di un'accurata codifica, ad alcune centinaia di istruzioni. Ciascuna di queste operazioni può quindi richiedere da 1 a 100 microsecondi. D'altra parte, il tempo di trasferimento di pagina è probabilmente vicino a 8 millisecondi (un disco ha in genere un tempo di latenza di 3 millisecondi, un tempo di posizionamento di 5 millisecondi e un tempo di trasferimento di 0,05 millisecondi, quindi il tempo totale della paginazione è dell'ordine di 8 millisecondi, comprendendo le tempistiche hardware e software). Inoltre nel calcolo si è considerato solo il tempo di servizio del dispositivo. Se una coda di processi è in attesa del dispositivo è necessario considerare anche il tempo di accodamento del dispositivo, poiché occorre attendere che il dispositivo di paginazione sia libero per servire la richiesta, quindi il tempo di trasferimento aumenta ulteriormente.

Considerando un tempo medio di servizio dell'eccezione di page fault di 8 millisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$\text{tempo d'accesso effettivo} = (1 - p) \times 200 + p (8 \text{ millisecondi}) = (1 - p) \times 200 + p \times 8.000.000 = 200 + 7.999.800 \times p$$

Il tempo d'accesso effettivo è direttamente proporzionale al tasso di page fault (*page-fault rate*). Se un accesso su 1000 accusa un page fault, il tempo d'accesso effettivo è di 8,2 microsecondi. Impiegando la paginazione su richiesta, il calcolatore è rallentato di un fattore pari a 40! Se si desidera un rallentamento inferiore al 10 per cento, occorre contenere la probabilità di page fault al seguente livello:

$$220 > 200 + 7.999.800 \times p \quad 20 > 7.999.800 \times p \quad p < 0.00000025$$

Quindi, per mantenere a un livello ragionevole il rallentamento dovuto alla paginazione, si può permettere meno di un page fault ogni 399.990 accessi alla memoria. In un sistema con paginazione su richiesta, è cioè importante tenere basso il tasso di page fault, altrimenti il tempo effettivo d'accesso aumenta, rallentando molto l'esecuzione del processo.

Un altro aspetto della paginazione su richiesta è la gestione e l'uso generale dell'area di swap. L'i/o di un disco relativo all'area di swap è generalmente più rapido di quello relativo al file system (Capitolo 11): ciò si deve al fatto che lo spazio di swap è allocato in blocchi molto grandi e non vengono utilizzate ricerche e riferimenti indiretti. Perciò il sistema può migliorare l'efficienza della paginazione copiando tutta l'immagine di un file nell'area di swap all'avvio del processo e di lì eseguire la paginazione su richiesta. Un'altra possibilità consiste nel richiedere inizialmente le pagine al file system, ma scrivere le pagine nell'area di swap al momento della sostituzione. Questo metodo assicura che si leggano sempre dal file system solo le pagine necessarie, ma che tutta la paginazione successiva sia fatta dall'area di swap.

Alcuni sistemi tentano di limitare l'area di swap utilizzata per file binari: le pagine richieste per questi file si prelevano direttamente dal file system; tuttavia, quando è richiesta una sostituzione di pagine, i frame possono semplicemente essere sovrascritti, dato che

non sono mai stati modificati, e le pagine, se è necessario, possono essere nuovamente lette dal file system. Seguendo questo criterio, lo stesso file system funziona da memoria ausiliaria (*backing store*). L'area di swap si deve in ogni caso usare per le pagine che non sono relative ai file (la cosiddetta memoria anonima); queste comprendono lo stack e lo heap di un processo. Questa tecnica che sembra essere un buon compromesso si usa in diversi sistemi tra cui Linux e unix bsd.

Come descritto nel Paragrafo 9.5.3, i sistemi operativi mobili non supportano, in genere, lo swapping, ma, in caso di carenza di memoria, richiedono pagine al file system e recuperano pagine di sola lettura (come il codice) dalle applicazioni. Se necessario, questi dati possono essere richiesti di nuovo al file system. In ios non vengono mai riprese a un'applicazione le pagine di memoria anonima a meno che l'applicazione sia terminata o abbia rilasciato la memoria in maniera volontaria. Nel Paragrafo 10.7 presenteremo la memoria compressa, un'alternativa frequentemente utilizzata per lo swapping sui sistemi mobili.

10.3 Copiatura su scrittura

Nel Paragrafo 10.2 si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione. La generazione dei processi tramite `fork()`, però, può inizialmente evitare la paginazione su richiesta per mezzo di una tecnica simile alla condivisione delle pagine (Paragrafo 9.3.4), che garantisce la celere generazione dei processi riuscendo anche a minimizzare il numero di nuove pagine allocate al processo appena creato.

Si ricordi che la chiamata di sistema `fork()` crea un processo figlio come duplicato del genitore. Nella sua versione originale la `fork()` creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la loro creazione la chiamata di sistema `exec()`, questa operazione di copiatura può essere inutile. Come alternativa, si può impiegare una tecnica nota come copiatura su scrittura (*copy-on-write*), il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina. La copiatura su scrittura è illustrata nella Figura 10.7 e nella Figura 10.8, che mostrano il contenuto della memoria fisica prima e dopo che il processo 1 abbia modificato la pagina C.

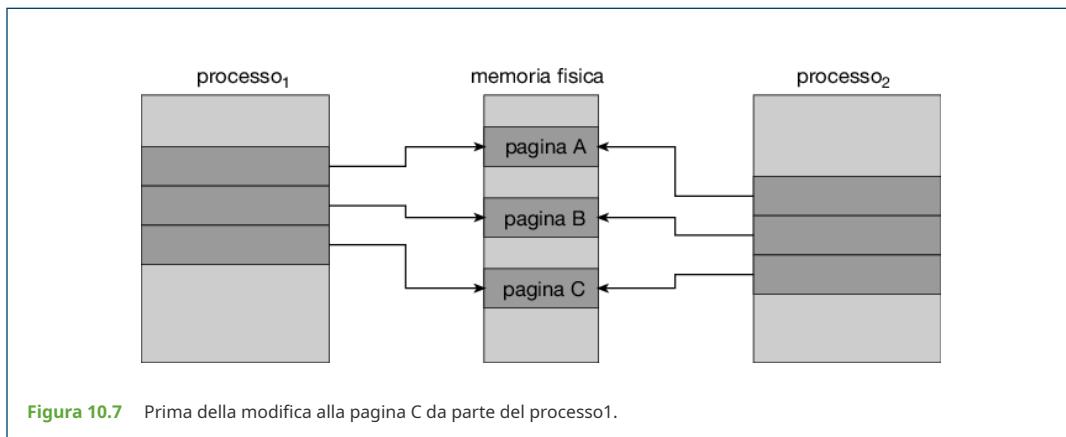


Figura 10.7 Prima della modifica alla pagina C da parte del processo1.

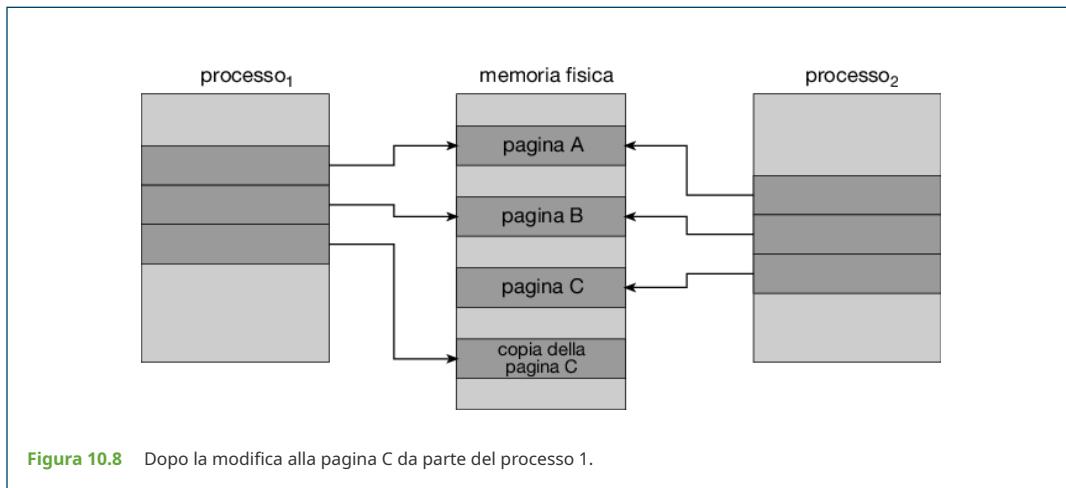


Figura 10.8 Dopo la modifica alla pagina C da parte del processo 1.

Si consideri per esempio un processo figlio che cerchi di modificare una pagina contenente parti della stack, quando le pagine sono contrassegnate come copy-on write. Il sistema operativo crea una copia della pagina nello spazio degli indirizzi del processo figlio. Il processo figlio modifica la sua copia della pagina e non la pagina appartenente al processo genitore. È chiaro che, adoperando la tecnica di copiatura su scrittura, si copiano soltanto le pagine modificate da uno dei due processi, mentre tutte le altre sono condivisibili dai processi genitore e figlio. Si noti inoltre che soltanto le pagine modificabili si devono contrassegnare come copy-on write, mentre quelle che non si possono modificare (per esempio, le pagine contenenti codice eseguibile) sono condivisibili dai

processi genitore e figlio. La tecnica di copiatura su scrittura è piuttosto comune e si usa in diversi sistemi operativi, tra i quali Windows, Linux e macos.

Diverse versioni di unix (compreso Linux, macos e unix bsd) offrono anche una variante della chiamata di sistema `fork()` – detta `vfork()` (per virtual memory fork). La `vfork()` offre un'alternativa all'uso della `fork()` con copiatura su scrittura. Con la `vfork()` il processo genitore viene sospeso e il processo figlio usa lo spazio d'indirizzi del genitore. Poiché la `vfork()` non usa la copiatura su scrittura, se il processo figlio modifica qualche pagina dello spazio d'indirizzi del genitore, le pagine modificate saranno visibili al processo genitore non appena riprenderà il controllo. È quindi necessaria molta attenzione nell'uso di `vfork()`, per assicurarsi che il processo figlio non modifichi lo spazio d'indirizzi del genitore. La chiamata di sistema `vfork()` è adatta al caso in cui il processo figlio esegua una `exec()` immediatamente dopo la sua creazione. Poiché non richiede alcuna copiatura delle pagine, la `vfork()` è un metodo di creazione dei processi molto efficiente, in alcuni casi impiegato per realizzare le interfacce shell in unix.

10.4 Sostituzione delle pagine

Nelle descrizioni fatte finora sul tasso di page fault abbiamo supposto che ogni pagina poteva dar luogo al massimo a un fault, la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'i/o necessario per caricare le cinque pagine che non sono mai usate. Inoltre il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

Aumentando il grado di multiprogrammazione, si *sovrassegna* la memoria. Eseguendo sei processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s'incrementerebbero l'utilizzo e la produttività della cpu e si avrebbero ancora 10 frame disponibili. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'i/o impegnano una rilevante quantità di memoria. Ciò può aumentare le difficoltà agli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all'i/o e quanta alle pagine dei programmi è un problema complesso. Alcuni sistemi riservano una quota fissa di memoria per l'i/o, altri permettono sia ai processi utenti sia al sottosistema di i/o di competere per tutta la memoria del sistema. Il Paragrafo 14.6 approfondisce la relazione tra i buffer di i/o e le tecniche di memoria virtuale.

La sovrallocazione (*over-allocation*) si può illustrare come segue. Durante l'esecuzione di un processo utente si verifica un page fault. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è vuota: tutta la memoria è in uso. Questa situazione è illustrata nella Figura 10.9, dove il punto interrogativo indica il fatto che non ci sono frame liberi.

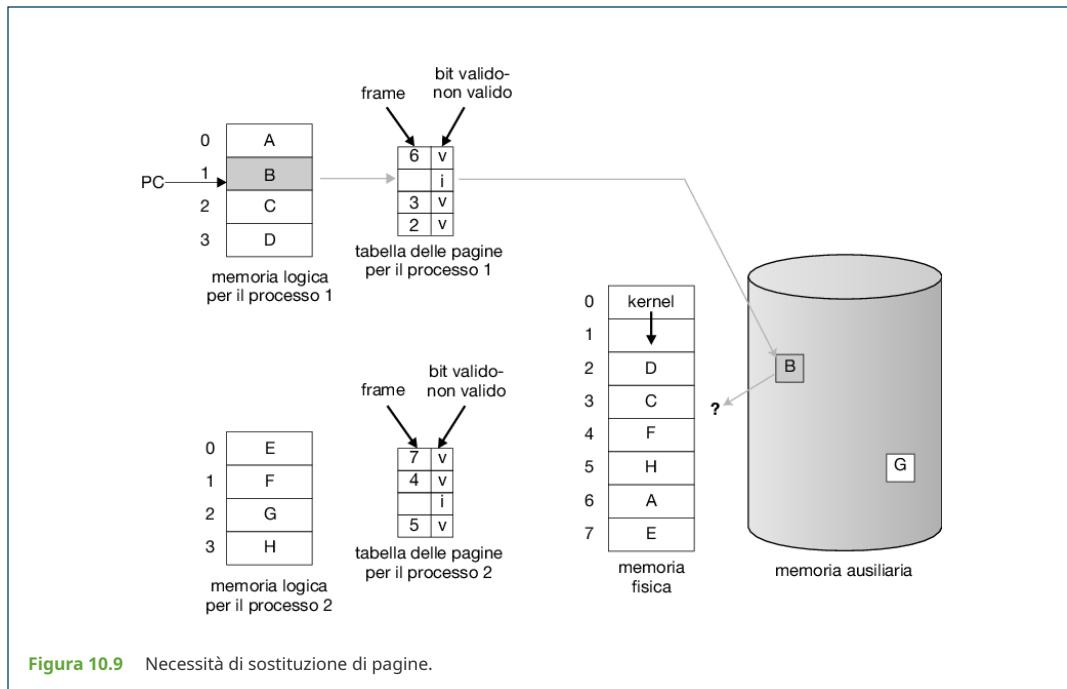


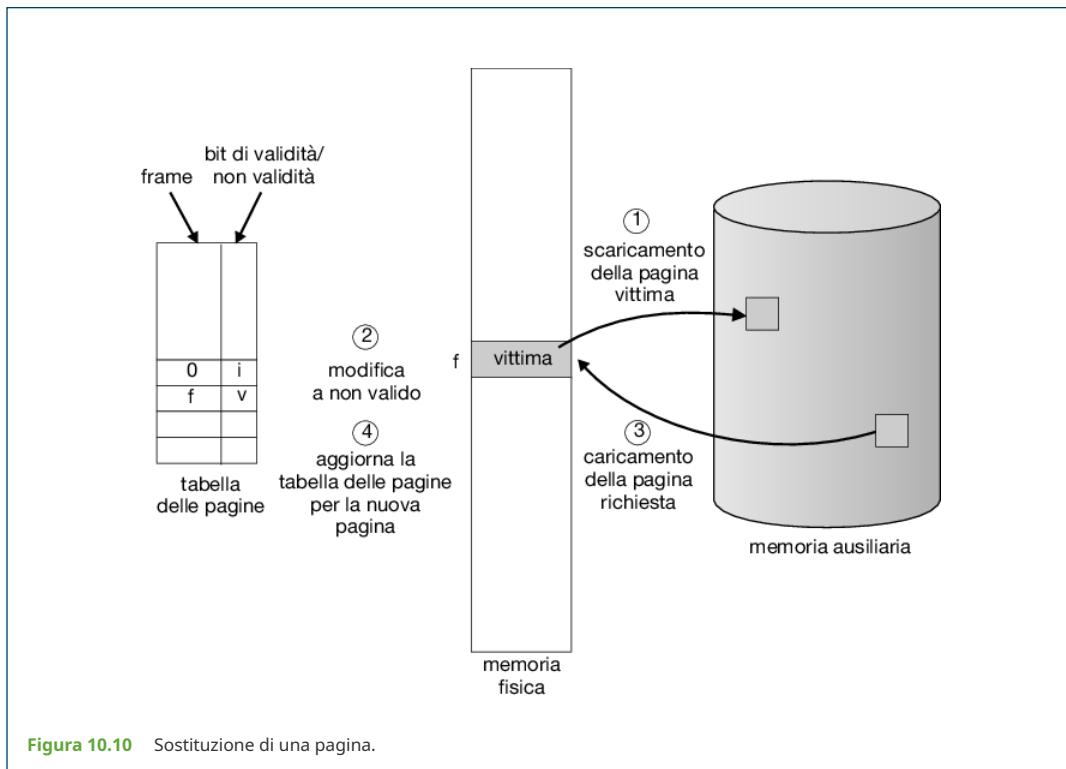
Figura 10.9 Necessità di sostituzione di pagine.

A questo punto il sistema operativo può scegliere tra diverse possibilità, per esempio può terminare il processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l'utilizzo e la produttività del sistema di calcolo. Gli utenti non dovrebbero sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l'utente, quindi la terminazione del processo non costituisce la scelta migliore.

Il sistema operativo può scaricare dalla memoria un intero processo, liberando tutti i suoi frame e riducendo il livello di multiprogrammazione. Tuttavia, come discusso nel Paragrafo 9.5, lo swapping standard non è più utilizzato dalla maggior parte dei sistemi operativi a causa del sovraccarico indotto dalla copia di interi processi tra la memoria e lo spazio di swap. La maggior parte dei sistemi operativi combina ora lo swapping con la sostituzione di pagina, una tecnica che descriviamo in dettaglio nel resto di questo paragrafo.

10.4.1 Sostituzione di pagina

La sostituzione delle pagine segue il seguente criterio: se nessun frame è libero, ne viene liberato uno attualmente inutilizzato. È possibile liberarlo scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine (e tutte le altre tabelle) per indicare che la pagina non si trova più in memoria (Figura 10.10). Il frame liberato si può usare per memorizzare la pagina che ha causato il fault. Si modifica la procedura di servizio dell'eccezione di page fault in modo da includere la sostituzione della pagina:



1. s'individua la locazione su disco della pagina richiesta;
2. si cerca un frame libero:
 - a. se esiste, lo si usa;
 - b. altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un frame vittima;
 - c. si scrive la pagina "vittima" nel disco; si di conseguenza le tabelle delle pagine e quelle dei frame;
 - d. si scrive la pagina richiesta nel frame appena liberato; si modificano le tabelle delle pagine e dei frame;
4. si riprende il processo dal punto in cui si è verificato il page fault.

Occorre notare che, se non esiste alcun frame libero sono necessari *due* trasferimenti di pagine, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e aumenta di conseguenza anche il tempo effettivo d'accesso.

Questo sovraccarico si può ridurre usando un bit di modifica (*modify bit* o *dirty bit*). In questo caso l'hardware del calcolatore dispone di un bit di modifica, associato a ogni pagina (o frame), che viene posto a 1 ogni volta che nella pagina si scrive un byte, indicando che la pagina è stata modificata. Quando si sceglie una pagina da sostituire si esamina il suo bit di modifica; se è a 1, significa che quella pagina è stata modificata rispetto a quando era stata letta dal disco; in questo caso la pagina deve essere scritta nel disco. Se il bit di modifica è rimasto a 0, significa che la pagina *non* è stata modificata da quando è stata caricata in memoria, quindi non è necessario scrivere nel disco la pagina di memoria: c'è già. Questa tecnica vale anche per le pagine di sola lettura, per esempio pagine di codice binario. Queste pagine non possono essere modificate, quindi si possono rimuovere in ogni momento. Questo schema può ridurre in modo considerevole il tempo per il servizio del page fault, poiché dimezza il tempo di i/o, *se* la pagina non è stata modificata.

La sostituzione di una pagina è fondamentale al fine della paginazione su richiesta, perché completa la separazione tra memoria logica e memoria fisica. Con questo meccanismo si può mettere a disposizione dei programmati una memoria virtuale enorme con una memoria fisica più piccola. Senza la paginazione su richiesta, gli indirizzi utente si fanno corrispondere a indirizzi fisici e i due insiemi di indirizzi possono essere diversi. Tuttavia tutte le pagine di un processo devono ancora essere in memoria fisica. Con la paginazione su richiesta la dimensione dello spazio degli indirizzi logici non è più limitata dalla memoria fisica. Per esempio, un processo utente formato da 20 pagine si può eseguire in 10 frame semplicemente usando la paginazione su richiesta e un algoritmo di sostituzione per localizzare un frame libero ogni qual volta sia necessario. Se una pagina modificata deve essere sostituita, si copia nel

disco il suo contenuto. Un successivo riferimento a quella pagina causa un'eccezione di page fault. In quel momento, la pagina viene riportata in memoria, eventualmente sostituendo un'altra pagina.

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un algoritmo di allocazione dei frame e un algoritmo di sostituzione delle pagine. Ossia, se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire. La progettazione di algoritmi idonei a risolvere questi problemi è un compito importante, poiché l'i/o nei dischi è molto oneroso. Anche miglioramenti minimi ai metodi di paginazione su richiesta apportano notevoli incrementi alle prestazioni del sistema.

Esistono molti algoritmi di sostituzione delle pagine; probabilmente ogni sistema operativo ha il proprio schema di sostituzione. È quindi necessario stabilire un criterio per selezionare un algoritmo di sostituzione particolare; comunemente si sceglie quello con il minimo tasso di page fault.

Un algoritmo si valuta effettuandone l'esecuzione su una particolare successione di riferimenti alla memoria e calcolando il numero di page fault. La successione dei riferimenti alla memoria è detta, appunto, successione dei riferimenti. Queste successioni si possono generare artificialmente (per esempio con un generatore di numeri casuali), oppure analizzando un dato sistema e registrando l'indirizzo di ciascun riferimento alla memoria. Quest'ultima opzione genera un numero elevato di dati, dell'ordine di un milione di indirizzi al secondo. Per ridurre questa quantità di dati occorre notare due fatti.

Innanzitutto, per una pagina di dimensioni date, generalmente fissate dall'architettura del sistema, si considera solo il numero della pagina anziché l'intero indirizzo. In secondo luogo, se si ha un riferimento a una pagina p , i riferimenti alla stessa pagina immediatamente successivi al primo non generano page fault: dopo il primo riferimento, la pagina p è presente in memoria.

Esaminando un processo si potrebbe per esempio registrare la seguente successione di indirizzi:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

che, a 100 byte per pagina, si riduce alla seguente successione di riferimenti:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Per stabilire il numero di page fault relativo a una particolare successione di riferimenti e a un particolare algoritmo di sostituzione delle pagine, occorre conoscere anche il numero dei frame disponibili. Naturalmente, aumentando il numero di quest'ultimi diminuisce il numero di page fault. Per la successione dei riferimenti precedentemente esaminata, per esempio, dati tre o più blocchi di memoria avremmo solo tre fault: uno per il primo riferimento di ogni pagina. D'altra parte, se si dispone di un solo frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 page fault. In generale ci si aspetta una curva simile a quella della Figura 10.11. Aumentando il numero dei frame, il numero di page fault diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

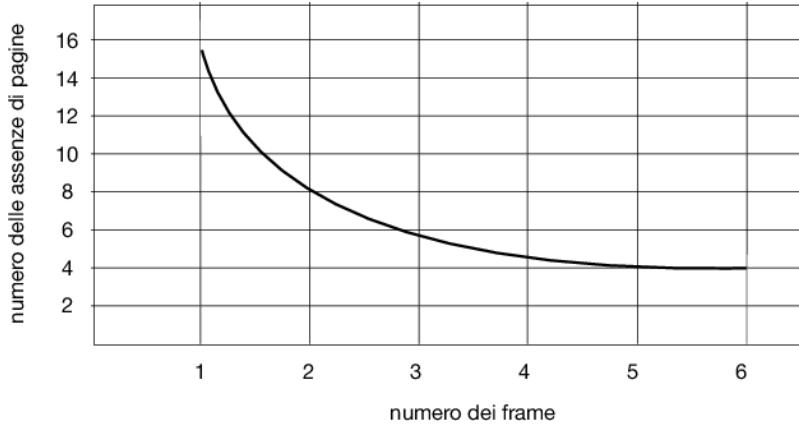


Figura 10.11 Grafico che illustra il numero di page fault rispetto al numero dei frame.

Per illustrare gli algoritmi di sostituzione delle pagine impiegheremo la seguente successione di riferimenti

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

per una memoria con tre frame.

10.4.2 Sostituzione delle pagine secondo l'ordine d'arrivo (fifo)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo fifo. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si può creare una coda fifo di tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova in testa alla coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

Nella successione di riferimenti di esempio, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) accusano ciascuno un page fault con conseguente caricamento delle relative pagine nei frame vuoti. Il riferimento successivo (2) causa la sostituzione della pagina 7, perché essa è stata caricata per prima in memoria. Siccome 0 è il riferimento successivo e si trova già in memoria, per questo riferimento non ha luogo alcun page fault. Il primo riferimento a 3 causa la sostituzione della pagina 0, che ora è la prima pagina in coda. A causa di questa sostituzione il riferimento successivo, a 0, causerà un page fault. La pagina 1 è poi sostituita dalla pagina 0. Questo processo prosegue come è illustrato nella Figura 10.12. Ogni volta che si verifica un page fault sono indicate le pagine presenti nei tre frame. Complessivamente si hanno 15 page fault.

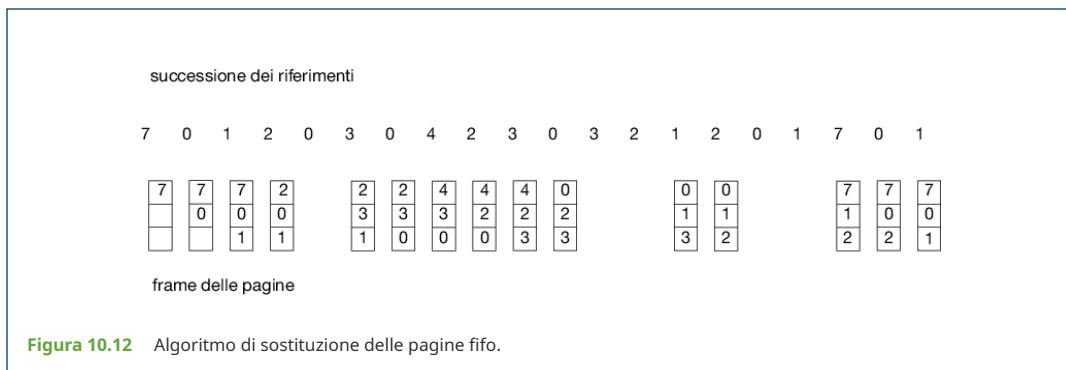


Figura 10.12 Algoritmo di sostituzione delle pagine fifo.

L'algoritmo fifo di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di page fault per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, una cattiva scelta della pagina da sostituire aumenta il tasso di page fault e rallenta l'esecuzione del processo, ma non causa errori.

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine fifo, si consideri la seguente successione di riferimenti:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Nella Figura 10.13 è illustrata la curva dei page fault per questa successione di riferimenti in funzione del numero dei frame disponibili. Occorre notare che il numero dei page fault (10) per quattro frame è maggiore del numero dei page fault (9) per tre frame. Questo inatteso risultato è noto col nome di anomalia di Belady: con alcuni algoritmi di sostituzione delle pagine, il tasso di page fault può aumentare con il tasso minimo di page fault aumentare del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. In alcune delle prime ricerche sperimentali si notò invece che questo presupposto non sempre è vero; venne così individuata l'anomalia di Belady.

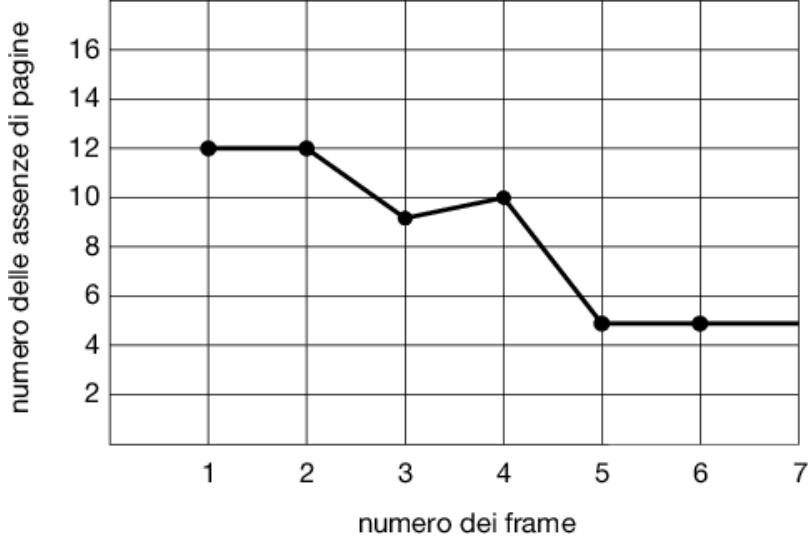


Figura 10.13 Curva dei page fault per la sostituzione fifo su una successione di riferimenti.

10.4.3 Sostituzione ottimale delle pagine

In seguito alla scoperta dell'anomalia di Belady si è ricercato un algoritmo ottimale di sostituzione delle pagine. Tale algoritmo è quello che fra tutti gli algoritmi presenta il tasso minimo di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato opt o min. Consiste semplicemente nel:

sostituire la pagina che non verrà usata per il periodo di tempo più lungo.

L'uso di quest'algoritmo di sostituzione delle pagine assicura il tasso minimo di page fault per un dato numero di frame.

Per esempio, nella successione dei riferimenti d'esempio, l'algoritmo ottimale di sostituzione delle pagine produce nove page fault, come è mostrato nella Figura 10.14. I primi tre riferimenti causano page fault che riempiono i tre blocchi di memoria vuoti. Il riferimento alla pagina 2 determina la sostituzione della pagina 7, perché la 7 non è usata fino al riferimento 18, mentre la pagina 0 viene usata al 5 e la pagina 1 al 14. Il riferimento alla pagina 3 causa la sostituzione della pagina 1, poiché la pagina 1 è l'ultima delle tre pagine in memoria cui si fa nuovamente riferimento. Con sole nove page fault, la sostituzione ottimale risulta assai migliore di quella ottenuta con un algoritmo fifo, dove i page fault erano 15. Ignorando i primi tre page fault, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore rispetto all'algoritmo fifo; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove page fault.

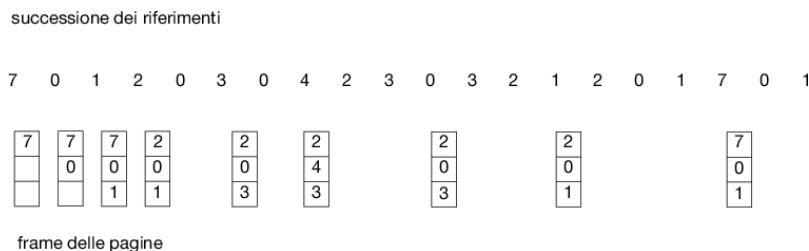


Figura 10.14 Algoritmo ottimale di sostituzione delle pagine.

Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti (una situazione analoga si è riscontrata con l'algoritmo sjf di scheduling della cpu, nel Paragrafo 5.3.2). Quindi, l'algoritmo ottimale si impiega soprattutto per studi comparativi. Per esempio, può risultare abbastanza utile sapere che, sebbene un algoritmo nuovo non sia ottimale, nel peggiore dei casi le sue prestazioni sono inferiori del 12,3 per cento rispetto a quelle dell'algoritmo ottimale, e mediamente questa percentuale è del 4,7 per cento.

10.4.4 Sostituzione delle pagine usate meno recentemente (lru)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi fifo e opt, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo fifo impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo opt impiega l'istante in cui una pagina sarà *usata*. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che *non è stata usata* per il periodo più lungo. Il metodo appena descritto è noto come algoritmo lru (*least recently used*).

La sostituzione lru associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo lru sceglie quella che non è stata usata per il periodo più lungo. Possiamo interpretare questa strategia come l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Un risultato interessante per l'analogia dei due algoritmi è il seguente. Supponendo che S^R sia la successione inversa di una successione di riferimenti S , il tasso di page fault per l'algoritmo opt su S è uguale a quello su S^R . Allo stesso modo, il tasso di page fault per l'algoritmo lru su S è uguale a quella su S^R .

Il risultato dell'applicazione dell'algoritmo lru alla successione dei riferimenti dell'esempio è illustrato nella Figura 10.15. L'algoritmo lru produce 12 page fault. Occorre notare che i primi cinque page fault sono gli stessi della sostituzione ottimale. Quando si presenta il riferimento alla pagina 4, però, l'algoritmo lru trova che, fra i tre blocchi di memoria, quello usato meno recentemente è della pagina 2. Quindi, l'algoritmo lru sostituisce la pagina 2 senza sapere che sta per essere usata. Quando si verifica il fault della pagina 2, l'algoritmo lru sostituisce la pagina 3, poiché, fra le tre pagine in memoria (0, 3, 4), la pagina 3 è quella usata meno recentemente. Nonostante questi problemi, la sostituzione lru, con 12 page fault, è molto migliore della sostituzione fifo, con 15 page fault.

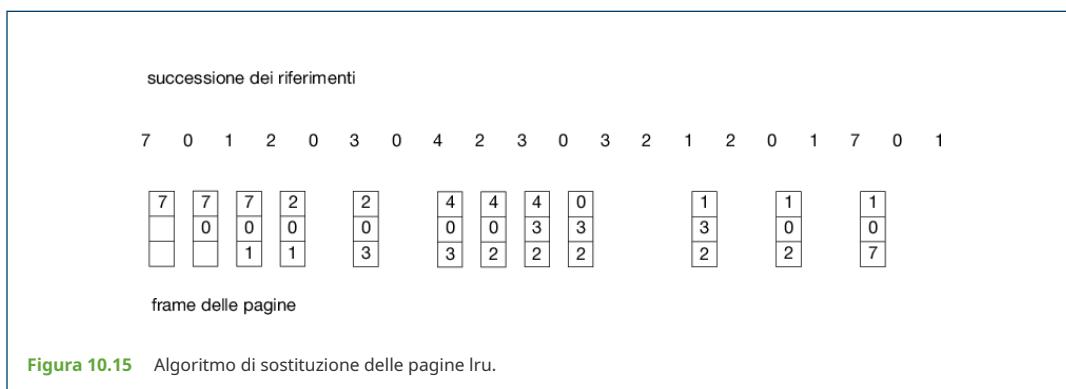


Figura 10.15 Algoritmo di sostituzione delle pagine lru.

Il criterio lru si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la sua implementazione. Un algoritmo di sostituzione delle pagine lru può richiedere una notevole assistenza da parte dell'hardware. Il problema consiste nel determinare un ordine per i frame definito dal momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni.

- Contatori. Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo *momento di utilizzo*, e alla cpu si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo *momento di utilizzo* nella voce della page table relativa a quella pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina usata meno recentemente (lru), e una scrittura in memoria (nel campo *momento di utilizzo* della tabella delle pagine) per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della cpu, si modificano le tabelle delle pagine. Occorre infine considerare l'overflow del contatore.
- Stack. Un altro metodo per la realizzazione della sostituzione delle pagine lru prevede l'utilizzo di uno stack dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest'ultimo. In questo modo, in cima allo stack si trova sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente, com'è illustrato dalla Figura 10.16. Poiché gli elementi si devono estrarre dal centro dello stack, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dallo stack e collocarla in cima, nel caso peggiore è necessario modificare sei puntatori. Ogni aggiornamento è un po' più costoso, ma per una sostituzione non si deve compiere alcuna ricerca: il puntatore dell'elemento di coda punta alla pagina lru. Questo metodo è adatto soprattutto alle realizzazioni programmate (o microprogrammate) della sostituzione lru.

successione dei riferimenti

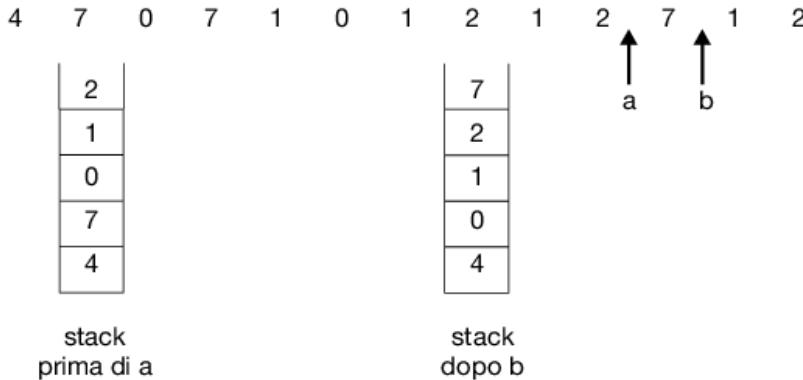


Figura 10.16 Uso di uno stack per registrare i più recenti riferimenti alle pagine.

Né la sostituzione ottimale né quella lru sono soggette all'anomalia di Belady. Entrambe appartengono a una classe di algoritmi di sostituzione delle pagine, chiamati algoritmi a stack, che non presenta l'anomalia di Belady. Un algoritmo a stack è un algoritmo per il quale è possibile mostrare che l'insieme delle pagine in memoria per n frame è sempre un *sottoinsieme* dell'insieme delle pagine che dovrebbero essere in memoria per $n + 1$ frame. Per la sostituzione lru, l'insieme di pagine in memoria è costituito delle n pagine cui si è fatto riferimento più recentemente. Se il numero dei frame è aumentato, queste n pagine continuano a essere quelle cui si è fatto riferimento più recentemente e quindi restano in memoria.

Si noti che senza un supporto hardware (aggiuntivo rispetto ai registri tlb standard) sarebbero inconcepibili entrambe le implementazioni della sostituzione lru. L'aggiornamento dei campi del contatore o dello stack si deve effettuare per *ogni* riferimento alla memoria. Se per ogni riferimento si dovesse adoperare un'interruzione per permettere al sistema operativo di modificare tali strutture dati, tutti i riferimenti alla memoria sarebbero rallentati di un fattore almeno pari a 10, quindi anche tutti i processi utenti sarebbero rallentati di un uguale fattore. Pochi sistemi potrebbero permettere un tale sovraccarico per la gestione della memoria.

10.4.5 Sostituzione delle pagine per approssimazione a lru

Sono pochi i sistemi di calcolo che dispongono del supporto hardware per una vera sostituzione lru delle pagine. Nei sistemi che non offrono alcun supporto hardware si devono impiegare altri algoritmi di sostituzione delle pagine, per esempio l'algoritmo fifo. Molti sistemi tuttavia possono fornire un aiuto: un bit di riferimento. Il bit di riferimento a una pagina è impostato automaticamente dall'hardware del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzerà tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'hardware imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'*ordine* d'uso. Questa informazione è alla base di molti algoritmi per la sostituzione delle pagine che approssimano lru.

10.4.5.1 Algoritmo con bit supplementari di riferimento

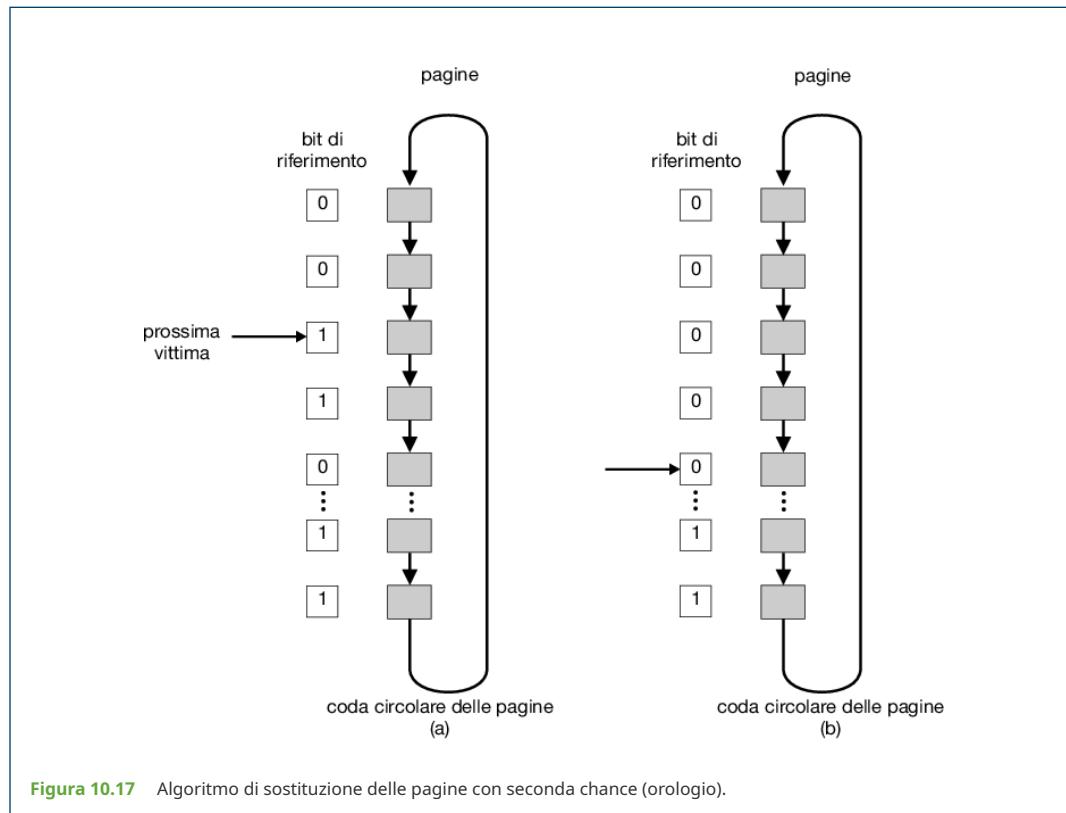
Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria con, per esempio, un byte per ogni pagina. A intervalli regolari, per esempio di 100 millisecondi, un segnale d'interruzione del timer trasferisce il controllo al sistema operativo. Questo inserisce il bit di riferimento per ciascuna pagina nel bit più significativo del byte, shiftando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento di 8 bit contengono la storia dell'utilizzo delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene la successione di bit 00000000, significa che la pagina associata non è stata usata da otto periodi di tempo; a una pagina usata almeno una volta per ogni periodo corrisponde la successione 11111111 nel registro a scorrimento. Una pagina cui corrisponde la successione 11000100, è stata usata più recentemente di una pagina a cui corrisponde 01110111. Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina lru, e può essere sostituita. Si noti che l'unicità dei numeri non è garantita. Si possono sostituire tutte le pagine con il valore minore, oppure si può ricorrere a una selezione fifo.

Il numero dei bit può ovviamente essere variato: si sceglie secondo l'hardware disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento. In questo caso l'algoritmo è noto come algoritmo di sostituzione delle pagine con seconda chance.

10.4.5.2 Algoritmo con seconda chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo fifo. Tuttavia, dopo aver selezionato una pagina, si controlla il bit di riferimento: se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e si passa alla successiva pagina fifo. Quando una pagina riceve la seconda chance, si azzerza il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non siano state sostituite, oppure non sia stata data loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, da mantenere il suo bit di riferimento impostato a 1, non viene mai sostituita.

Un metodo per implementare l'algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull'uso di una coda circolare, in cui un puntatore (lancetta) indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzerà il bit di riferimento appena esaminato (Figura 10.17). Una volta trovata una pagina "vittima", la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione fifo.



10.4.5.3 Algoritmo con seconda chance migliorato

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica (descritto nel Paragrafo 10.4.1) come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

1. (0, 0) né recentemente usato né modificato – migliore pagina da sostituire;
2. (0, 1) non usato recentemente, ma modificato – la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
3. (1, 0) usato recentemente ma non modificato – probabilmente la pagina sarà presto usata nuovamente;
4. (1, 1) usato recentemente e modificato – probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esamina la classe a cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che si può dover scandire la coda circolare più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo a orologio è che qui si dà la preferenza alle pagine modificate, al fine di ridurre il numero di i/o richiesti.

10.4.6 Sostituzione delle pagine basata su conteggio

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Per esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi.

- Algoritmo di sostituzione delle pagine meno frequentemente usate (*least frequently used*, lfu); richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Si ha però un problema quando una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Poiché è stata usata intensamente il suo conteggio è alto, quindi rimane in memoria anche se non è più necessaria. Una soluzione può essere quella di spostare i valori dei contatori a destra di un bit a intervalli regolari, misurando l'utilizzo con un peso esponenziale decrescente.
- Algoritmo di sostituzione delle pagine più frequentemente usate (*most frequently used*, mfu); è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Le sostituzioni mfu e lfu non sono molto comuni, poiché la realizzazione di questi algoritmi è abbastanza onerosa; inoltre, tali algoritmi non approssimano bene la sostituzione opt.

10.4.7 Algoritmi con buffering delle pagine

Oltre a uno specifico algoritmo per la sostituzione delle pagine, si usano spesso anche altre procedure; per esempio, i sistemi hanno generalmente un gruppo di frame liberi (*pool of free frames*). Quando si verifica un page fault, si sceglie innanzitutto un frame vittima, ma prima che la vittima sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame al gruppo dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogniqualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si resetta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Un'altra modifica consiste nell'usare un gruppo di frame liberi, ma ricordare quale pagina era contenuta in ciascun frame. Poiché quando il contenuto di un frame viene scritto su disco tale contenuto non cambia, la vecchia pagina è ancora utilizzabile prendendola dal gruppo dei frame liberi, se ce n'è bisogno prima che sia riusato quel frame. In questo caso non è necessario alcun i/o. Se si verifica un page fault si controlla prima se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Alcune versioni di unix adottano questo metodo insieme all'algoritmo con seconda chance. In effetti, si tratta di un'utile integrazione a qualunque algoritmo di sostituzione, al fine di ridurre il prezzo pagato per l'eventuale errata scelta della pagina vittima. Queste e altre modifiche sono descritte nel Paragrafo 10.5.3.

10.4.8 Applicazioni e sostituzione della pagina

In taluni casi, le applicazioni che accedono ai dati tramite la memoria virtuale del sistema operativo hanno prestazioni peggiori di quelle che avrebbero se il sistema operativo non offrisse alcun buffering. Si pensi, quale esempio tipico, a un database che gestisce la memoria e il buffering dell'i/o in modo autonomo. Applicazioni come questa capiscono il proprio utilizzo della memoria e del disco meglio di quanto possa fare un sistema operativo, che applica algoritmi adatti a un uso generale. Se il sistema operativo adotta un buffer per l'i/o, e così pure fa l'applicazione, la quantità di memoria necessaria per l'i/o sarà inutilmente raddoppiata.

Un altro esempio proviene dai data warehouse, che effettuano spesso lunghe letture sequenziali del disco, seguite da calcoli e scritture. L'algoritmo lru eliminerebbe le pagine vecchie per conservare le nuove, mentre in questo caso è ragionevole attendersi la lettura delle pagine vecchie in luogo di quelle nuove (quando l'applicazione inizia nuovamente la lettura sequenziale). In queste circostanze l'algoritmo mfu sarebbe più efficiente di lru.

Per risolvere tali problemi, alcuni sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto disco di basso livello (*raw disk*), e il relativo i/o è denominato i/o di basso livello (*raw i/o*). L'i/o di basso livello bypassa tutti i servizi del file system, come la paginazione su richiesta dell'i/o su file, il locking dei file, il prefetching, l'allocazione dello spazio, la gestione dei nomi dei file e le directory. Si noti però che, sebbene alcune applicazioni siano più efficienti quando gestiscono i propri servizi specifici di memorizzazione sul disco di basso livello, quasi tutte hanno una resa migliore quando operano con i servizi regolari del file system.

10.5 Allocazione dei frame

Consideriamo ora il problema dell'allocazione. Occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, se abbiamo 93 frame liberi e due processi, quanti frame assegnamo a ciascuno?

Si consideri un sistema che disponga di 128 frame. Complessivamente sono presenti 128 frame. Il sistema operativo può occuparne 35, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di page fault. I primi 93 page fault ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la novantaquattresima, si può usare un algoritmo di sostituzione delle pagine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Vi sono molte variazioni di questa semplice strategia. Si può richiedere che il sistema operativo assegna tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un page fault sia sempre disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento, si può scegliere una pagina da rimpiazzare, che viene poi scritta nel disco mentre il processo utente continua l'esecuzione. Sono possibili anche altre varianti, ma la strategia di base è chiara: al processo utente si assegna qualsiasi frame libero.

10.5.1 Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Esaminiamo quest'ultimo requisito in maggiore dettaglio.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati a ciascun processo aumenta il tasso di page fault, con conseguente rallentamento dell'esecuzione dei processi. Inoltre va ricordato che, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata. Di conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.

Si consideri, per esempio, un calcolatore in cui tutte le istruzioni di riferimento alla memoria hanno solo un indirizzo di memoria; in questo caso occorre almeno un frame per l'istruzione e uno per il riferimento alla memoria. Inoltre se è ammesso un indirizzamento indiretto a un livello (come nel caso di un'istruzione `load` presente nella pagina 16 che può far riferimento a un indirizzo della pagina 0, che costituisce a sua volta un riferimento indiretto alla pagina 23) la paginazione richiede allora almeno tre frame per ogni processo. Si immagini che cosa accadrebbe nel caso di un processo che disponga di due soli frame.

Il numero minimo di frame è definito dall'architettura del calcolatore. Per esempio, se l'istruzione di `move` in una data architettura, per alcune modalità di indirizzamento, è costituita da più di una parola, la stessa istruzione può stare a cavallo tra due pagine. Inoltre, ciascuno dei suoi due operandi può essere un riferimento indiretto, per un totale di sei frame. Come altro esempio, l'istruzione di `move` su architetture Intel a 32 e 64 bit consente di spostare un dato solamente da registro a registro o tra registro e memoria, ma non permette lo spostamento diretto da memoria a memoria, limitando in questo modo il numero minimo di frame per un processo. Il numero minimo di frame per ciascun processo è definito dall'architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. In mezzo vi è un ampio spazio di scelta.

10.5.2 Algoritmi di allocazione

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà una parte uguale, m/n frame (ignorando per ora i frame di cui il sistema operativo ha bisogno). Dati 93 frame e cinque processi, ogni processo riceve 18 frame. I tre frame lasciati liberi si potrebbero usare come buffer di frame liberi. Questo schema è chiamato allocazione uniforme.

Un'alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 kb. Se un piccolo processo utente di 10 kb e una base di dati interattiva di 127 kb sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati.

Per risolvere questo problema è possibile ricorrere all'allocazione proporzionale, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Si supponga che s_i sia la dimensione della memoria virtuale per il processo p_i . Si definisce la seguente quantità:

$$S = \sum s_i.$$

Quindi, se il numero totale dei frame disponibili è m , al processo p_i si assegnano a_i frame, dove a_i è approssimativamente

$$a_i = s_i / S \times m.$$

Naturalmente è necessario scegliere ciascun a_i in modo che sia un intero maggiore del numero minimo di frame richiesti dall'instruction set e in modo che la somma di tutti gli a_i non sia maggiore di m .

Usando l'allocazione proporzionale, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti:

$$10/137 \times 62 \approx 4 \quad 127/137 \times 62 \approx 57.$$

In questo modo entrambi i processi condividono i frame disponibili secondo le rispettive necessità, e non in modo uniforme.

Sia nell'allocazione uniforme sia in quella proporzionale, l'allocazione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo rimosso si possono distribuire tra quelli che restano.

Occorre notare che sia con l'allocazione uniforme sia con l'allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l'uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

10.5.3 Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: sostituzione globale e sostituzione locale. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall'insieme di tutti i frame, anche se quel frame è al momento allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri per esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Un processo può scegliere per la sostituzione uno dei suoi rame o uno di quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito dei processi a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale, invece, può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché altri non scelgano per la sostituzione i *suo*i frame.

L'algoritmo di sostituzione globale risente di un problema: un processo non può controllare il proprio tasso di page fault, infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso processo può comportarsi in modi molto diversi, per esempio impiegando 0,5 secondi per un'esecuzione e 4,3 secondi per quella successiva, a causa di circostanze del tutto esterne. Con l'algoritmo di sostituzione locale questo problema non si presenta. Infatti l'insieme di pagine in memoria per un processo subisce l'effetto del comportamento di paginazione di quel solo processo. Tuttavia la sostituzione locale può penalizzare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

Ci soffermeremo ora su una possibile strategia per implementare una politica globale di sostituzione delle pagine. In questo approccio soddisfiamo tutte le richieste di memoria mediante la lista dei frame liberi, ma piuttosto che aspettare che la lista si svuoti prima di iniziare a selezionare le pagine per la sostituzione, attiviamo la sostituzione delle pagine quando la dimensione della lista scende al di sotto di una certa soglia. Questa strategia cerca di garantire che ci sia sempre sufficiente memoria libera per soddisfare nuove richieste.

La strategia è illustrata nella Figura 10.18. Come accennato, lo scopo è di mantenere la quantità di memoria libera al di sopra di una soglia minima: quando si scende al di sotto di questa soglia, viene avviata una routine del kernel che inizia a recuperare pagine da tutti i processi nel sistema (in genere escludendo il kernel). Queste routine del kernel sono note come reaper ("mietitrici") e possono applicare qualsiasi algoritmo di sostituzione delle pagine tra quelli trattati nel Paragrafo 10.4. Quando la quantità di memoria libera raggiunge la soglia massima, la routine reaper viene sospesa, per poi essere riavviata quando la memoria libera scende nuovamente al di sotto della soglia minima.

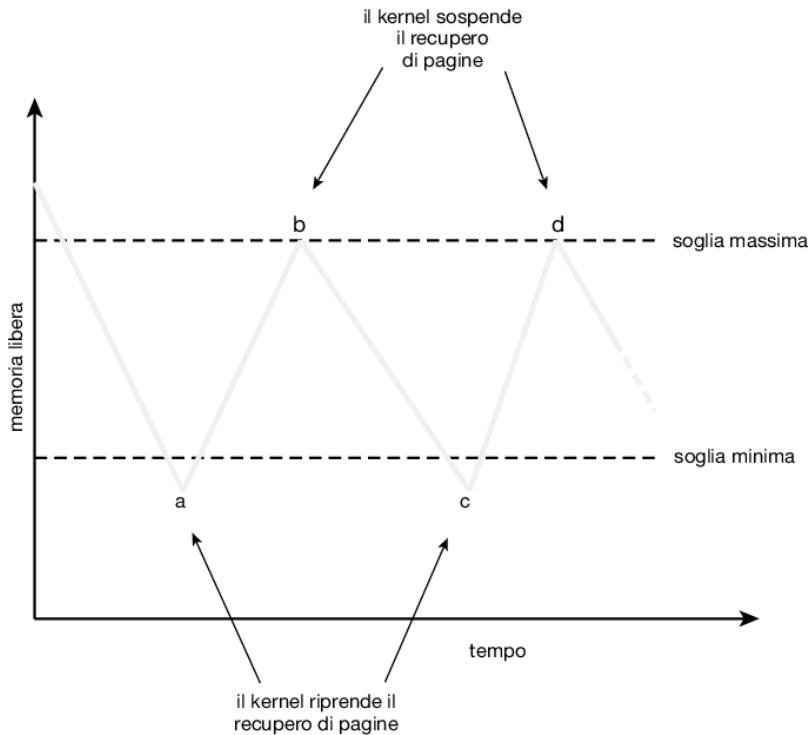


Figura 10.18 Recupero di pagine.

PAGE FAULT PRINCIPALI E SECONDARI

Come descritto nel Paragrafo 10.2.1, si verifica un page fault quando una pagina non ha una mappatura valida nello spazio degli indirizzi di un processo. I sistemi operativi distinguono generalmente tra due tipi di **page fault: principali, o major, e secondari, o minor** (Windows si riferisce ai fault principali e secondari rispettivamente come fault hard e soft). Un page fault principale si verifica quando si fa riferimento a una pagina che non è in memoria. Per risolvere un page fault principale occorre caricare la pagina desiderata dalla memoria ausiliaria in un frame libero e aggiornare la tabella delle pagine. La paginazione su richiesta genera di solito diversi page fault principali nella fase iniziale.

I page fault secondari si verificano quando un processo non ha una mappatura logica per una pagina, anche se la pagina è in memoria. I page fault secondari possono verificarsi per uno dei due seguenti motivi. Una prima possibilità si ha quando un processo fa riferimento a una libreria condivisa presente in memoria, ma non ha una mappatura alla libreria nella sua tabella delle pagine. In questo caso è necessario solamente aggiornare la tabella delle pagine per fare riferimento alla pagina esistente in memoria. Una seconda causa di page fault secondario si ha quando una pagina viene tolta a un processo e inserita nella lista dei frame liberi, ma la pagina non è stata ancora azzerata e allocata a un altro processo. Quando questo tipo di errore si verifica, il frame può essere rimosso dalla lista dei frame liberi e riassegnato al processo. Come ci si può aspettare, la risoluzione di un page fault secondario è in genere molto meno dispendiosa in termini di tempo rispetto alla risoluzione di un page fault principale.

È possibile osservare il numero di page fault principali e secondari in un sistema Linux utilizzando il comando `ps -eo min_flt, maj_flt, cmd`, che restituisce il numero di page fault secondari e principali, insieme al comando che ha avviato il processo. Un output di esempio di questo comando ps è il seguente:

	MINFL	MAJFL	CMD
186509	32		/usr/lib/systemd/systemd-logind
76822	9		/usr/sbin/sshd -D
1937	0		vim 10.tex
699	14		/sbin/auditd -n

È interessante notare che per la maggior parte dei comandi il numero di page fault principali è in genere piuttosto basso, mentre il numero di page fault secondari è molto più elevato. Ciò indica che probabilmente i processi di Linux sfruttano molto le librerie condivise e una volta che una libreria è caricata in memoria, i seguenti page fault sono solo secondari.

Osserviamo la Figura 10.18. Nel punto *a* la quantità di memoria libera scende al di sotto della soglia minima e il kernel inizia a recuperare pagine e ad aggiungerle alla lista dei frame liberi. Questo procedimento continua fino al raggiungimento della soglia massima (punto *b*). Con il passare del tempo arrivano nuove richieste di memoria e al punto *c* la quantità di memoria libera scende nuovamente al di sotto della soglia minima: riprende quindi il recupero di pagine, che poi viene di nuovo sospeso quando la quantità di memoria libera raggiunge la soglia massima (punto *d*). Questo procedimento continua finché il sistema resta in esecuzione.

Come accennato in precedenza, la routine reaper del kernel può adottare qualsiasi algoritmo di sostituzione di pagina, ma in genere utilizza un algoritmo che approssima lru. Consideriamo ciò che potrebbe accadere quando la routine reaper non è in grado di mantenere la lista dei frame liberi al di sopra della soglia minima. In queste circostanze, la routine inizia a recuperare le pagine in modo più aggressivo, per esempio sospendendo l'algoritmo con seconda chance e utilizzando una tecnica fifo pura. Un altro esempio più estremo si può verificare in Linux: quando la quantità di memoria libera scende a livelli molto bassi, una routine nota come oom (Out-Of-Memory killer) seleziona un processo da terminare, liberando così la sua memoria. In che modo Linux determina quale processo terminare? Ogni processo è dotato del cosiddetto punteggio oom, dove un punteggio più alto aumenta la probabilità che il processo possa essere terminato dalla routine oom. I punteggi oom sono calcolati in base alla percentuale di memoria utilizzata da un processo: maggiore è la percentuale, maggiore è il punteggio oom (i punteggi oom possono essere visualizzati nel file system `/proc`, dove, per esempio, il punteggio per un processo con pid 2500 è visualizzabile in `/proc/2500/oom_score`).

In generale, non è soltanto l'aggressività delle routine reaper a variare, ma possono essere modificati anche i valori delle soglie minima e massima. Questi valori possono essere impostati su valori predefiniti, ma alcuni sistemi consentono a un amministratore di sistema di configurarli in base alla quantità di memoria fisica presente nel sistema.

10.5.4 Accesso non uniforme alla memoria

Fino a questo momento trattando il tema della memoria virtuale abbiamo assunto che le diverse parti della memoria centrale siano uguali, o almeno che vi si potesse accedere nello stesso modo. Su sistemi con accesso non uniforme alla memoria (numa) dotati di più cpu (Paragrafo 1.3.2) non è così. Su questi sistemi una determinata cpu può accedere ad alcune sezioni della memoria principale più velocemente di quanto possa accedere ad altre. Queste differenze di prestazioni sono causate dal modo in cui cpu e memoria sono interconnesse all'interno del sistema. Un sistema di questo tipo è costituito da più cpu, ciascuna con la propria memoria locale (Figura 10.19); le cpu sono organizzate utilizzando un'interconnessione di sistema condivisa e, come ci si potrebbe aspettare, una cpu può accedere più velocemente alla propria memoria locale rispetto alla memoria locale di un'altra cpu. I sistemi numa sono, senza eccezioni, più lenti dei sistemi in cui tutti gli accessi alla memoria principale sono trattati allo stesso modo. Tuttavia, come descritto nel Paragrafo 1.3.2, questi sistemi possono ospitare diverse cpu e raggiungere quindi livelli maggiori di throughput e parallelismo.

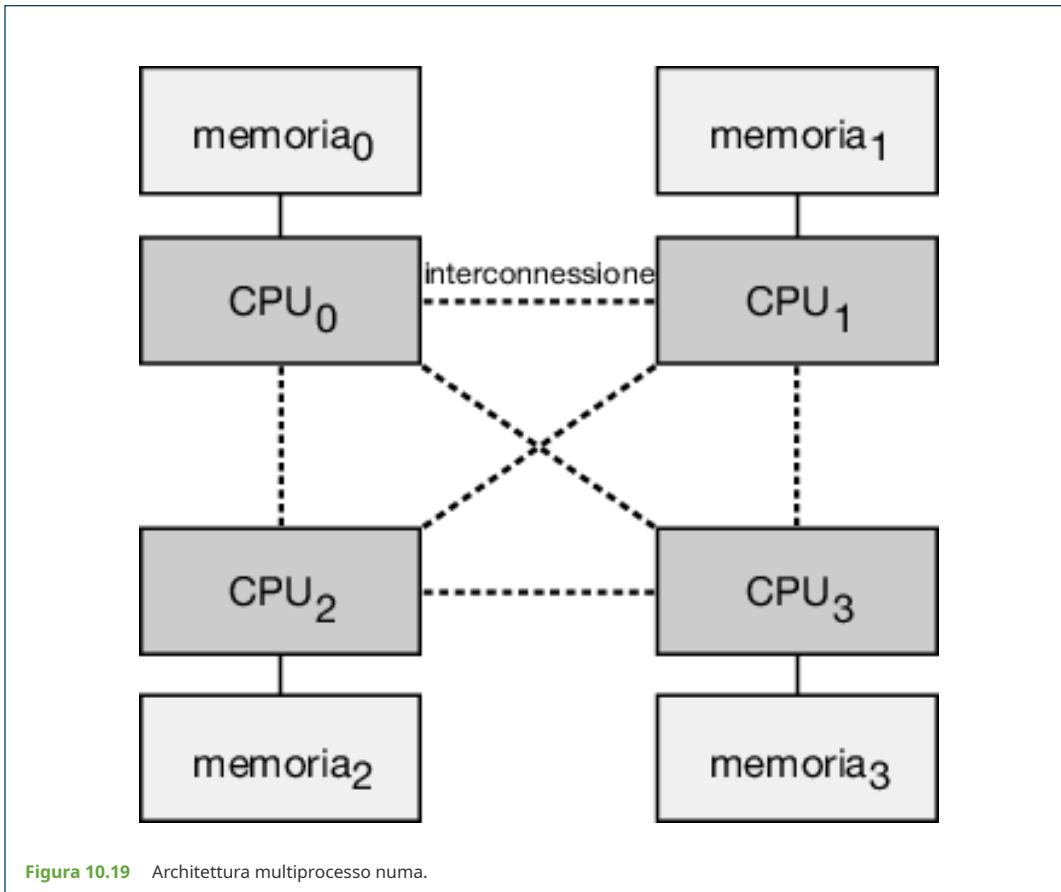


Figura 10.19 Architettura multiprocesso numa.

Le decisioni su quali frame di pagina memorizzare in quale posizione possono condizionare in modo significativo le prestazioni nei sistemi numa. Se, in sistemi del genere, consideriamo uniforme la memoria, i processori potrebbero dover aspettare molto più a lungo per accedere alla memoria rispetto al caso in cui gli algoritmi di allocazione della memoria siano modificati per tenere in conto il numa. Abbiamo descritto alcune di queste modifiche nel Paragrafo 5.5.4. Il loro obiettivo è quello di disporre di frame di memoria allocati “il più vicino possibile” alla cpu su cui è in esecuzione il processo (per “vicino” intendiamo “con latenza minima”, ovvero, di solito, sulla stessa scheda della cpu). Pertanto, quando un processo incorre in un page fault, un sistema di memoria virtuale consci del numa assegna a quel processo un frame il più vicino possibile alla cpu su cui è in esecuzione. Per tenere in considerazione numa, lo scheduler deve tenere traccia dell’ultima cpu su cui è stato eseguito ciascun processo. Se lo scheduler tenta di schedulare ciascun processo sulla cpu precedente e il sistema di gestione della memoria virtuale tenta di allocare i frame per il processo vicino alla cpu su cui il processo viene schedulato, si otterrà un incremento di cache hit e una diminuzione del tempo di accesso alla memoria.

La questione diventa ancora più complicata con l’aggiunta dei thread. Per esempio, un processo con molti thread in esecuzione potrebbe vedere quei thread schedulati su differenti schede del sistema. Come viene allocata la memoria in questo caso?

Come discusso nel Paragrafo 5.7.1 Linux gestisce questa situazione facendo in modo che il kernel identifichi una gerarchia di domini di scheduling. Lo scheduler cfs di Linux non consente ai thread di migrare su domini diversi e quindi incorre in penalità nell’accesso alla memoria. Linux, inoltre, ha una lista dei frame liberi distinta per ogni nodo numa, garantendo in tal modo che a un thread sia allocata memoria dal nodo su cui è in esecuzione. Solaris risolve il problema in modo simile creando le entità lgroup (“gruppi di località”) nel kernel. Ogni lgroup raccoglie alcune cpu e memoria e ogni cpu del gruppo può accedere alla memoria del gruppo entro un intervallo di latenza definito. Inoltre, esiste una gerarchia di lgroup basata sulla latenza tra i gruppi, simile alla gerarchia dei domini di scheduling in Linux. Solaris tenta di pianificare tutti i thread di un processo e di allocare tutta la memoria di un processo nell’ambito di un solo lgroup. Se una tale soluzione non è possibile, per il resto delle risorse necessarie vengono utilizzati gli lgroup più vicini, in modo da minimizzare la latenza complessiva della memoria e massimizzare il tasso di successo della cache del processore.

10.6 Thrashing

Si consideri un qualsiasi processo che non disponga di un numero di frame “sufficiente”. Se non vi sono abbastanza frame per ospitare le pagine del working set, il processo incorre rapidamente in un page fault.

A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono attive, si deve sostituire una pagina che sarà subito necessaria, e di conseguenza si verificano parecchi page fault poiché si sostituiscono pagine che saranno immediatamente riportate in memoria.

Questa intensa paginazione è nota come *thrashing*. Un processo in thrashing spende più tempo per la paginazione che per l'esecuzione dei processi. Com'è immaginabile, il thrashing causa gravi problemi di prestazioni.

10.6.1 Cause del thrashing

Il thrashing causa notevoli problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione.

Il sistema operativo controlla l'utilizzo della cpu. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di page fault, cui segue la sottrazione di frame ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi dei page fault, con conseguente sottrazione di frame ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della cpu diminuisce.

Lo scheduler della cpu rileva questa riduzione dell'utilizzo della cpu e *aumenta* il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori page fault e allungando la coda per il dispositivo di paginazione. Come risultato l'utilizzo della cpu scende ulteriormente, e lo scheduler della cpu tenta di aumentare ancora il grado di multiprogrammazione. Si è in una situazione di thrashing che fa precipitare la produttività del sistema. Il tasso dei page fault aumenta enormemente, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell'attività di paginazione.

Questo fenomeno è illustrato nella Figura 10.20, in cui si riporta l'utilizzo della cpu in funzione del grado di multiprogrammazione. Aumentando il grado di multiprogrammazione aumenta anche l'utilizzo della cpu, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della cpu. In questa situazione, per aumentare l'utilizzo della cpu e bloccare il thrashing occorre *ridurre* il grado di multiprogrammazione.

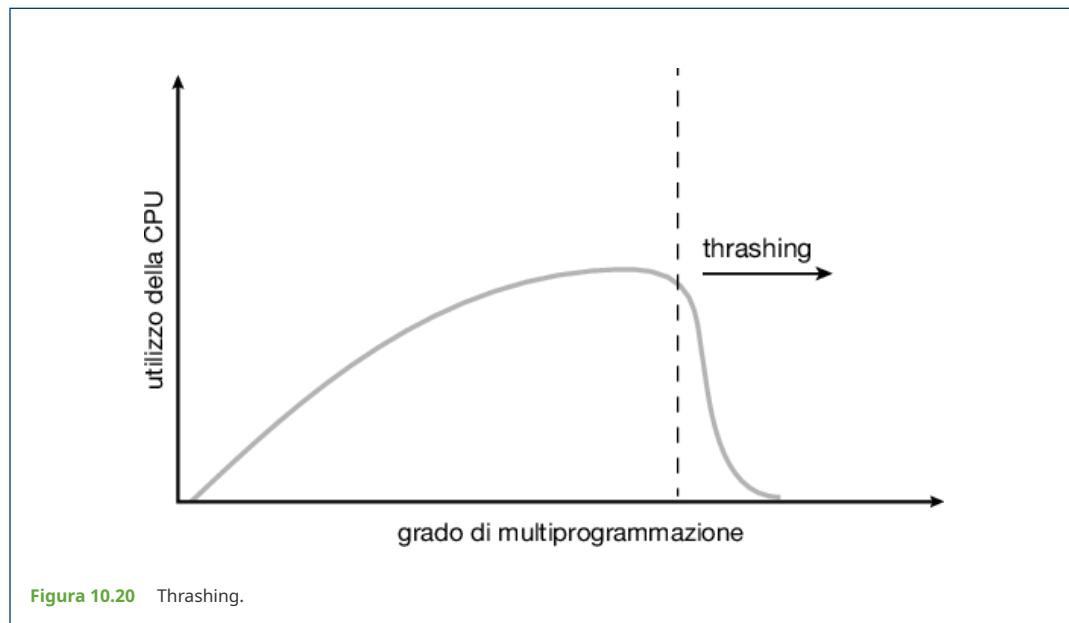


Figura 10.20 Thrashing.

Gli effetti di questa situazione si possono limitare usando un algoritmo di sostituzione locale, o algoritmo di sostituzione per priorità. Con la sostituzione locale, se un processo entra in thrashing, non può sottrarre frame a un altro processo e quindi provocarne a sua

volta la degenerazione. Tuttavia il problema non è completamente risolto. I processi in thrashing rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un page fault aumenta a causa dell'allungamento della coda media d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso in memoria aumenta anche per gli altri processi.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame "servano" a un processo si impiegano diverse tecniche. L'approccio del working-set comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo approccio definisce il modello di località d'esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente insieme. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili. Per esempio, quando s'invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali. Quando la procedura termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente. Potrà tornare più tardi in questa località.

La Figura 10.21 illustra il concetto di località e come la località di un processo cambia nel tempo. Al momento (a), la località è l'insieme di pagine {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. Al momento (b), la località cambia in {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Si noti la sovrapposizione: alcune pagine (per esempio, 18, 19 e 20) fanno parte di entrambe le località.

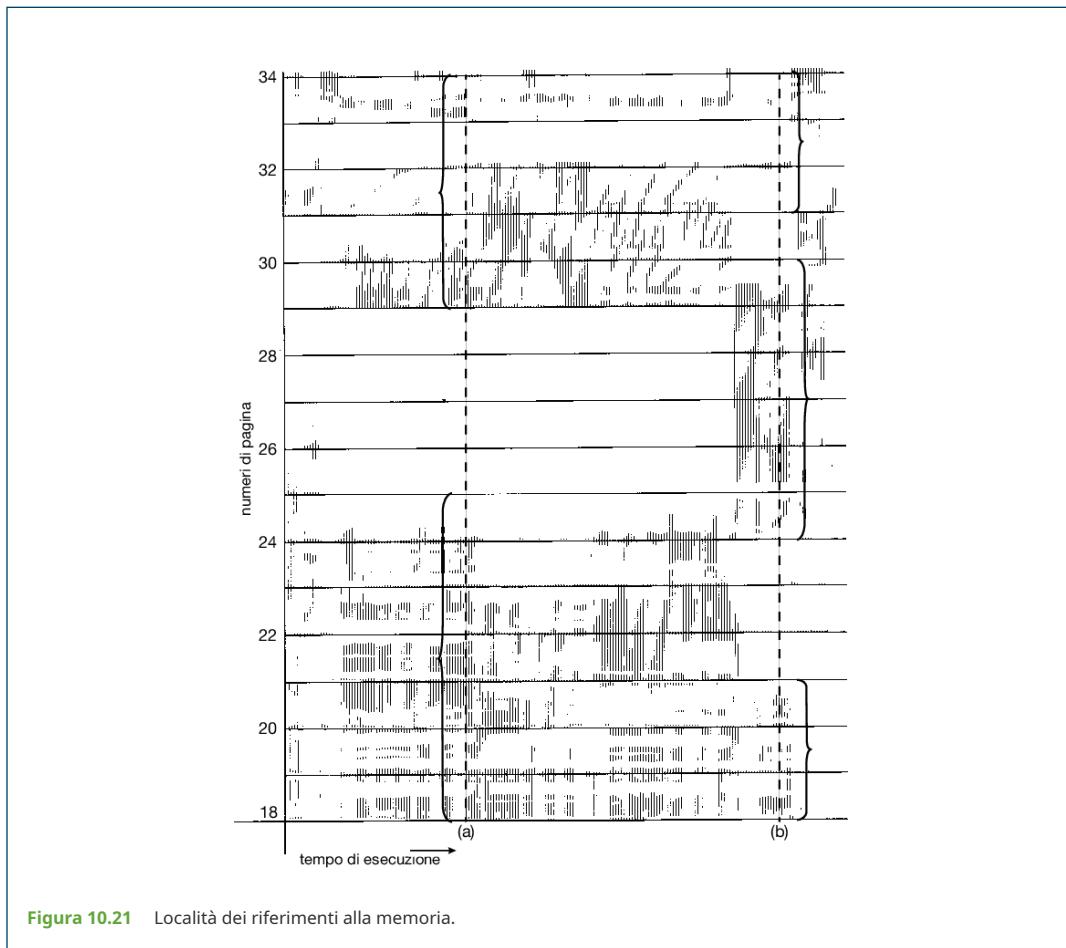


Figura 10.21 Località dei riferimenti alla memoria.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all'analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano le assenze delle pagine relative a tali località; quindi, finché le località non vengono modificate, non hanno luogo altri page fault. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

10.6.2 Modello del working set

Come già accennato, il modello del working set è basato sull'ipotesi di località. Questo modello usa un parametro, D , per definire la finestra del working set. L'idea consiste nell'esaminare i più recenti D riferimenti alle pagine. L'insieme di pagine nei più recenti D riferimenti è il working set (Figura 10.22). Se una pagina è in uso attivo si trova nel working set; se non è più usata esce dal working set D unità di tempo dopo il suo ultimo riferimento. Quindi, il working set non è altro che un'approssimazione della località del programma.

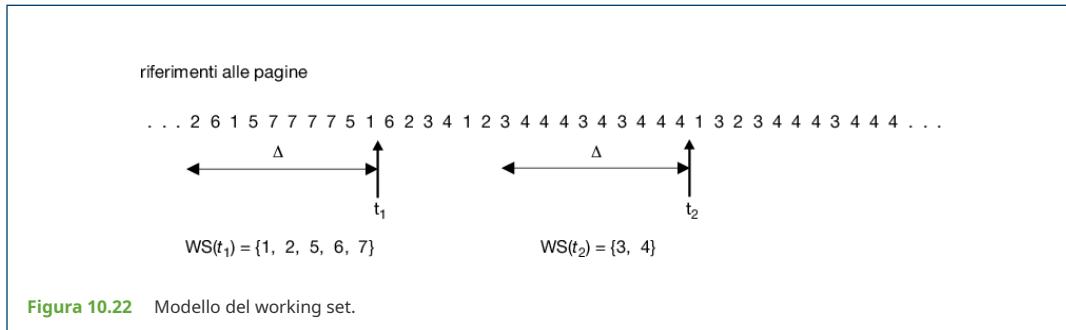


Figura 10.22 Modello del working set.

Per esempio, data la successione di riferimenti alla memoria mostrata nella Figura 10.22, se $D = 10$ riferimenti alla memoria, il working set all'istante t_1 è $\{1, 2, 5, 6, 7\}$. All'istante t_2 il working set è diventato $\{3, 4\}$.

La precisione del working set dipende dalla scelta del valore di D . Se D è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se D è infinito il working set coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione.

La caratteristica più importante del working set è la sua dimensione. Calcolandone la dimensione wss_i , per ciascun processo p_i del sistema, si può determinare la richiesta totale di frame, cioè D :

$$D = \sum wss_i$$

Ogni processo usa attivamente le pagine del proprio working set. Quindi, il processo i necessita di wss_i frame. Se la richiesta totale è maggiore del numero totale di frame liberi ($D > m$), si avrà thrashing, poiché alcuni processi non dispongono di un numero sufficiente di frame.

Una volta scelto D , l'uso del modello del working set è abbastanza semplice. Il sistema operativo controlla il working set di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo working set. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni dei working set aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i suoi frame ad altri processi. Il processo sospeso può essere ripreso successivamente.

Questa strategia impedisce il thrashing, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della cpu.

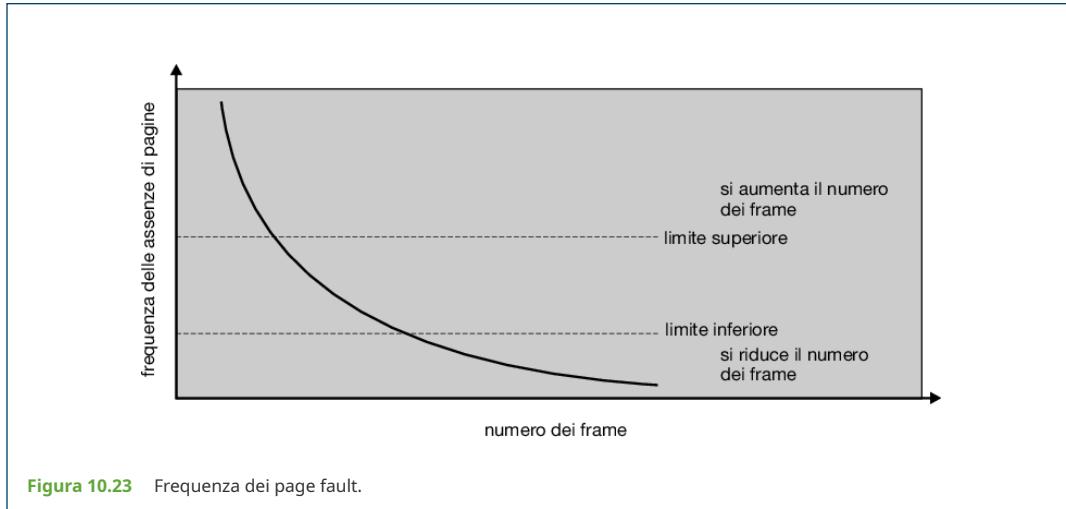
Poiché la finestra del working set è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono il working set stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nel working set se esiste un riferimento a essa in qualsiasi punto della finestra del working set.

Si può approssimare il modello con un interrupt da timer a intervalli fissi e un bit di riferimento. Si supponga, per esempio, che D sia pari a 10.000 riferimenti e che sia possibile ottenere un segnale d'interruzione dal timer ogni 5000 riferimenti. Quando si verifica uno di tali segnali d'interruzione, i valori dei bit di riferimento di ciascuna pagina vengono copiati in memoria e poi azzerati. Così, quando si verifica un page fault è possibile esaminare il bit di riferimento corrente e 2 bit in memoria per stabilire se una pagina sia stata usata entro gli ultimi 10.000-15.000 riferimenti. Se lo è stata, almeno uno di questi bit è attivo. Se non lo è stata, questi bit sono tutti inattivi. Le pagine con almeno un bit attivo si considerano appartenenti al working set. Occorre notare che questo schema non è del tutto preciso, poiché non è possibile stabilire dove si è verificato un riferimento entro un intervallo di 5000. L'incertezza si può ridurre aumentando il numero dei bit di cronologia e la frequenza dei segnali d'interruzione, per esempio, 10 bit e un'interruzione ogni 1000 riferimenti. Tuttavia, il costo per servire questi segnali d'interruzione più frequenti aumenta in modo corrispondente.

10.6.3 Frequenza dei page fault

Il modello del working set ha avuto successo, e la sua conoscenza può servire per la prepaginazione (Paragrafo 10.9.1), ma appare un modo alquanto goffo per controllare il thrashing. La strategia basata sulla frequenza dei page fault (*page fault frequency*, pff) è più diretta.

Il problema specifico è la prevenzione del thrashing. La frequenza dei page fault in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza dei page fault è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata dei page fault (Figura 10.23). Se la frequenza effettiva dei page fault per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore, si sottrae un frame a quel processo. Quindi, per prevenire il thrashing, si può misurare e controllare direttamente la frequenza dei page fault.

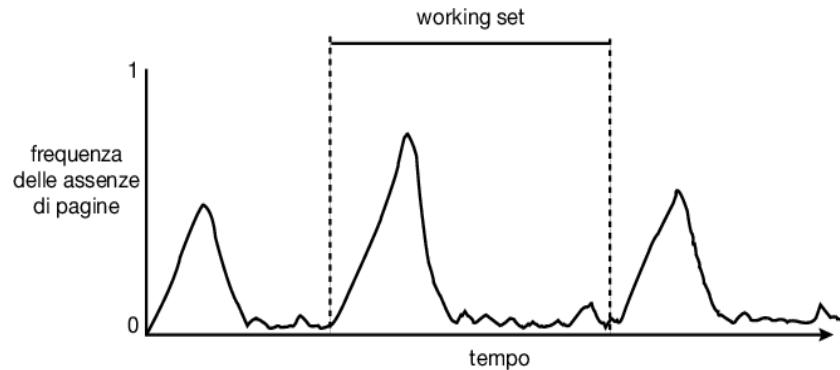


Come nel caso della strategia del working set, può essere necessaria lo swapping di un processo. Se la frequenza dei page fault aumenta e non ci sono frame disponibili, occorre selezionare un processo e spostarlo sul backing store. I frame liberati si distribuiscono ai processi con elevate frequenze di page fault.

WORKING SET E TASSI DI PAGE FAULT

Vi è una relazione diretta tra il working set di un processo e il tasso di page fault. Come mostra la Figura 10.22, il working set di un processo cambia nel corso del tempo, mentre i riferimenti ai dati e alle parti del codice passano da una località all'altra.

Assumendo memoria sufficiente a contenere il working set di un processo (ossia, a evitare il thrashing), il tasso di page fault oscillerà, in un dato periodo di tempo, tra picchi e valli. Questa tendenza generale è illustrata di seguito.



Un picco nel tasso di page fault si verifica alle richieste di paginazione relative a una nuova località. Tuttavia, una volta che il working set interessato è in memoria, il tasso di page fault precipita. Quando il processo entra in un nuovo

working set, il tasso si impenna ancora una volta verso un picco; quando il nuovo working set è caricato in memoria, il tasso crolla nuovamente. L'intervallo di tempo tra l'inizio di un picco e quello del picco successivo descrive la transizione da un working set a un altro.

10.6.4 Regole correntemente adottate

Il thrashing e l'avvicendamento dei processi hanno un pessimo impatto sulle prestazioni. La miglior regola adottata attualmente nella realizzazione dei computer è quella di includere una quantità di memoria fisica sufficiente a evitare thrashing e avvicendamento, quando possibile. Sia quando si parla di smartphone che nel caso di mainframe, fornire una quantità di memoria sufficiente a mantenere tutti gli insiemi di lavoro contemporaneamente in memoria, eccetto in condizioni estreme, offre all'utente la migliore esperienza d'uso possibile.

diviene scarsa e utilizzando solo in seguito la paginazione, quando la compressione non risolve il problema. I test prestazionali indicano che la compressione della memoria è più veloce della paginazione anche in caso d'utilizzo di ssd sui sistemi macos laptop e desktop.

Sebbene la compressione della memoria richieda l'assegnazione di frame liberi per contenere le pagine compresse, è possibile ottenere un notevole risparmio di memoria, che dipende dalla riduzione ottenuta dall'algoritmo di compressione (nell'esempio precedente, i tre frame sono stati ridotti a un terzo della loro dimensione originale). Come in ogni forma di compressione dei dati, la velocità dell'algoritmo di compressione e la quantità di riduzione che può essere raggiunta (nota come il rapporto di compressione) sono in conflitto tra loro. In generale, i rapporti di compressione più elevati (maggior riduzione delle dimensioni) possono essere ottenuti con algoritmi più lenti, più costosi dal punto di vista della computazione. La maggior parte degli algoritmi in uso oggi bilancia questi due fattori, raggiungendo rapporti di compressione relativamente alti con l'utilizzo di algoritmi veloci. Inoltre, gli algoritmi di compressione sono migliorati sfruttando più core di elaborazione ed eseguendo la compressione in parallelo. Per esempio, gli algoritmi di compressione Microsoft Xpress e Apple wkdm sono considerati veloci e riescono a comprimere le pagine al 30-50% della loro dimensione originale.

10.8 Allocazione di memoria del kernel

Quando un processo eseguito in modalità utente necessita di memoria aggiuntiva, le pagine sono allocate dalla lista dei frame disponibili mantenuta dal kernel. Per formare questa lista, si applica in genere uno degli algoritmi di sostituzione delle pagine esaminati nel Paragrafo 10.4; molto verosimilmente, la lista conterrà pagine non utilizzate sparse per tutta la memoria, come abbiamo visto in precedenza. Va inoltre ricordato che, se un processo utente richiede un solo byte di memoria, si ottiene frammentazione interna, poiché al processo viene garantito un intero frame.

Il kernel, per allocare la propria memoria, attinge spesso a una riserva di memoria libera differente dalla lista usata per soddisfare gli ordinari processi in modalità utente. Questo avviene principalmente per due motivi.

1. Il kernel richiede memoria per strutture dati dalle dimensioni variabili; alcune di loro corrispondono a meno di una pagina. Deve quindi fare un uso oculato della memoria, tentando di contenere al minimo gli sprechi dovuti alla frammentazione. Questo fattore è di particolare rilevanza, se si considera che, in molti sistemi operativi, il codice e i dati del kernel non sono paginabili.
2. Le pagine allocate ai processi in modalità utente non devono necessariamente essere contigue nella memoria fisica. Alcuni dispositivi, però, interagiscono direttamente con la memoria fisica, senza il vantaggio dell'interfaccia della memoria virtuale; di conseguenza, possono richiedere memoria che risieda in pagine fisicamente contigue.

Nei paragrafi successivi esamineremo due strategie per la gestione della memoria libera assegnata ai processi del kernel: il cosiddetto "sistema buddy" e l'allocazione a lastre.

10.8.1 Sistema buddy

Il "sistema buddy" utilizza un segmento di grandezza fissa per l'allocazione della memoria, costituito da pagine fisicamente contigue. La memoria è assegnata mediante un cosiddetto allocatore a potenze di 2, che alloca memoria in unità di dimensioni pari a potenze di 2 (4 kb, 8 kb, 16 kb, e via di seguito). Le differenti quantità richieste sono arrotondate alla successiva potenza di 2. Per esempio, una richiesta di 11 kb viene soddisfatta con un segmento di 16 kb.

Consideriamo un semplice esempio e ipotizziamo che la grandezza di un segmento di memoria sia inizialmente di 256 kb e che il kernel richieda 21 kb di memoria. In primo luogo il segmento è suddiviso in due buddy ("compagni"), che chiameremo A_S e A_D , ciascuno dei quali misura 128 kb. Uno di questi è ulteriormente dimezzato in 2 buddy da 64 kb, diciamo B_S e B_D . Poiché la minima potenza di 2 che supera 21 kb è pari a 32 kb, occorre suddividere ancora B_S , oppure B_D , in due buddy la cui dimensione è 32 kb; chiamiamoli C_S e C_D . Uno di questi è utilizzato per soddisfare la richiesta di 21 kb. Lo schema è illustrato nella Figura 10.26, dove C_S è il segmento allocato per la richiesta di 21 kb.

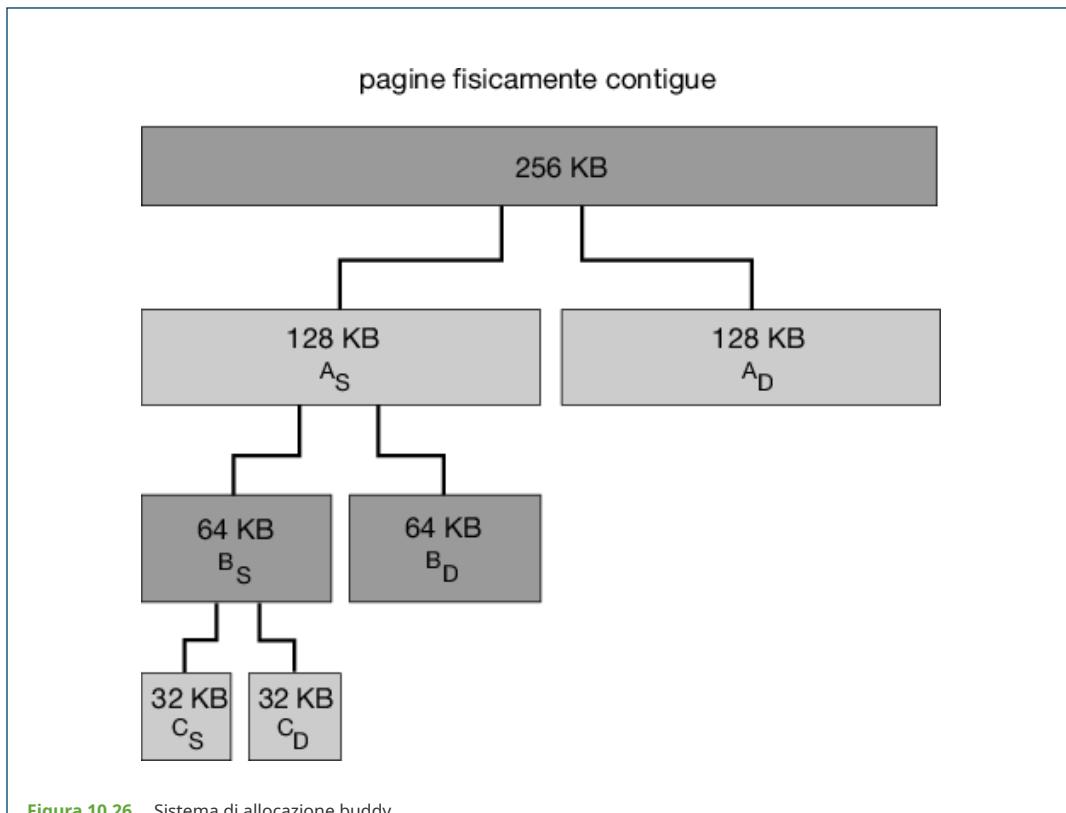


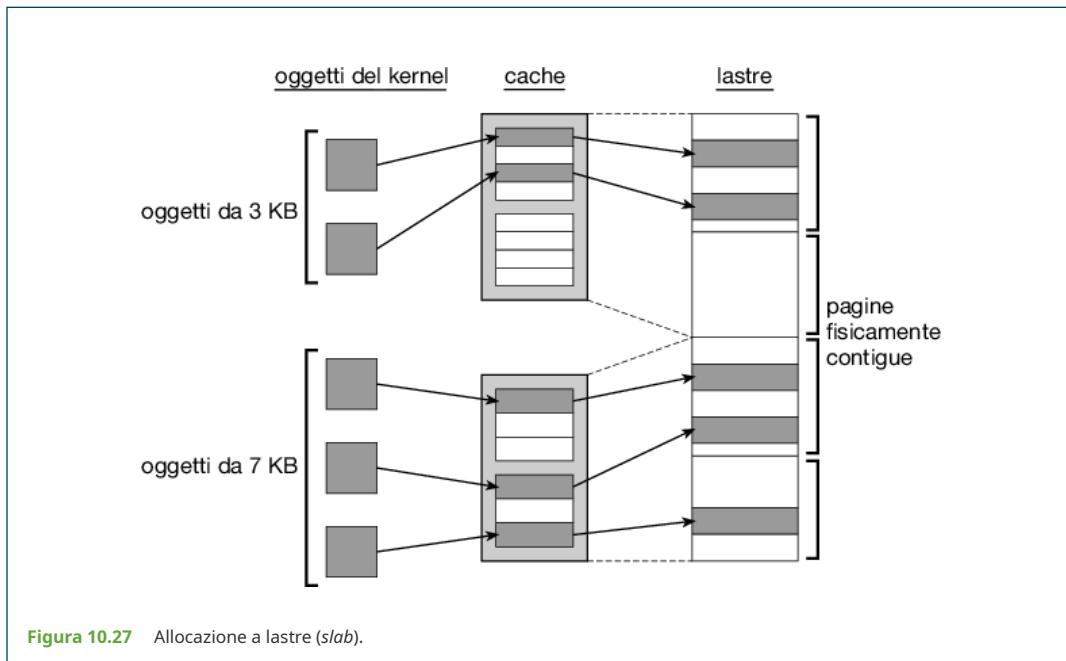
Figura 10.26 Sistema di allocazione buddy.

Questo sistema offre il vantaggio di poter congiungere rapidamente buddy adiacenti per formare segmenti più grandi tramite una tecnica nota come fusione (*coalescing*). Nella Figura 10.26, per esempio, quando il kernel rilascia l'unità C_S che gli era stata allocata, il sistema può fondere C_S e C_D in un segmento di 64 kb. Questo segmento, B_S , può a sua volta fondersi con il proprio compagno B_D , costituendo un segmento di 128 kb. Con l'ultima operazione di fusione si può ritornare al segmento originale di 256 kb.

L'ovvio inconveniente di questo sistema è che l'arrotondamento per eccesso a una potenza di 2 può facilmente generare frammentazione all'interno dei segmenti allocati. Una richiesta di 33 kb, per esempio, può essere soddisfatta solo con un segmento di 64 kb. Proprio per effetto della frammentazione interna, dunque, risulta impossibile garantire che lo spreco dell'unità allocata resterà al di sotto del 50%. Nel paragrafo successivo presentiamo una tecnica di allocazione della memoria priva dello spreco dovuto alla frammentazione.

10.8.2 Allocazione a lastre

Una seconda strategia per assegnare la memoria del kernel è detta allocazione a lastre (*slab allocation*). Una lastra è composta da una o più pagine fisicamente contigue. Una cache consiste di una o più lastre. Vi è una sola cache per ciascuna categoria di struttura dati del kernel: una cache dedicata alla struttura dati che rappresenta i descrittori dei processi, una dedicata agli oggetti che rappresentano i file, un'altra per i semafori, e così via. Ogni cache è popolata da oggetti, istanze della struttura dati del kernel rappresentata dalla cache. La cache che rappresenta i semafori, per esempio, memorizza istanze di oggetti semaforo; quella che rappresenta i descrittori dei processi memorizza istanze di descrittori dei processi, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 10.27; essa mostra due oggetti del kernel che misurano 3 kb e tre oggetti che misurano 7 kb, memorizzati nelle rispettive cache.



L'algoritmo di allocazione a lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, un certo numero di oggetti, inizialmente dichiarati liberi, viene assegnato alla cache. Questo numero dipende dalla grandezza della lastra associata alla cache. Per esempio, una lastra di 12 kb (formata da tre pagine contigue di 4 kb) potrebbe contenere sei oggetti di 2 kb ciascuno. Al principio, tutti gli oggetti nella cache sono contrassegnati come liberi. Quando una struttura dati del kernel ha bisogno di un oggetto, per soddisfare la richiesta l'allocatore può selezionare dalla cache qualunque oggetto libero; l'oggetto tratto dalla cache è quindi contrassegnato come usato.

Consideriamo una situazione in cui il kernel richieda all'allocatore delle lastre la memoria per un oggetto rappresentante un descrittore dei processi. Nei sistemi Linux, un descrittore dei processi ha tipo `struct task_struct`, che richiede circa 1,7 kb di memoria. Quando il kernel di Linux crea un nuovo task, richiede alla propria cache la memoria necessaria per l'oggetto di tipo `struct task_struct`. La cache darà corso alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e rechi il contrassegno "libero".

In Linux una lastra può essere in uno dei seguenti stati.

1. Piena. Tutti gli oggetti della lastra sono contrassegnati come usati.

2. Vuota. Tutti gli oggetti della lastra sono contrassegnati come liberi.
3. Parzialmente occupata. La lastra contiene oggetti sia usati sia liberi.

L'allocatore a lastre, per soddisfare una richiesta, tenta in primo luogo di estrarre un oggetto libero da una lastra parzialmente occupata; se non ne esistono, assegna un oggetto libero da una lastra vuota; in mancanza di lastre vuote disponibili, crea una nuova lastra da pagine fisiche contigue e la alloca a una cache; da tale lastra si attinge la memoria da allocare all'oggetto.

L'allocatore a lastre offre due vantaggi principali.

1. Annulla lo spreco di memoria derivante da frammentazione. La frammentazione non è un problema, poiché ogni struttura dati del kernel ha una cache associata; ciascuna delle cache è composta da un numero variabile di lastre, suddivise in spezzoni di grandezza pari a quella degli oggetti rappresentati. Pertanto, quando il kernel richiede memoria per un oggetto, l'allocatore a lastre restituisce la quantità esatta di memoria necessaria per rappresentare l'oggetto.
2. Le richieste di memoria possono essere soddisfatte rapidamente. La tecnica di allocazione a lastre si rivela particolarmente efficace quando, nella gestione della memoria, gli oggetti sono frequentemente allocati e deallocati, come spesso accade con le richieste del kernel. In termini di tempo, allocare e deallocare memoria può essere un processo dispendioso. Tuttavia, gli oggetti sono creati in anticipo e possono dunque essere allocati rapidamente dalla cache. Inoltre, quando il kernel rilascia un oggetto di cui non ha più bisogno, questo è dichiarato libero e restituito alla propria cache, rendendolo così immediatamente disponibile ad altre richieste del kernel.

L'allocatore a lastre ha fatto la sua prima apparizione nel kernel di Solaris 2.4. Per la sua natura generale è ora applicato da Solaris anche ad alcune richieste di memoria in modalità utente. Linux adottava, originariamente, il sistema buddy; tuttavia, a partire dalla versione 2.2, il kernel di Linux include l'allocazione a lastre.

Le distribuzioni recenti di Linux includono altri due allocator di memoria del kernel, slob e slub. (L'implementazione Linux dell'allocazione a lastre viene chiamata slab).

L'allocatore slob è progettato per sistemi con una quantità limitata di memoria, come per esempio i sistemi embedded. slob (acronimo di simple list of blocks, "semplice lista di blocchi") funziona mantenendo tre liste di oggetti: piccoli (per gli oggetti più piccoli di 256 byte), medi (per gli oggetti più piccoli di 1.024 byte) e grandi (per gli oggetti oltre i 1.024 byte). Le richieste di memoria sono soddisfatte utilizzando una politica first-fit.

A partire dalla versione 2.6.24, l'allocatore slub ha sostituito slab come allocatore di default per il kernel Linux. slub risolve i problemi di prestazioni dell'allocazione a lastre riducendo gran parte dell'overhead richiesto da slab. Uno dei cambiamenti costituisce nello spostamento dei metadati archiviati con ogni lastra nell'allocazione slab alla struttura `page` che il kernel Linux utilizza per ogni pagina. Inoltre, in slub sono state eliminate le code per singola cpu, mantenute da slab per gli oggetti in ogni cache. Per sistemi con un gran numero di processori, la quantità di memoria allocata a queste code non era del tutto insignificante. slub fornisce quindi prestazioni migliori al crescere del numero di processori.

10.9 Altre considerazioni

Le due scelte fondamentali nella progettazione dei sistemi di paginazione sono la definizione dell'algoritmo di sostituzione e della politica di allocazione, già analizzate in questo capitolo. Si devono però fare anche molte altre considerazioni, che riportiamo nel seguito.

10.9.1 Prepaginazione

Una caratteristica ovvia per un sistema di paginazione su richiesta puro, consiste nell'alto numero di page fault che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La prepaginazione rappresenta un tentativo di prevenire questo alto livello di paginazione iniziale.

In un sistema che usa il modello del working set, per esempio, a ogni processo si può associare una lista delle pagine contenute nel suo working set. Se occorre sospendere un processo a causa di un'attesa di i/o oppure dell'assenza di frame liberi, si memorizza il suo working set. Al momento di riprendere l'esecuzione del processo (perché l'i/o è terminato o un numero sufficiente di frame liberi è diventato disponibile), prima di riavviare il processo, si riporta in memoria il suo intero working set.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per servire i corrispondenti page faults. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepaginate s pagine e sia effettivamente usata una frazione a di queste s pagine ($0 \leq a \leq 1$). Occorre sapere se il costo delle as eccezioni di page fault risparmiate sia maggiore o minore del costo di prepaginazione di $(1 - a)s$ pagine non necessarie. Se il parametro a è prossimo allo 0, la prepaginazione non è conveniente; se è prossimo a 1, la prepaginazione certamente lo è.

Si noti inoltre che la prepaginazione di un programma eseguibile può essere difficile, perché non sempre è chiaro quali pagine debbano essere portate in memoria. La prepaginazione di un file è spesso maggiormente prevedibile, poiché l'accesso ai file è solitamente sequenziale. La chiamata di sistema `readahead()` di Linux precarica il contenuto di un file in memoria in modo che gli accessi successivi al file si verifichino nella memoria principale.

10.9.2 Dimensione delle pagine

È raro che chi progetta un sistema operativo per un calcolatore esistente possa scegliere le dimensioni delle pagine. Tuttavia, se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine. Come s'intuisce non esiste un'unica dimensione migliore, ma più fattori sono a sostegno delle diverse dimensioni. Le dimensioni delle pagine sono invariabilmente potenze di 2, in genere comprese tra 4096 (2^{12}) e 4.194.304 (2^{22}) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la dimensione della tabella delle pagine. Per un dato spazio di memoria virtuale, diminuendo la dimensione delle pagine aumenta il numero delle stesse e quindi la dimensione della tabella delle pagine. Per una memoria virtuale di 4 mb (2^{22}), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano grandi.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione 00000 e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata (le pagine sono unità di allocazione) una parte della pagina finale (frammentazione interna). Supponendo che le dimensioni del processo e delle pagine siano indipendenti è probabile che, in media, metà dell'ultima pagina di ogni processo sia sprecata. Questa perdita è di soli 256 byte per una pagina di 512 byte, ma di 4096 byte per una pagina di 8192 byte. Quindi, per ridurre la frammentazione interna occorrono pagine di piccole dimensioni.

Un altro problema è il tempo richiesto per leggere o scrivere una pagina. Come si vedrà nel Paragrafo 11.1, quando il dispositivo di archiviazione è un hdd, il tempo di i/o è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che siano preferibili pagine piccole. Tuttavia, il tempo di trasferimento è normalmente molto piccolo se confrontato con il tempo di latenza e il tempo di posizionamento. A una velocità di trasferimento di 50 mb al secondo, per trasferire 512 byte s'impiegano 0,01 millisecondi. D'altra parte, il tempo di latenza è di circa 3 millisecondi e quello di posizionamento 5 millisecondi. Perciò, del tempo totale di i/o (8,01 millisecondi), solo lo 0,1 per cento è attribuibile al trasferimento effettivo. Raddoppiando le dimensioni delle pagine, il tempo di i/o aumenta solo fino a 8,02 millisecondi. S'impiegano 8,02 millisecondi per leggere una sola pagina di 1024 byte, ma 16,02 millisecondi per leggere la stessa quantità di byte come due pagine di 512 byte l'una. Quindi, per ridurre il tempo di i/o occorre avere pagine di dimensioni maggiori.

Tuttavia, con pagine di piccole dimensioni si dovrebbe ridurre l'i/o totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di adattarsi con maggior precisione alla località del programma. Si consideri, per esempio, un processo di 200 kb, dei quali solo la metà (100 kb) sono effettivamente usati durante l'esecuzione. Se si dispone di una sola ampia pagina, occorre inserirla tutta, sicché vengono trasferiti e assegnati 200 kb. Disponendo di pagine di 1 byte, si potrebbero invece portare in memoria i soli 100 kb effettivamente usati, con trasferimento e allocazione di quei soli 100 kb. Con pagine di piccole dimensioni è possibile avere una migliore risoluzione, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre allocare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole danno come risultato una minore attività di i/o e una minore memoria totale allocata.

D'altra parte occorre notare che con pagine di 1 byte si verifica un page fault per *ciascun* byte. Un processo di 200 kb che usasse solo metà di tale memoria genererebbe un solo page fault con una pagina di 200 kb, ma 102.400 page fault con le pagine di 1 byte. Ciascun

page fault causa un rilevante sovraccarico necessario a elaborare l'eccezione, salvare i registri, sostituire una pagina, attendere nella coda del dispositivo di paginazione e aggiornare le tabelle. Per ridurre il numero di page fault al minimo sono necessarie pagine di grandi dimensioni.

Occorre considerare altri fattori, come la relazione tra la dimensione delle pagine e quella dei settori del mezzo di paginazione. Non esiste una risposta ottimale al problema considerato. Alcuni fattori (frammentazione interna, località) sono a favore delle piccole dimensioni, mentre altri (dimensione delle tabelle, tempo di i/o) sono a favore delle grandi dimensioni. Tuttavia la tendenza è storicamente verso l'incremento delle dimensioni delle pagine e questo vale anche per i sistemi mobili. Nella prima edizione di questo testo (1983) si considerava un valore di 4096 byte come limite superiore alla dimensione delle pagine. Nel 1990 tale dimensione delle pagine era la più comune. I sistemi moderni possono impiegare pagine di dimensioni assai maggiori, come vedremo nel paragrafo successivo.

10.9.3 Portata del tlb

Il tasso di successi (*hit ratio*) di un tlb – si veda in proposito il Capitolo 9 – si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dal tlb anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi del tlb, e un modo per migliorarlo è aumentare il numero di voci nel tlb. Tuttavia, la memoria associativa che si usa per costruire il tlb è costosa e consuma molta energia.

Una metrica collegata al tasso di successi, detto portata del tlb (*tlb reach*), esprime la quantità di memoria accessibile dal tlb, ed è dato semplicemente dal numero di elementi moltiplicato per la dimensione delle pagine. Idealmente, il tlb dovrebbe contenere il working set di un processo; altrimenti, il processo passerà molto tempo traducendo riferimenti alla memoria nella page table invece che nel tlb. Se si raddoppia il numero di elementi del tlb, se ne raddoppia la portata; per alcune applicazioni che comportano un uso intensivo della memoria ciò potrebbe rivelarsi ancora insufficiente per la memorizzazione del working set.

Un altro metodo per aumentare la portata del tlb consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare pagine di diverse dimensioni. Se si aumenta la dimensione delle pagine, per esempio da 4 kb a 16 kb, la portata del tlb si quadruplica. Quest'aumento potrebbe però condurre a una maggiore frammentazione della memoria relativamente alle applicazioni che non richiedono pagine così grandi. Come alternativa, molte architetture possono utilizzare diverse dimensioni di pagina e il sistema operativo può essere configurato per trarne vantaggio. Per esempio, la dimensione di pagina predefinita sui sistemi Linux è 4 kb; tuttavia, Linux fornisce anche le *huge page* ("pagine enormi"), designando una regione di memoria fisica in cui è possibile utilizzare pagine più grandi (per esempio di 2 mb).

Ricordiamo dal Paragrafo 9.7 che l'architettura armv8 fornisce supporto per pagine e regioni di diverse dimensioni. Inoltre, ciascun elemento del tlb in armv8 contiene un bit di contiguità. Se, per un determinato elemento del tlb, il bit di contiguità è impostato a 1, l'elemento mappa blocchi di memoria contigui (adiacenti). La portata del tlb risulta così aumentata, poiché tre possibili disposizioni di blocchi contigui possono essere mappate in un unico elemento del tlb:

1. elemento del tlb da 64 kb comprendente 16 blocchi adiacenti da 4 kb;
2. elemento del tlb da 1 gb comprendente 32 blocchi adiacenti da 32 mb;
3. elemento del tlb da 2 mb comprendente di 32 blocchi adiacenti da 64 kb o 128 blocchi adiacenti da 16 kb.

L'uso di diverse dimensioni delle pagine richiede però che la gestione del tlb sia svolta dal sistema operativo e non direttamente dall'hardware. Per esempio, uno dei campi degli elementi del tlb deve indicare la dimensione della pagina fisica cui il contenuto di ciascun elemento fa riferimento, oppure, nel caso di architetture arm, deve indicare che l'elemento fa riferimento a un blocco di memoria contiguo. La gestione del tlb svolta dal sistema operativo e non esclusivamente dall'architettura comporta una penalizzazione delle prestazioni. Tuttavia, i vantaggi dovuti all'aumento del tasso di successi e della portata del tlb compensano i costi prestazionali.

10.9.4 Tabella delle pagine invertita

Nel Paragrafo 9.4.3 si è introdotto il concetto di tabella delle pagine invertita come sistema di gestione delle pagine che consente di ridurre la quantità di memoria fisica necessaria per tener traccia della corrispondenza tra gli indirizzi virtuali e gli indirizzi fisici. Tale riduzione si ottiene tramite una tabella con un elemento per pagina fisica, indicizzato dalla coppia *<id-processo, numero di pagina>*.

Poiché contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, che sono necessarie se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di page fault. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

L'uso delle tabelle esterne delle pagine non pregiudica l'utilità della tabella delle pagine invertita; infatti si fa riferimento alle tabelle esterne solo nel caso di un page fault; quindi non è necessario che siano immediatamente disponibili ed esse stesse sono paginate dentro e fuori dalla memoria quando è necessario. Sfortunatamente, in questo modo un primo page fault può far sì che il gestore della memoria virtuale generi un altro page fault quando carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale nel backing store. Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e causa un ritardo nell'elaborazione della ricerca della pagina.

10.9.5 Struttura dei programmi

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l'utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono migliorare se l'utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di 128×128 elementi. Il tipico codice è il seguente:

```
int i, j;

int[128][128] data;

for (j = 0; j < 128; j++)

    for (i = 0; i < 128; i++)

        data[i][j] = 0;
```

Occorre notare che l'array è memorizzato per riga, vale a dire che è disposto in memoria secondo l'ordine `data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]`. In pagine di 128 parole, ogni riga occupa una pagina, quindi il frammento di codice precedente azzerà una parola per pagina, poi un'altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa $128 \times 128 = 16.384$ page fault. D'altra parte, cambiando il codice in

```
int i, j;

int[128][128] data;

for (i = 0; i < 128; i++)

    for (j = 0; j < 128; j++)

        data[i][j] = 0;
```

si azzerano tutte le parole di una pagina prima che si inizi la pagina successiva, riducendo a 128 il numero di page fault.

Un'attenta scelta delle strutture dati e delle strutture di programmazione può aumentare la località e quindi ridurre il tasso di page fault e il numero di pagine del working set. Una buona località è quella di uno stack, poiché l'accesso avviene sempre alla sua parte superiore. Una tabella hash, invece, è progettata proprio per distribuire i riferimenti, causando una località non buona. Naturalmente, la località dei riferimenti rappresenta soltanto una misura dell'efficienza d'uso di una struttura dati. Altri fattori rilevanti sono rapidità di ricerca, numero totale dei riferimenti alla memoria e numero totale delle pagine coinvolte.

In uno studio successivo, anche il compilatore e il loader possono avere un effetto notevole sulla paginazione. La separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate. Nel sostituire le pagine non modificate, non occorre scriverle in memoria ausiliaria. Il loader può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono "impaccare" nella stessa pagina. Questa forma di impaccamento è una variante del problema del *bin-packing* della ricerca operativa: cercare di impaccare i segmenti di dimensione variabile in pagine di dimensione fissa, in modo da ridurre al minimo i riferimenti tra pagine diverse. Un metodo di questo tipo è utile soprattutto per pagine di grandi dimensioni.

10.9.6 Vincolo di i/o e vincolo delle pagine

Quando si usa la paginazione su richiesta, talvolta occorre permettere che alcune pagine si possano vincolare in memoria (*locked in memory*). Una situazione di questo tipo si presenta quando l'i/o si esegue verso o dalla memoria (virtuale) utente. Spesso il sistema di i/o comprende un processore dedicato; al controllore di un dispositivo di memorizzazione usb, per esempio, generalmente si indica il numero di byte da trasferire e un indirizzo di memoria per il buffer (Figura 10.28). Completato il trasferimento, la cpu riceve un segnale d'interruzione.

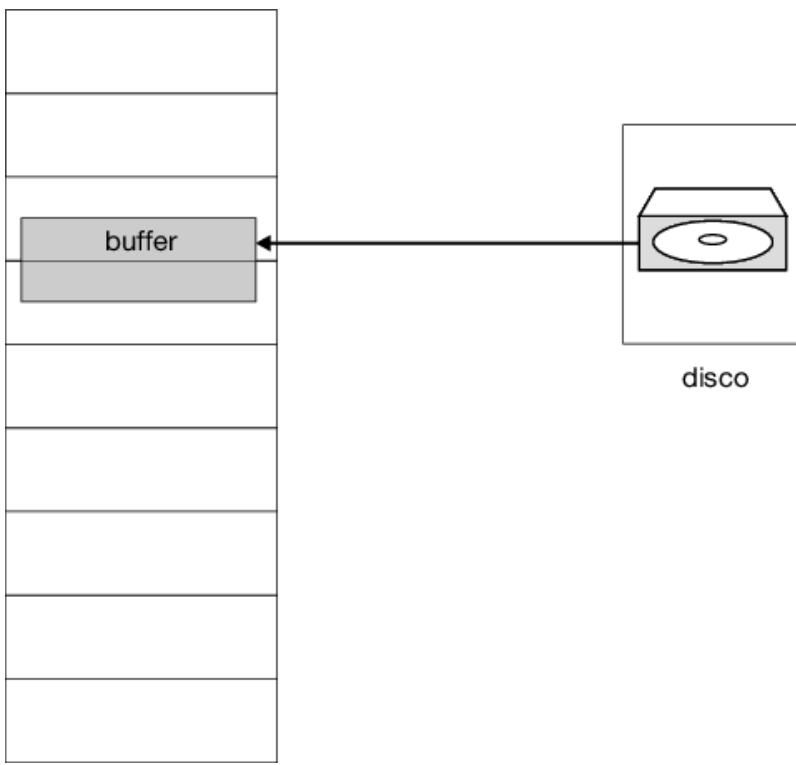


Figura 10.28 Ragione per i cui i frame usati per l'i/o devono essere presenti in memoria.

Occorre essere certi che non si verifichi la seguente successione di eventi: un processo effettua una richiesta di i/o ed è messo in coda per il relativo dispositivo. Nel frattempo si assegna la cpu ad altri processi che accusano page fault e, usando un algoritmo di sostituzione globale, uno di questi sostituisce la pagina contenente il buffer di i/o del primo processo, pagina che viene scaricata dalla memoria. Qualche tempo dopo, quando la richiesta di i/o raggiunge la prima posizione della coda d'attesa per il dispositivo, l'operazione di i/o avviene all'indirizzo specificato, ma questo frame è ora impiegato per una pagina appartenente a un altro processo.

Questo problema si può risolvere in due modi. Una soluzione prevede di non eseguire operazioni di i/o in memoria utente, ma di copiare i dati sempre tra la memoria di sistema e la memoria utente. In questo modo l'i/o avviene solo tra la memoria di sistema e il dispositivo di i/o. Per scrivere dati in un nastro, occorre prima copiarli in memoria di sistema, quindi trasferirli all'unità a nastro. Tale copia supplementare può causare un sovraccarico inaccettabile.

Un'altra soluzione consiste nel permettere che le pagine siano vincolate in memoria. A ogni frame si associa un bit di vincolo (*lock bit*); se tale bit è attivato, la pagina contenuta in tale frame non può essere selezionata per la sostituzione. Con questo metodo, per scrivere dati in un nastro occorre vincolare alla memoria le pagine contenenti tali dati, quindi il sistema può continuare come di consueto. Le pagine vincolate non si possono sostituire. Completato l'i/o, si rimuove il vincolo.

I bit di vincolo sono usati in varie situazioni. Spesso il kernel del sistema operativo, o una sua parte, è vincolato alla memoria. La maggior parte dei sistemi non può tollerare un page fault generato dal kernel o da un suo modulo, incluso il modulo incaricato della gestione della memoria. Anche i processi utente possono aver bisogno di vincolare pagine alla memoria. Per esempio, a un processo database potrebbe tornare utile la possibilità di gestire una porzione di memoria, spostando autonomamente blocchi tra il disco e la memoria, perché ha una migliore conoscenza di come vorrà utilizzare i suoi dati. Una simile gestione delle pagine in memoria (*pinning*) è abbastanza comune e la maggior parte dei sistemi operativi dispone di una chiamata di sistema che consente a una applicazione di richiedere che una regione del suo spazio di indirizzamento logico sia vincolata. Si noti che questa caratteristica potrebbe essere usata eccessivamente creando problemi agli algoritmi di gestione della memoria. Per questa ragione un'applicazione ha spesso bisogno di privilegi speciali per poter effettuare una richiesta di questo tipo.

Un altro uso del bit di vincolo riguarda la normale sostituzione di pagine. Si consideri la seguente successione d'eventi: un processo a bassa priorità subisce un page fault. Selezionando un frame per la sostituzione, il sistema di paginazione carica in memoria la pagina necessaria. Pronto per continuare, il processo con priorità bassa entra nella coda dei processi pronti per l'esecuzione e attende l'allocazione della cpu. Giacché si tratta di un processo con bassa priorità, può non essere selezionato dallo scheduler della cpu per un certo tempo. Mentre il processo con priorità bassa attende, un processo ad alta priorità ha un page fault. Durante la ricerca per la sostituzione, il sistema di paginazione individua una pagina in memoria alla quale non sono stati fatti riferimenti o modifiche; si tratta

della pagina che il processo con bassa priorità ha appena caricato. Questa pagina sembra una sostituzione perfetta: non è stata modificata, non è necessario scriverla in memoria secondaria e, apparentemente, non è stata usata da molto tempo.

Stabilire se la pagina del processo con bassa priorità si debba sostituire a vantaggio del processo con alta priorità è un problema di politica di gestione. Dopo tutto, si ritarda semplicemente un processo con bassa priorità a vantaggio di quello con priorità alta. D'altra parte, però, si spreca il lavoro fatto per trasferire in memoria la pagina del processo con bassa priorità. Se si vuole evitare che una pagina appena caricata sia sostituita prima che sia usata almeno una volta si può usare il bit di vincolo. Se una pagina viene portata in memoria, si attiva il suo bit di vincolo: tale bit rimane attivato finché si esegue nuovamente il processo che ha avuto il page fault.

Tuttavia, l'uso dei bit di vincolo può essere pericoloso: se un bit non viene mai disattivato, per esempio a causa di un baco del sistema operativo, il frame relativo alla pagina vincolata diventa inutilizzabile. Su un sistema a singolo utente, l'abuso di tale meccanismo può causare danni soltanto allo stesso utente. Ciò non si può consentire nei sistemi multiutente. Il sistema operativo Solaris, per esempio, consente l'impiego di "suggerimenti" (*hint*) di vincolo delle pagine, che si possono però trascurare se l'insieme delle pagine libere diviene troppo piccolo o se un singolo processo richiede che troppe pagine siano vincolate in memoria.

10.10 Esempi di sistemi operativi

In questo paragrafo si descrive la realizzazione della memoria virtuale nei sistemi operativi Linux, Windows e Solaris.

10.10.1 Linux

Nel Paragrafo 10.8.2 abbiamo discusso di come Linux gestisce la memoria del kernel usando l'allocazione a lastre. Vediamo ora come Linux gestisca la memoria virtuale. Linux usa la paginazione a richiesta, allocando pagine da una lista dei frame liberi. Inoltre, utilizza una politica di sostituzione globale delle pagine simile all'algoritmo a orologio descritto nel Paragrafo 10.4.5.2. Per gestire la memoria Linux mantiene due tipi di liste di pagine: `active_list`, che contiene le pagine considerate in uso, e `inactive_list`, che contiene pagine a cui non è stato fatto riferimento di recente e la cui memoria può essere recuperata.

Ogni pagina ha un bit `accessed` che viene settato a ogni riferimento alla pagina (i bit effettivamente usati per contrassegnare l'accesso alla pagina variano a seconda dell'architettura). Quando una pagina viene allocata per la prima volta, il suo bit `accessed` viene impostato e la pagina viene aggiunta in fondo alla `active_list`. Allo stesso modo, ogni volta che si fa riferimento a una pagina della `active_list`, il bit `accessed` viene impostato e la pagina viene spostata in fondo alla lista stessa. Periodicamente, i bit `accessed` delle pagine nella `active_list` vengono resettati. La pagina utilizzata meno di recente si sposterà nel tempo all'inizio della `active_list` e da lì potrà essere spostata in fondo alla `inactive_list`. Quando si fa riferimento a una pagina nella `inactive_list`, questa viene inserita in coda alla `active_list`. Questo modello è illustrato nella Figura 10.29.

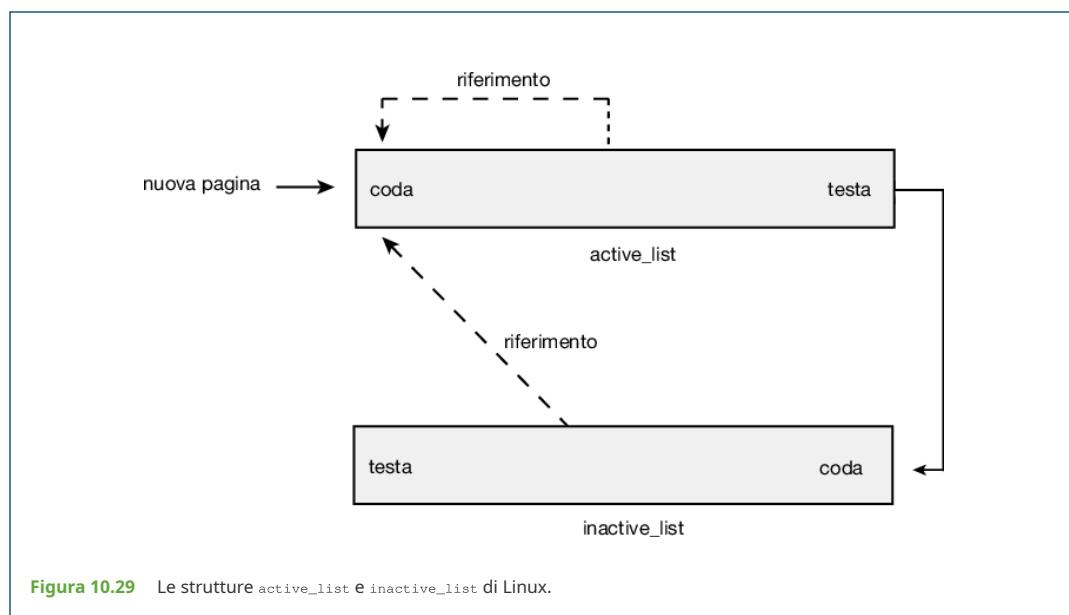


Figura 10.29 Le strutture `active_list` e `inactive_list` di Linux.

Le due liste sono mantenute bilanciate tra loro e quando la `active_list` diventa molto più grande della `inactive_list`, le pagine in testa alla `active_list` passano alla `inactive_list`, dove diventano idonee per il recupero di spazio di memoria. Il demone `kswapd` del kernel Linux si risveglia periodicamente e controlla la quantità di memoria disponibile nel sistema: se la memoria libera scende al di sotto di una certa soglia, `kswapd` inizia la scansione delle pagine nella `inactive_list` e le recupera per la lista dei frame liberi. La gestione della memoria virtuale di Linux è discussa in maggior dettaglio nel Capitolo 20 (reperibile sulla piattaforma MyLab).

10.10.2 Windows

Windows 10 supporta sistemi a 32 e 64 bit in esecuzione su architetture Intel (ia-32 e x86-64) e arm. Nei sistemi a 32 bit, lo spazio di indirizzi virtuali predefinito per un processo è 2 gb, sebbene possa essere esteso a 3 gb. I sistemi a 32 bit supportano 4 gb di memoria fisica. Sui sistemi a 64 bit, Windows 10 ha uno spazio di indirizzi virtuali da 128 tb e supporta fino a 24 tb di memoria fisica (le versioni di Windows Server supportano fino a 128 tb di memoria fisica). Windows 10 implementa la maggior parte delle funzioni di gestione della memoria descritte finora, incluse le librerie condivise, la paginazione su richiesta, la copiatura su scrittura, la paginazione e la compressione della memoria.

Windows 10 realizza la memoria virtuale impiegando la paginazione su richiesta per gruppi, detti cluster di pagine (*demand paging with clustering*), una tecnica che riconosce la località dei riferimenti alla memoria e gestisce i page fault caricando in memoria non solo la pagina richiesta, ma più pagine a essa immediatamente precedenti e successive. La dimensione di un cluster varia in base al

tipo di pagina: per una pagina di dati un cluster contiene tre pagine (la pagina stessa, la precedente e la successiva), mentre in tutti gli altri casi la dimensione di un cluster è 7.

Un componente chiave del gestore di memoria virtuale in Windows 10 è la gestione del working set. Quando viene creato un processo, gli viene assegnato un working set minimo di 50 pagine e un working set massimo di 345 pagine. Il working set minimo è il numero minimo di pagine in memoria garantite a un processo; se è disponibile memoria sufficiente, a un processo può essere assegnato un numero di pagine fino al working set massimo. A meno che un processo non sia configurato con limiti stretti (*hard*), questi valori possono essere ignorati e un processo può crescere oltre il limite massimo impostato nel caso in cui sia disponibile memoria sufficiente. Analogamente, la quantità di memoria allocata a un processo può ridursi al di sotto del minimo nei periodi di elevata richiesta di memoria.

Windows utilizza l'algoritmo a orologio (che approssima lru) descritto nel Paragrafo 10.4.5.2, con una combinazione di politiche globali e locali di sostituzione delle pagine. Il gestore della memoria virtuale mantiene una lista dei frame liberi a cui è associato un valore di soglia che indica se è disponibile sufficiente memoria libera. Se si verifica un page fault in un processo che si trova al di sotto del limite massimo impostato, il gestore della memoria virtuale assegna al processo una pagina dalla lista delle pagine libere. Se un processo che ha raggiunto il suo working set massimo incorre in un page fault ed è disponibile sufficiente memoria, al processo viene assegnata una pagina libera, che gli consente di crescere oltre il suo limite massimo di working set. Se invece la quantità di memoria disponibile è insufficiente, il kernel deve selezionare una pagina dal working set del processo ed effettuare una sostituzione usando una politica lru di sostituzione locale delle pagine.

Quando la quantità di memoria disponibile scende al di sotto della soglia, il gestore della memoria virtuale utilizza una tattica di sostituzione globale nota come trimming di working set automatico per ripristinare il valore a un livello superiore alla soglia. Il taglio automatico del set di lavoro funziona valutando il numero di pagine assegnate ai processi. Se a un processo sono state assegnate più pagine rispetto al minimo impostato, il gestore della memoria virtuale rimuove le pagine dal working set finché non è disponibile memoria sufficiente o se il processo ha raggiunto il minimo impostato. Processi più grandi e inattivi vengono considerati prima di processi più piccoli e attivi. La procedura di trimming continua finché non vi è sufficiente memoria libera, anche nel caso in cui sia necessario rimuovere le pagine da un processo già al di sotto del working set minimo. Windows realizza la regolazione automatica del working set sia per processi utente che nel caso dei processi di sistema.

10.10.3 Solaris

Il sistema operativo Solaris assegna una pagina a un thread ogni volta che si verifica un page fault, prendendola dalla lista delle pagine libere mantenuta dal kernel. È quindi essenziale che il kernel riesca a mantenere una quantità sufficiente di memoria libera. Un parametro, `lotsfree`, associato alla lista delle pagine libere, rappresenta una soglia per l'inizio del processo di paginazione. `lotsfree` è di solito fissato a 1/64 della dimensione della memoria fisica. Il kernel verifica, quattro volte al secondo, se la quantità di memoria libera è inferiore a `lotsfree`. Se il numero di pagine libere scende sotto `lotsfree`, si avvia un processo noto come pageout. Questo processo è simile all'algoritmo con seconda chance descritto nel Paragrafo 10.4.5.2, tranne per il fatto che non usa una ma due lancette per scorrere le pagine.

Il suo funzionamento prevede che la prima lancetta scorra lungo tutte le pagine della memoria, azzerandone il bit di riferimento; più tardi, la seconda lancetta esamina il bit di riferimento delle pagine in memoria, ponendo le pagine in cui il bit di riferimento è ancora nullo in coda alla lista delle pagine libere, e scrivendone i contenuti sulla memoria secondaria in caso di modifica. Solaris gestisce anche page fault secondari (*minor page fault*), consentendo a un processo di richiamare una pagina dalla lista delle pagine libere quando si accede alla pagina prima che sia riassegnata a un altro processo.

Per controllare la frequenza di scansione delle pagine (chiamata anche `scanrate`) l'algoritmo pageout si serve di diversi parametri. Questa frequenza è espressa in pagine al secondo ed è compresa tra i valori `slowscan` e `fastscan`. Quando la memoria libera scende sotto `lotsfree`, la scansione delle pagine avviene alla frequenza `slowscan`, e sale fino a `fastscan` a secondo della quantità di memoria libera disponibile. Il valore predefinito di `slowscan` è 100, mentre `fastscan` è di solito fissato a $(\text{numero totale delle pagine fisiche})/2$ con un massimo di 8192 pagine al secondo. Questa variazione di frequenza è illustrata nella Figura 10.30 (con `fastscan` fissato al massimo).



La distanza (in pagine) tra le lancette dell'orologio è determinata dal parametro di sistema, `handspread`. L'intervallo tra l'azzeramento di un bit da parte della lancetta anteriore e l'esame del suo valore da parte della lancetta posteriore dipende sia da `scanrate` sia da `handspread`. Se il valore di `scanrate` è pari a 100 pagine al secondo e quello di `handspread` è pari a 1024 pagine, possono passare 10 secondi tra la scrittura di un bit da parte della lancetta anteriore e la sua verifica da parte di quella posteriore. Tuttavia, visti i requisiti imposti a un sistema di memoria, non sono rari valori di `scanrate` di diverse migliaia di pagine al secondo. Ciò significa che l'intervallo tra l'azzeramento e il controllo di un bit è spesso di pochi secondi.

Come si è descritto sopra, il processo `pageout` controlla la memoria quattro volte al secondo. Tuttavia, se la memoria libera scende sotto `desfree` (la quantità desiderata di memoria libera nel sistema) `pageout` sarà eseguito un centinaio di volte al secondo con lo scopo di tenere una quantità di memoria libera almeno pari a `desfree` (Figura 10.30). Se il processo `pageout` non riesce a mantenere al valore `desfree` la quantità media di memoria libera calcolata in un intervallo di 30 secondi, il kernel incomincia a effettuare lo swap di processi, liberando, in questo caso, tutte le pagine assegnate a un processo spostato dalla memoria. In generale, il kernel cerca i processi che sono rimasti inattivi per lunghi periodi. Infine, se il sistema non riesce a mantenere la quantità di memoria libera a `minfree`, invoca il processo `pageout` a ogni richiesta di una nuova pagina.

L'algoritmo di scansione delle pagine salta le pagine appartenenti alle librerie che sono condivise da diversi processi, anche se queste sono potenzialmente selezionabili. L'algoritmo distingue inoltre tra pagine allocate a processi e pagine allocate a ordinari file di dati. Questa tecnica è nota come paginazione con priorità ed è trattata nel Paragrafo 14.6.2.

10.11 Sommario

- La memoria virtuale astrae la memoria fisica in un enorme e uniforme array di memorizzazione.
- I vantaggi della memoria virtuale includono: (1) un programma può essere più grande della memoria fisica, (2) un programma non ha bisogno di risiedere interamente in memoria, (3) i processi possono condividere la memoria e (4) i processi possono essere creati in modo più efficiente.
- La paginazione su richiesta è una tecnica che consiste nel caricare le pagine solo quando vengono richieste durante l'esecuzione del programma. Le pagine che non vengono mai richieste non vengono quindi mai caricate in memoria.
- Si verifica un page fault quando si accede a una pagina che non è al momento in memoria. La pagina deve essere trasferita dalla memoria ausiliaria in una pagina fisica disponibile nella memoria principale.
- La copiatura su scrittura (copy-on-write) consente a un processo figlio di condividere lo spazio d'indirizzi del genitore. Se uno dei due processi, il figlio o il padre, scrive una pagina, modificandola, viene eseguita una copia della pagina.
- Quando la memoria disponibile si esaurisce, un algoritmo di sostituzione delle pagine seleziona una pagina presente in memoria per sostituirla con una nuova pagina. Gli algoritmi di sostituzione delle pagine includono fifo, ottimale e lru. Gli algoritmi lru puri non sono pratici da implementare e la maggior parte dei sistemi utilizza quindi algoritmi che approssimano lru.
- Gli algoritmi di sostituzione globale delle pagine selezionano una pagina per la sostituzione da qualsiasi processo nel sistema, mentre gli algoritmi di sostituzione locale selezionano una pagina dal processo che ha generato il fault.
- Il thrashing si verifica quando un sistema spende più tempo per la paginazione rispetto al tempo destinato all'esecuzione.
- Una località è un insieme di pagine che vengono utilizzate attivamente insieme. Durante l'esecuzione, un processo si sposta di località in località. Il concetto di working set si basa sulla località ed è definito come l'insieme di pagine utilizzate da un processo in un dato istante.
- La compressione della memoria è una tecnica di gestione della memoria che consiste nel comprimere un certo numero di pagine in una singola pagina. La memoria compressa è un'alternativa alla paginazione e viene utilizzata su sistemi mobili che non supportano la paginazione.
- La memoria del kernel è allocata in modo differente rispetto a quanto avviene per i processi in modalità utente, utilizzando blocchi contigui di dimensioni variabili. Due tecniche comuni per l'allocazione della memoria del kernel sono (1) il sistema buddy e (2) l'allocazione a lastre.
- La portata del tlb esprime la quantità di memoria accessibile dal tlb ed è pari al numero di elementi nel tlb moltiplicato per la dimensione della pagina. Una tecnica per aumentare la portata del tlb è aumentare la dimensione delle pagine.
- Linux, Windows e Solaris gestiscono la memoria virtuale in modo simile, utilizzando, tra l'altro, la paginazione su richiesta e la copia su scrittura. Ogni sistema utilizza anche una variante per approssimazione di lru nota come algoritmo a orologio.

Esercizi di ripasso

10.1 In quali circostanze si verifica un page fault? Descrivete le azioni che vengono intraprese dal sistema operativo in questo caso.

10.2 Considerate una successione di riferimenti alle pagine di memoria per un processo con m frame (inizialmente tutti vuoti). La successione ha lunghezza p ; in essa vi sono n distinti numeri di pagina. Rispondete alle seguenti domande relative agli algoritmi di sostituzione delle pagine in generale:

- Qual è un limite inferiore del numero di page fault?
- Qual è un limite superiore del numero di page fault?

10.3 Considerate i seguenti algoritmi di sostituzione delle pagine e valutateli basandovi su di una scala a cinque valori da “pessimo” a “ottimo” a seconda del tasso di page fault. Separate gli algoritmi che soffrono dell’anomalia di Belady da quelli che non ne sono affetti:

- sostituzione delle pagine usate meno recentemente (lru),
- sostituzione delle pagine secondo l’ordine d’arrivo (fifo);
- sostituzione ottimale;
- sostituzione alla seconda chance.

10.4 Un sistema operativo supporta la memoria virtuale paginata, utilizzando un processore centrale con una durata di ciclo di 1 microsecondo. L’accesso a una pagina diversa da quella corrente richiede 1 ulteriore microsecondo. Le pagine hanno 1.000 parole e lo strumento di paginazione è un tamburo che ruota a 3.000 giri al minuto e trasferisce un milione di parole al secondo. Dal sistema si ottengono le seguenti misurazioni statistiche.

- L’1 per cento di tutte le istruzioni eseguite hanno avuto accesso a una pagina diversa dalla pagina corrente.
- L’80 per cento di queste istruzioni – che hanno cioè avuto accesso a un’altra pagina – hanno avuto accesso a una pagina già in memoria.
- Nel caso in cui sia stata richiesta una nuova pagina, la pagina sostituita è stata modificata nel 50 per cento dei casi.

Calcolate il tempo effettivo di esecuzione delle istruzioni di questo sistema, assumendo che il sistema stia eseguendo un unico processo e che il processore sia fermo durante i trasferimenti dal tamburo.

10.5 Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici a 12 bit con pagine di 256 byte mostrata di seguito.

La lista dei frame di pagina liberi è D, E, F (dove D è in testa alla lista, E al secondo posto, ed F è in coda). Un trattino al posto del frame di pagina indica che la pagina non è in memoria.

Pagina	Frame di pagina
0	-
1	2
2	C
3	A
4	-
5	4
6	3
7	-
8	B
9	0

Figura 10.31 Tabella delle pagine per l’Esercizio 10.5.

Convertite i seguenti indirizzi virtuali nei corrispondenti indirizzi fisici, in esadecimale. Tutti i numeri dati sono esadecimali. (Il trattino nella colonna dei frame di pagina indica che la pagina non è in memoria.)

- 9EF
- 111
- 700
- 0FF

10.6 Analizzate le funzionalità hardware necessarie alla paginazione su richiesta.

10.7 Considerate un array bidimensionale A:

```
int A [] [] = new int[100][100];
```

dove `A[0][0]` si trova nella posizione 200 in un sistema di memoria paginato con pagine di dimensione 200. Un piccolo processo che manipola la matrice risiede alla pagina 0 (posizioni da 0 a 199); ogni fetch di istruzioni avverrà così dalla pagina 0.

Per tre frame di pagina, quanti page fault vengono generati dai seguenti cicli di inizializzazione dell'array? Utilizzate la sostituzione lru e ipotizzate che un frame contenga il processo e gli altri due frame siano inizialmente vuoti.

```
a.   for (int j = 0; j < 100; j++)  
  
      for (int i = 0; i < 100; i++)  
  
         A[i][j] = 0;  
  
  
b.   for (int i = 0; i < 100; i++)  
  
      for (int j = 0; j < 100; j++)  
  
         A[i][j] = 0;
```

10.8 Considerate la seguente successione di riferimenti a pagine di memoria:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Quante eccezioni di page fault si verificherebbero per i seguenti algoritmi di sostituzione, assumendo uno, due, tre, quattro, cinque, sei e sette frame? Ricordate che tutti i frame sono inizialmente vuoti, per cui le vostre prime pagine uniche costeranno un'eccezione ciascuna.

- Sostituzione lru.
- Sostituzione fifo.
- Sostituzione ottimale.

10.9 Considerate la seguente successione di riferimenti alle pagine:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.

Assumendo di utilizzare la paginazione su richiesta con tre frame, quanti page fault si verificherebbero con i tre seguenti algoritmi di sostituzione?

- lru.
- fifo.
- Ottimale.

10.10 Supponete di voler utilizzare un algoritmo di paginazione che richiede un bit di riferimento (come nella sostituzione alla seconda chance o nel modello del working set) che non viene però fornito dall'hardware. Delineate come potreste simulare un bit di riferimento anche se non fornito dall'hardware, oppure spiegate perché non è possibile mettere in pratica una tale ipotesi. Se possibile, calcolate il costo di questa soluzione.

10.11 Avete progettato un nuovo algoritmo per la sostituzione delle pagine che pensate possa essere ottimale. In alcuni complicati test di controllo si verifica l'anomalia di Belady. L'algoritmo può essere considerato ottimale? Argomentate la vostra risposta.

10.12 La segmentazione è simile alla paginazione, ma utilizza "pagine" di dimensione variabile. Definite due algoritmi di sostituzione dei segmenti basati sugli schemi di sostituzione delle pagine fifo e lru. Ricordate che, siccome i segmenti non hanno la stessa dimensione, il segmento scelto per essere sostituito può essere troppo piccolo per poter contenere abbastanza locazioni di memoria consecutive per il segmento richiesto. Considerate strategie per sistemi nei quali i segmenti non possono essere rilocati e per sistemi nei quali ciò è invece possibile.

10.13 Considerate un sistema informatico a paginazione su richiesta nel quale il grado di multiprogrammazione sia attualmente fissato a quattro. Il sistema è stato recentemente sottoposto a misurazioni volte a determinare l'utilizzo del processore e del disco di paginazione. Le alternative che seguono mostrano tre possibili risultati. Per ognuno di questi casi, che cosa sta avvenendo? Il livello di multiprogrammazione può essere incrementato per migliorare l'utilizzo del processore? La paginazione sta dimostrandosi utile?

- a. Utilizzo del processore 13 per cento; utilizzo del disco 97 per cento.
- b. Utilizzo del processore 87 per cento; utilizzo del disco 3 per cento.
- c. Utilizzo del processore 13 per cento; utilizzo del disco 3 per cento.

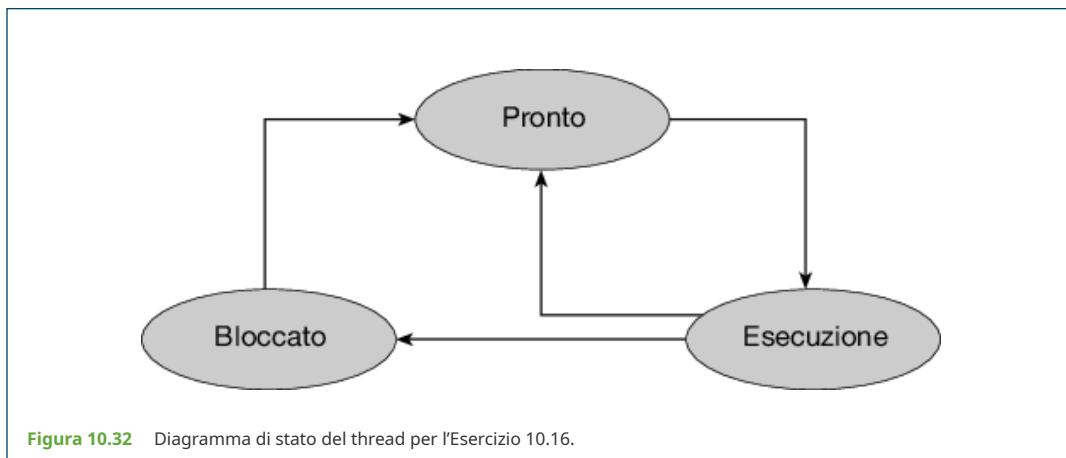
10.14 Considerate un sistema operativo per una macchina che utilizza registri base e limite, ma supponete di aver modificato la macchina di modo che metta a disposizione una tabella delle pagine. Si possono configurare le tabelle in modo da simulare registri base e limite? Come? Oppure, perché la cosa non è possibile?

Esercizi

10.15 Supponete che un programma abbia appena fatto riferimento a un indirizzo nella memoria virtuale. Descrivete uno scenario nel quale si verifichi ognuno dei seguenti eventi. (Se non è possibile che si verifichi un particolare scenario, spiegatene il motivo).

- Insuccesso del tlb senza page fault.
- Insuccesso del tlb con page fault.
- Successo del tlb senza page fault.
- Successo del tlb con page fault.

10.16 Una visione semplificata degli stati di un thread è Pronto (*Ready*), Esecuzione (*Running*) e Bloccato (*Blocked*), dove un thread è pronto e in attesa di essere schedulato, oppure è in esecuzione nel processore, oppure è bloccato (per esempio è in attesa di i/o), come illustrato nella Figura 10.32. Assumendo che un thread si trovi nello stato di Esecuzione, rispondete alle seguenti domande, argomentando le risposte.



- Il thread cambierà di stato se incorrerà in un page fault? In caso affermativo, quale sarà il nuovo stato?
- Il thread cambierà di stato se genererà un insuccesso del tlb che viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?
- Il thread cambierà di stato se il riferimento all'indirizzo viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?

10.17 Considerate un sistema che utilizza la paginazione su richiesta pura.

- Quando un processo inizia a essere eseguito, come caratterizzereste il tasso di page fault?
- Una volta che il working set di un processo viene caricato nella memoria, come caratterizzereste il tasso di page fault?
- Supponete che un processo cambi la propria località e che la dimensione del nuovo working set sia troppo grande per essere caricata nella memoria libera disponibile. Identificate alcune opzioni che i progettisti di sistemi potrebbero scegliere per gestire questa situazione.

10.18 Quella che segue è una tabella delle pagine per un sistema con indirizzi virtuali e fisici di 12 bit e pagine da 256 byte. I frame di pagina liberi devono essere assegnati nell'ordine 9, F, D. Un trattino per il page frame indica che la pagina non è in memoria.

Pagina	Frame di pagina
0	0 x 4
1	0 x B

Pagina	Frame di pagina
2	0 x A
3	-
4	-
5	0 x 2
6	-
7	0 x 0
8	0 x C
9	0 x 1

Convertite i seguenti indirizzi virtuali nel loro equivalente indirizzo fisico. Tutti i numeri sono espressi in formato esadecimale. Nel caso di un page fault è necessario utilizzare uno dei frame liberi per aggiornare la tabella delle pagine e tradurre l'indirizzo logico nel suo corrispondente indirizzo fisico.

- 0x2A1
- 0x4E6
- 0x94A
- 0x316

10.19 Che cos'è la funzionalità della copiatura su scrittura? In quali circostanze l'uso di tale funzionalità è vantaggioso? Quale hardware è richiesto per implementarla?

10.20 Un elaboratore fornisce ai propri utenti uno spazio di memoria virtuale di 2^{32} byte. L'elaboratore dispone di 2^{22} byte di memoria fisica. La memoria virtuale è implementata tramite paginazione, e la dimensione delle pagine è di 4096 byte. Un processo utente genera l'indirizzo virtuale 11123456. Spiegate in che modo il sistema determina la corrispondente locazione fisica, distinguendo fra operazioni software e hardware.

10.21 Ipotizzate di avere una memoria paginata su richiesta. La tabella delle pagine è conservata in registri. Se un frame vuoto è disponibile o se la pagina sostituita non è modificata, per ovviare alla mancanza di una pagina sono necessari 8 millisecondi, mentre occorrono 20 millisecondi, qualora la pagina sostituita abbia subito modifiche. Il tempo di accesso alla memoria è pari a 100 nanosecondi.

Supponete che la pagina da sostituire subisca modifiche nel 70 per cento dei casi. Per un tempo effettivo di accesso non superiore a 200 nanosecondi, qual è il tasso massimo tollerabile di page fault?

10.22 Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici di 16 bit e pagine da 4.096 byte.

Pagina	Frame di pagines	Bit di riferimento
0	9	0
1	-	0
2	10	0
3	15	0
4	6	0
5	13	0
6	8	0
7	12	0
8	7	0
9	-	0
10	5	0
11	4	0
12	1	0
13	0	0
14	-	0

Pagina	Frame di pagine	Bit di riferimento
15	2	0

Il bit di riferimento per una pagina è impostato su 1 quando la pagina è stata referenziata. Periodicamente, un thread azzerà tutti i valori del bit di riferimento. Un trattino per il page frame indica che la pagina non è in memoria. L'algoritmo di sostituzione delle pagine è lru locale e tutti i numeri sono in decimale.

- a. Convertite i seguenti indirizzi virtuali (in esadecimale) negli indirizzi fisici equivalenti. Potete fornire le risposte in esadecimale o decimale. Impostate anche il bit di riferimento per la voce appropriata nella tabella delle pagine.

- 0x621C
- 0xF0A3
- 0xBC1A
- 0x5BAA
- 0xBA1

- b. Utilizzando gli indirizzi di cui sopra come guida, fornire un esempio di indirizzo logico (in esadecimale) che provoca un page fault.
- c. Quale set di frame di pagina verrà utilizzato dall'algoritmo lru per risolvere un page fault?

10.23 Quando manca una pagina, il processo che ha richiesto la pagina deve bloccarsi mentre aspetta che la pagina venga portata dal disco alla memoria fisica. Posto che esiste un processo con cinque thread a livello utente e che il mapping dei thread utente sui thread del kernel sia di molti a uno, se un thread utente incorre in un page fault quando accede al suo stack, allora anche gli altri thread utente appartenenti allo stesso processo sono coinvolti nel page fault – ossia devono anch'essi aspettare che la pagina mancante venga portata in memoria? Motivate la risposta.

10.24 Applicate gli algoritmi di sostituzione (1) fifo, (2) lru e (3) ottimale (opt) alle seguenti sequenze di riferimenti di pagina:

- 2, 6, 9, 2, 4, 2, 1, 7, 3, 0, 5, 2, 1, 2, 9, 5, 7, 3, 8, 5
- 0, 6, 3, 0, 2, 6, 3, 5, 2, 4, 1, 3, 0, 6, 1, 4, 2, 3, 5, 7
- 3, 1, 4, 2, 5, 4, 1, 3, 5, 2, 0, 1, 1, 0, 2, 3, 4, 5, 0, 1
- 4, 2, 1, 7, 9, 8, 3, 5, 2, 6, 8, 1, 0, 7, 2, 4, 1, 3, 5, 8
- 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0

Indicate il numero di page fault per ciascun algoritmo, assumendo paginazione su richiesta con tre frame.

10.25 Si supponga di monitorare la velocità con cui si muove il puntatore nell'algoritmo a orologio (che indica la pagina candidata per la sostituzione). Che cosa si può inferire sul sistema sapendo che:

- a. il puntatore si muove velocemente;
- b. il puntatore si muove lentamente.

10.26 Esamineate a quali condizioni l'algoritmo di sostituzione delle pagine meno frequentemente usate (lfu) genera un numero inferiore di page fault rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente (lru). Descrivete anche le circostanze nelle quali è vero il contrario.

10.27 Considerate a quali condizioni l'algoritmo di sostituzione delle pagine più frequentemente usate (mifu) genera un numero inferiore di page fault rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente (lru). Descrivete anche le circostanze nelle quali è vero il contrario.

10.28 Il sistema khie utilizza un algoritmo di sostituzione fifo per le pagine residenti, nonché un gruppo di frame liberi costituito dalle pagine recentemente usate. Supponete che il gruppo di frame liberi sia gestito utilizzando il criterio di sostituzione lru. Rispondete alle seguenti domande.

- a. Se si verifica il fault di una pagina che non si trova nel gruppo di frame, come si genera lo spazio libero per la pagina appena richiesta?
- b. Se si verifica il fault di una pagina che si trova nel gruppo di frame, come va impostata la pagina residente e in che modo deve essere gestito il gruppo di frame per far spazio alla pagina richiesta?

c. In che cosa degenera il sistema di paginazione se il numero delle pagine residenti è impostato a uno?

d. In che cosa degenera il sistema di paginazione se il numero delle pagine nel gruppo di frame è zero?

10.29 Considerate un sistema con paginazione su richiesta con le seguenti utilizzazioni:

utilizzo della cpu 20%

disco di paginazione 97,7%

altri dispositivi di i/o 5%

Indicate, fra le seguenti operazioni, quelle che consentono (o è probabile che consentano) di migliorare l'utilizzo della cpu:

a. installazione di una cpu più veloce;

b. installazione di un disco di paginazione più grande;

c. aumento del grado di multiprogrammazione;

d. riduzione del grado di multiprogrammazione;

e. installazione di una maggiore quantità di memoria centrale;

f. installazione di un disco più veloce o di più controllori di unità con dischi multipli;

g. aggiunta della prepaginazione agli algoritmi di fetch delle pagine;

h. aumento della dimensione delle pagine.

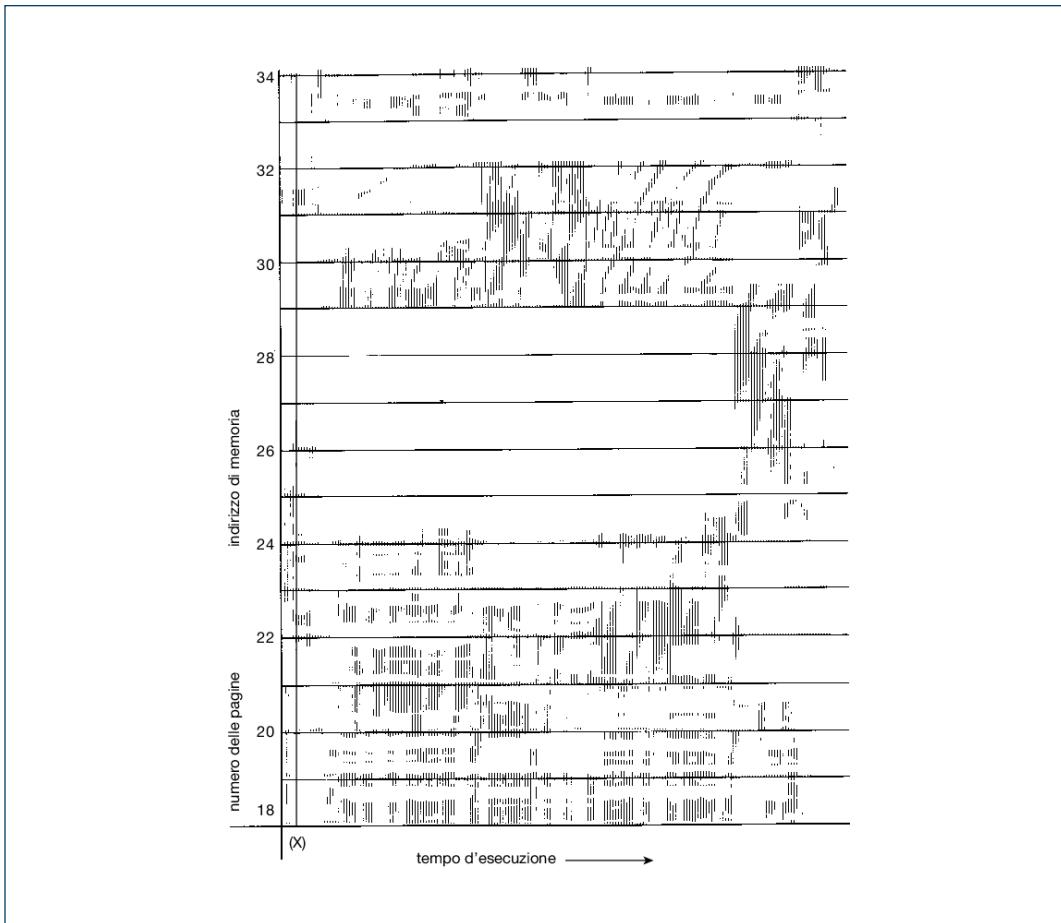
Motivate le risposte.

10.30 Spiegate come mai i page fault secondari richiedono meno tempo per essere risolti rispetto ai page fault principali.

10.31 Spiegate come mai viene utilizzata la compressione della memoria nei sistemi operativi per i dispositivi mobili

10.32 Supponete che una macchina fornisca istruzioni che possono accedere alle locazioni di memoria attraverso lo schema di indirizzamento indiretto a un livello. Quale sequenza di page fault si riscontra allorché tutte le pagine di un programma sono non residenti e la prima istruzione del programma è un'operazione di caricamento indiretto dalla memoria? Che cosa succede se il sistema adopera una tecnica di allocazione dei frame per processo e soltanto due pagine sono allocate al processo in questione?

10.33 Considerate i seguenti riferimenti alle pagine:



Quali pagine rappresentano la località al tempo (X)?

10.34 Si supponga che il criterio di sostituzione (in un sistema paginato) consista nel controllo regolare delle pagine, una per volta, eliminando ogni pagina che non sia stata usata dopo l'ultimo controllo. Che cosa offre in più tale criterio, e che cosa in meno, se paragonato all'algoritmo di sostituzione lru o con seconda chance?

10.35 Un algoritmo di sostituzione delle pagine dovrebbe ridurre al minimo il numero di page fault. Questa minimizzazione si può ottenere distribuendo in modo uniforme su tutta la memoria le pagine maggiormente usate, anziché lasciarle competere per un piccolo numero di frame. A ogni frame si può associare un contatore del numero delle pagine relative a quel frame. Quindi, per sostituire una pagina, si cerca il frame con il contatore più basso.

a. Definite un algoritmo di sostituzione delle pagine che si avvalga di questa idea di base. Affrontate in modo specifico i seguenti problemi:

1. qual è il valore iniziale dei contatori;
2. quando si incrementano i contatori;
3. quando si decrementano i contatori;
4. come si sceglie la pagina da sostituire.

b. Se sono disponibili quattro frame, dite quanti page fault avvengono per l'algoritmo che avete progettato, con la seguente successione di riferimenti:

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. Calcolate il numero minimo di page fault per una strategia di sostituzione delle pagine ottimale per la successione di riferimenti del punto b), con quattro frame.

10.36 Considerate un sistema di paginazione su richiesta con un disco di paginazione che abbia un tempo medio d'accesso e di trasferimento di 20 millisecondi. Gli indirizzi sono tradotti per mezzo di una tabella delle pagine che si trova in memoria centrale, con un tempo d'accesso di un microsecondo per ogni accesso alla memoria. Quindi, ogni riferimento alla memoria per mezzo della tabella delle pagine richiede due accessi. Per migliorare questo tempo, è stata aggiunta una memoria associativa che riduce il tempo d'accesso a un riferimento alla memoria, se l'elemento della tabella delle pagine si trova in memoria associativa.

Supponete che, per l'80 per cento degli accessi, l'elemento relativo si trovi in memoria associativa e che il 10 per cento dei restanti (cioè il 2 per cento del totale) causi un page fault. Calcolate il tempo effettivo d'accesso alla memoria.

10.37 Qual è la causa del thrashing? Come può il sistema accertarlo? E, una volta rivelato questo problema, che cosa può fare per eliminarlo?

10.38 Chiarite se un processo possa avere due working set, uno per rappresentare i dati e l'altro per rappresentare il codice.

10.39 Considerate il parametro D usato per definire la finestra del working set nell'ambito del modello omonimo. Impostando D a un valore basso, quale effetto ne deriva per la frequenza degli errori dovuti a page fault e per il numero di processi attivi (non sospesi) in esecuzione nel sistema? Qual è l'effetto quando D è impostato a un valore molto alto?

10.40 Ipotizzate di avere un segmento iniziale da 1024 kb allocato con il sistema buddy. Seguendo la Figura 10.26 come guida, tracciate l'albero che rappresenta l'allocazione di memoria derivante dalle richieste seguenti:

- richiesta di 5 kb;
- richiesta di 135 kb;
- richiesta di 14 kb;
- richiesta di 3 kb;
- richiesta di 12 kb.

Modificate adesso l'albero in conformità ai seguenti rilasci di memoria; applicate la fusione ogni qual volta è possibile:

- rilascio di 3 kb;
- rilascio di 5 kb;
- rilascio di 14 kb;
- rilascio di 12 kb.

10.41 Considerate un sistema in grado di gestire thread sia a livello utente sia a livello kernel. Il mappaggio in questo sistema è di uno a uno (a ogni thread del kernel corrisponde un thread utente). Un processo a più thread consiste allora di (a) un working set per l'intero processo, oppure di (b) un working set per ciascun thread?

10.42 L'algoritmo di allocazione delle lastre (*slab*) riserva una cache a ciascun oggetto di tipo diverso. Assumendo di avere una cache per tipo di oggetto, spiegate perché il metodo non scala bene su sistemi multiprocessore. Quale potrebbe essere la soluzione a tale problema di scalabilità?

10.43 Considerate un sistema che assegna ai propri processi pagine di dimensioni differenti. Quali vantaggi presenta tale schema di paginazione? Quali sono le modifiche da apportare al sistema di memoria virtuale per ottenere questa funzionalità?

10.44 Scrivete un programma che implementi gli algoritmi di sostituzione delle pagine fifo, lru e ottimale (opt) descritti nel Paragrafo 10.4. Generate inizialmente una successione di riferimenti casuale, in cui i numeri delle pagine siano compresi tra 0 e 9. Applicate ciascun algoritmo a tale successione e registrate i numeri di page fault che vengono generati. Passate il numero di page frame all'inizio del programma. Potete realizzare questo programma in un linguaggio a vostra scelta. (Potrete trovare utile la vostra implementazione dell'algoritmo fifo o lru per il progetto di programmazione relativo al gestore di memoria virtuale). Questo progetto consiste nell'implementazione di un programma che traduce indirizzi logici in indirizzi fisici per uno spazio di indirizzamento virtuale di dimensione $2^{16} = 65.536$ byte. Il programma leggerà da un file contenente indirizzi logici e, utilizzando un tlb e una tabella delle pagine, tradurrà ogni indirizzo logico nel corrispondente indirizzo fisico e restituirà il valore del byte memorizzato all'indirizzo fisico risultante. L'obiettivo principale di questo progetto è di simulare i passi necessari per tradurre indirizzi logici in indirizzi fisici. Questo comprenderà la risoluzione dei page fault utilizzando la paginazione su richiesta, gestire un tlb e implementare un algoritmo di sostituzione delle pagine.

Specifiche

Il vostro programma leggerà un file contenente diversi numeri interi a 32 bit che rappresentano indirizzi logici. Tratteremo tuttavia soltanto indirizzi a 16 bit: è quindi necessario mascherare i 16 bit più a destra di ogni indirizzo logico. Questi 16 bit sono suddivisi in (1) un numero di pagina di 8 bit e (2) un offset (scostamento) di pagina di 8 bit. La struttura degli indirizzi è la seguente:

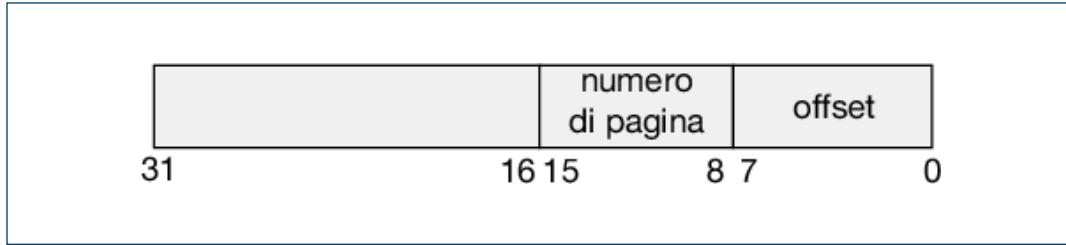
Le altre specifiche sono le seguenti:

- 2^8 elementi nella tabella delle pagine
- dimensione delle pagine di 2^8 byte
- 16 elementi nel tlb
- dimensione dei frame di 2^8 byte
- 256 frame
- memoria fisica di 65.536 byte (256 frame \times 256 byte di dimensione di ogni frame).

Inoltre, il programma deve preoccuparsi soltanto della lettura di indirizzi logici e della loro traduzione nei corrispondenti indirizzi fisici. Non è richiesto di supportare la scrittura sullo spazio di indirizzamento logico.

Traduzione degli indirizzi

Il programma tradurrà indirizzi logici in indirizzi fisici utilizzando un tlb e una tabella delle pagine, come descritto nel Paragrafo 9.3. Per prima cosa viene estratto dall'indirizzo logico il numero di pagina e si consulta il tlb. In caso di successo si ottiene il numero di frame dal tlb; in caso di insuccesso occorre consultare la tabella delle pagine. In quest'ultimo caso, si ricava dalla tabella delle pagine il numero di frame, oppure si verifica un page fault. Nella figura viene rappresentato graficamente il processo di traduzione degli indirizzi.



Gestione dei page fault

Il programma implementerà la paginazione su richiesta, come descritto nel Paragrafo 10.2. L'archivio di backup è rappresentato dal file `BACKING_STORE.bin`, un file binario di dimensione 65.536 byte. Quando si verifica un errore di pagina occorre leggere una pagina di 256 byte del `BACKING_STORE` e memorizzarla in un frame disponibile della memoria fisica. Per esempio, se un indirizzo logico con numero di pagina 15 ha provocato un page fault, il programma legge dal `BACKING_STORE` la pagina 15 (ricordate che le pagine cominciano da 0 e hanno dimensione di 256 byte) e salva la pagina in un frame della memoria fisica. Una volta che il frame è memorizzato (e sono aggiornate la tabella delle pagine e il tlb), i successivi accessi alla pagina 15 saranno risolti grazie al tlb o alla tabella delle pagine.

Sarà necessario trattare `BACKING_STORE.bin` come un file ad accesso casuale, in modo da poter accedere in lettura in modo casuale a diverse posizioni nel file. Consigliamo di utilizzare, per realizzare l'I/O, le funzioni della libreria standard C, tra cui `fopen()`, `fread()`, `fseek()` e `fclose()`.

La dimensione della memoria fisica è la stessa della dimensione dello spazio degli indirizzi virtuali (65.536 byte), non c'è quindi bisogno di preoccuparsi della sostituzione di pagina quando si verifica un page fault. Proporremo più avanti una versione modificata del presente progetto in cui si utilizza una minore quantità di memoria fisica. In tal caso sarà necessaria una strategia di sostituzione delle pagine.

File di test

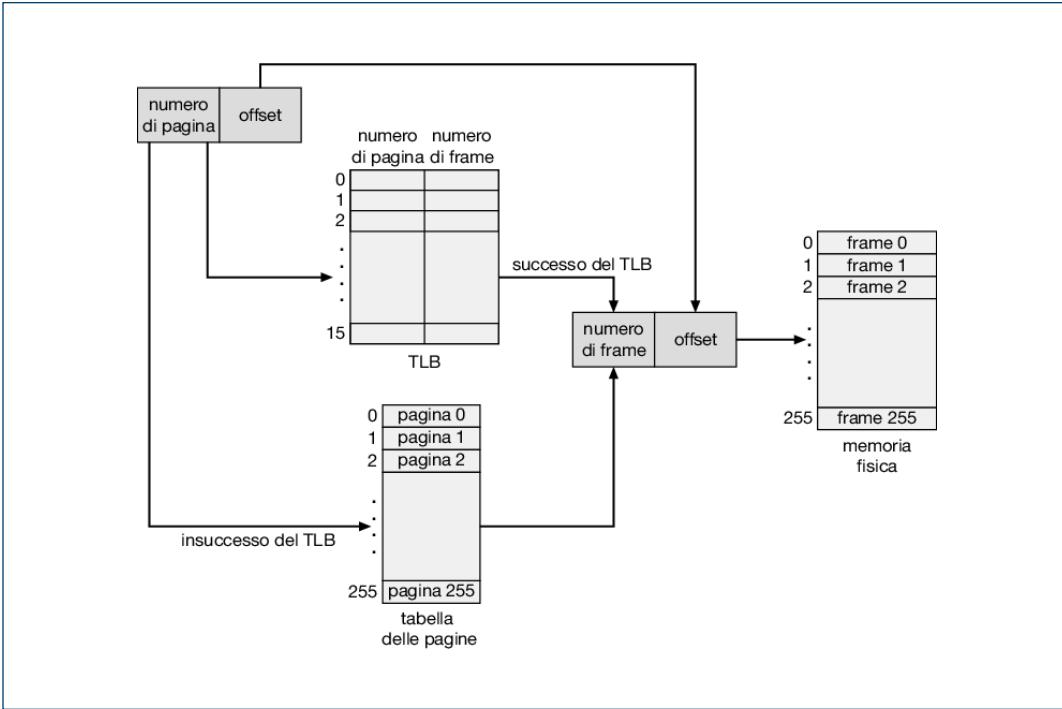
Viene fornito il file `addresses.txt` che contiene valori interi che rappresentano indirizzi logici tra 0 e 65.535 (la dimensione dello spazio di indirizzamento virtuale). Il vostro programma aprirà questo file, leggerà ogni indirizzo logico e lo tradurrà nel corrispondente indirizzo fisico, producendo in output il valore del byte che si trova all'indirizzo trovato.

Come iniziare

In primo luogo, scrivete un semplice programma che estrae il numero di pagina e l'offset, dai seguenti numeri interi

1, 256, 32768, 32769, 128, 65534, 33153

basandovi su



Probabilmente il modo più semplice per farlo è quello di utilizzare gli operatori per il mascheramento e lo shift dei bit. Una volta che siete riusciti a ricavare correttamente il numero di pagina e l'offset da un numero intero siete pronti per iniziare.

Suggeriamo di iniziare senza considerare il tlb e utilizzando soltanto una tabella delle pagine. È possibile integrare il tlb in un secondo tempo, una volta che la tabella delle pagine funziona correttamente.

Ricordate che la traduzione degli indirizzi può funzionare senza un tlb: il tlb rende solo più veloce il processo. Quando siete pronti per inserire il tlb, ricordate che ha solo 16 voci e che quindi sarà necessario utilizzare una strategia di sostituzione quando si aggiorna un tlb pieno. Potete aggiornare il vostro tlb utilizzando una politica fifo o una politica lru.

Come eseguire il programma

Il vostro programma deve essere lanciato nel modo seguente:

```
./ a.out addresses.txt
```

Il programma leggerà nel file `addresses.txt`, che contiene 1000 indirizzi logici compresi tra 0 e 65.535, convertirà ogni indirizzo logico in un indirizzo fisico e determinerà il contenuto del byte con segno memorizzato all'indirizzo fisico corretto. Ricordiamo che nel linguaggio C il tipo di dati char occupa un byte di memoria: si consiglia dunque di utilizzare valori char.

Il vostro programma deve restituire i seguenti valori:

1. L'indirizzo logico che viene tradotto (il valore intero letto dal file `addresses.txt`).
2. L'indirizzo fisico corrispondente (il risultato della traduzione dell'indirizzo logico).
3. Il valore del byte con segno memorizzato all'indirizzo fisico tradotto.

Viene fornito anche il file `correct.txt`, che contiene l'output corretto per il file `addresses.txt`. Dovete utilizzare questo file per verificare se il programma traduce correttamente gli indirizzi logici in indirizzi fisici.

Statistiche

Al termine dell'esecuzione il vostro programma dovrà produrre un report con le seguenti statistiche.

1. Page-fault rate: la percentuale di riferimenti a indirizzi che hanno provocato errori di page fault.
2. Tasso di successo del tlb (tlb hit rate): la percentuale di riferimenti a indirizzi che sono stati risolti nel tlb.

Dal momento che gli indirizzi logici in `addresses.txt` sono stati generati casualmente e non seguono alcun principio di località, non

	numero di pagina	offset
aspettatevi di avere un elevato tasso di successo del tlb.	31	16 15 8 7 0

Sostituzione delle pagine

Questo progetto presuppone che la memoria fisica abbia la stessa dimensione dello spazio degli indirizzi virtuali, mentre in realtà la memoria fisica è tipicamente molto più piccola di uno spazio degli indirizzi virtuali. Questa fase del progetto assume di utilizzare uno spazio di indirizzamento fisico più piccolo, con 128 frame piuttosto che 256. Tale cambiamento richiede di modificare il vostro programma in modo che tenga traccia dei frame liberi e che implementi una politica di sostituzione delle pagine di tipo fifo o lru (si veda il Paragrafo 10.4) per risolvere i page fault quando non c'è memoria libera.

11.9 Sommario

- Nella maggior parte dei calcolatori le unità a dischi e i dispositivi di memoria non volatile sono i principali dispositivi di i/o di memoria secondaria. La memoria secondaria moderna è strutturata come un grande array monodimensionale di blocchi logici.
- I dispositivi di entrambi i tipi possono essere collegati a un computer in tre modi: (1) attraverso le porte di i/o locali del computer host, (2) direttamente alle schede madri o (3) tramite una rete di comunicazione o di storage.
- Le richieste di i/o sui dispositivi di memoria secondaria sono generate sia dal file system sia dai sistemi di memoria virtuale, e ognuna di esse specifica l'indirizzo cui fare riferimento sul dispositivo sotto forma di numero di un blocco logico.
- Gli algoritmi di scheduling per i dischi possono aumentare l'ampiezza di banda e ridurre il tempo di risposta medio e la variabilità del tempo di risposta. Algoritmi come scan e c-scan sono progettati per realizzare questi miglioramenti tramite criteri di ordinamento della coda di richieste di accesso ai dischi. Le prestazioni degli algoritmi di scheduling sui dischi magnetici possono variare notevolmente. Al contrario, nel caso dei dischi a stato solido, che non hanno parti in movimento, non ci sono grandi differenze di prestazioni tra i vari algoritmi e molto spesso viene utilizzata una semplice strategia fcfs.
- L'archiviazione e la trasmissione dei dati sono operazioni complesse e causano spesso errori. Il rilevamento degli errori tenta di individuare tali problemi per avvisare il sistema della necessità di azioni correttive e per evitare la propagazione degli errori. La correzione degli errori può rilevare e talvolta riparare i problemi, a seconda della quantità di dati di ridondanza disponibili e della quantità di dati che è stata danneggiata.
- I dispositivi di archiviazione sono suddivisi in una o più partizioni. Ogni partizione può contenere un volume o essere parte di un volume più grande. Sui volumi vengono creati i file system.
- Il sistema operativo gestisce i blocchi di un dispositivo di memoria. I dispositivi nuovi sono generalmente venduti già formattati; in seguito, il disco può essere diviso in partizioni, si può creare il file system, e, se il dispositivo conterrà un sistema operativo, si possono assegnare i blocchi d'avvio che conterranno il bootstrap del sistema; infine, quando un blocco (o una pagina) diviene difettoso, il sistema deve essere in grado di isolarlo o di sostituirlo con un blocco (o una pagina) di riserva.
- Su alcuni sistemi un'area di swap efficiente è la chiave per ottenere buone prestazioni. Alcuni sistemi dedicano una partizione di basso livello all'area di swap, altri utilizzano un file all'interno del file system, altri ancora consentono all'utente o all'amministratore di sistema di prendere la decisione fornendo entrambe le opzioni.
- A causa della quantità di spazio di memorizzazione secondaria richiesta nei grandi sistemi e poiché i dispositivi di archiviazione subiscono diverse tipi di guasto, le unità sono spesso rese ridondanti tramite algoritmi raid. Questi algoritmi permettono l'uso di più unità per una data operazione, e consentono la prosecuzione del funzionamento del sistema e anche il ripristino automatico dei dati a fronte del guasto di un dispositivo. Gli algoritmi raid sono classificati in livelli che offrono diverse combinazioni di affidabilità e velocità di trasferimento.
- La memorizzazione di oggetti viene utilizzata per problemi di big data come l'indicizzazione di Internet e l'archiviazione di foto nel cloud. Gli oggetti sono raccolte di dati auto-descrittivi, indirizzate da un id piuttosto che dal nome di un file. Gli archivi di oggetti (object storage) utilizzano generalmente la replica per la protezione dei dati, eseguono la computazione sui sistemi in cui è presente una copia dei dati e sono scalabili orizzontalmente, ottenendo capacità notevoli e una facile espansione.

Esercizi di ripasso

11.1 Lo scheduling del disco, con algoritmi diversi dall'fcfs, è utile in un ambiente con un solo utente? Argomentate.

11.2 Spiegate perché l'sstf tende a favorire i cilindri centrali rispetto a quelli più interni e più esterni.

11.3 Perché la latenza di rotazione non viene solitamente presa in considerazione nello scheduling del disco? Come potreste modificare lo scheduling sstf, lo scan e quello c-scan per includervi l'ottimizzazione della latenza?

11.4 Perché è importante bilanciare gli i/o del file system tra i dischi e i controllori su un sistema in un ambiente multitasking?

11.5 Quali sono i tradeoff fra la rilettura delle pagine di codice da un file system e l'utilizzo dell'area di avvicendamento per memorizzarli?

11.6 Esiste un modo per realizzare una memorizzazione delle informazioni veramente stabile? Argomentate la risposta.

11.7 A volte un nastro è detto mezzo ad accesso sequenziale, mentre un disco magnetico è considerato un mezzo ad accesso diretto. In realtà, l'idoneità di un dispositivo di memorizzazione all'accesso diretto dipende dalla grandezza del trasferimento. Il termine *velocità di trasferimento in streaming* denota la velocità di un trasferimento di dati che è in corso, escluso l'effetto della latenza di accesso. La *velocità effettiva di trasferimento*, invece, è il rapporto dei byte totali per il totale dei secondi, inclusi i tempo di overhead come la latenza di accesso.

Ipotizzate che, in un computer, la cache di livello 2 abbia una latenza di accesso di 8 nanosecondi e una velocità di trasferimento in streaming di 800 megabyte al secondo, che la memoria principale abbia una latenza di accesso di 60 nanosecondi e una velocità di trasferimento in streaming di 80 megabyte al secondo, che il disco magnetico abbia una latenza di accesso di 15 millisecondi e una velocità di trasferimento in streaming di 5 megabyte al secondo, e che una unità a nastro abbia una latenza di accesso di 60 secondi e una velocità di trasferimento in streaming di 2 megabyte al secondo.

a. L'accesso diretto causa la diminuzione della velocità effettiva di trasferimento di un dispositivo, perché non vengono trasferiti dati durante il tempo di accesso. Per il disco descritto, qual è la velocità di trasferimento effettiva se in media un accesso è seguito da un trasferimento in streaming di (1) 512 byte, (2) 8 kilobyte, (3) 1 megabyte, e (4) 16 megabyte?

b. L'utilizzazione di un dispositivo è definita come il rapporto tra la velocità effettiva di trasferimento e la velocità di trasferimento in streaming. Calcolate l'utilizzazione dell'unità a disco per ognuna delle quattro grandezze di trasferimento indicate al punto a.

c. Supponete che un'utilizzazione del 25 per cento o più sia considerata accettabile. Utilizzando i valori di prestazione indicati, calcolate la dimensione minima di trasferimento per un disco che dia un'utilizzazione accettabile.

d. Completate la frase seguente: Un disco è un dispositivo ad accesso diretto per trasferimenti superiori a _____ byte ed è un dispositivo ad accesso sequenziale per trasferimenti inferiori.

e. Calcolate le dimensioni minime di trasferimento che diano utilizzazioni accettabili per cache, memoria e nastro.

f. Quando un nastro può essere considerato un dispositivo ad accesso diretto e quando invece è un dispositivo ad accesso sequenziale?

11.8 Può un'organizzazione raid a livello 1 ottenere prestazioni migliori sulle richieste di lettura rispetto a un'organizzazione raid a livello 0 (con striping senza ridondanza)? Se sì, come?

11.9 Fornite tre motivi per utilizzare gli hdd come memoria secondaria.

11.10 Fornite tre motivi per utilizzare i dispositivi nvm come memoria secondaria.

Esercizi

11.11 Eccetto l'fcfs, nessuno tra i criteri di scheduling del disco descritti è veramente *equo* (si può avere un'attesa indefinita).

- Spiegate perché questa affermazione è vera.
- Descrivete una maniera di modificare gli algoritmi come lo scan in modo che risultino equi.
- Spiegate perché l'equità è un obiettivo importante in un sistema a partizione del tempo.
- Date tre o più esempi di circostanze in cui è importante che il sistema operativo non sia equo nel servire le richieste di i/o.

11.12 Spiegate perché per i dischi ssd viene spesso utilizzato un algoritmo di scheduling fcfs.

11.13 Supponete che un'unità a disco abbia 5000 cilindri numerati da 0 a 4999. L'unità serve attualmente una richiesta relativa al cilindro 2150, e la richiesta precedente era relativa al cilindro 1850. La coda di richieste innevase, in ordine fifo, è composta di richieste riguardanti i cilindri

2069, 1212, 2296, 2800, 544, 1618, 356, 1523, 4965, 3681

Partendo dalla posizione attuale della testina, calcolate la distanza totale (in cilindri) che il braccio del disco percorre per soddisfare tutte le richieste innevase usando i seguenti algoritmi di scheduling:

- fcfs;
- scan;
- c-scan.

11.14 È noto dalla fisica elementare che quando un oggetto è sottoposto a un'accelerazione costante a , la relazione fra la distanza d e il tempo t è data da $d = 1/2 at^2$. Supponete che l'unità a disco dell'Esercizio 11.13, durante la ricerca di un cilindro, imprima al braccio del disco un'accelerazione costante per la prima metà del tragitto richiesto dalla ricerca, e imprima invece una decelerazione costante della stessa intensità per la seconda metà del tragitto. Ipotizzate che l'unità possa portare a termine in 1 millisecondo la ricerca di un cilindro adiacente, e in 18 millisecondi una ricerca a tutto raggio lungo i 5000 cilindri.

- La distanza di una ricerca è il numero di cilindri attraverso i quali la testina deve passare. Spiegate perché il tempo di ricerca è proporzionale alla radice quadrata della distanza percorsa.
- Scrivete il tempo di ricerca in funzione della distanza da percorrere. L'equazione dovrebbe avere la forma $t = x + y \sqrt{L}$, dove t è il tempo in millisecondi e L è la distanza da percorrere in cilindri.
- Calcolate il tempo totale di ricerca per ogni algoritmo di scheduling dell'Esercizio 11.13 relativamente alla coda di richieste lì descritta. Determinate quale algoritmo sia più veloce, cioè implica un tempo di ricerca totale minore.
- L'aumento percentuale di velocità è il tempo risparmiato diviso il tempo originariamente necessario. Calcolate l'aumento percentuale di velocità dell'algoritmo più veloce rispetto all'fcfs.

11.15 Supponete che il disco dell'Esercizio 11.14 ruoti alla velocità di 7200 giri al minuto.

- Calcolate la latenza di rotazione media di quest'unità a disco.
- Dite quale distanza di ricerca è possibile coprire nel tempo calcolato al punto a).

11.16 Descrivete vantaggi e svantaggi dell'utilizzo di unità hdd e nvm. Quali sono le migliori applicazioni per ciascun tipo?

11.17 Descrivete alcuni vantaggi e svantaggi dell'impiego di dispositivi nvm come livello di caching piuttosto che come sostituzione dei dischi, confrontandolo con l'utilizzo dei soli hdd.

11.18 Confrontate le prestazioni di scan e c-scan assumendo una distribuzione uniforme delle richieste di i/o. Considerate il tempo di risposta medio (cioè il tempo che intercorre fra l'arrivo di una richiesta e il completamento dell'operazione a essa associata), la variazione del tempo di risposta, e l'effettiva ampiezza di banda. Analizzate come le prestazioni dipendano dalle dimensioni relative del tempo di ricerca e della latenza di rotazione.

11.19 Le richieste non sono di solito uniformemente distribuite; per esempio un cilindro che contiene metadati del file system potrebbe essere visitato più spesso di un cilindro che contiene solo file. Supponete di sapere che il 50 per cento delle richieste sia relativo a un piccolo numero fisso di cilindri.

- Dite se uno fra gli algoritmi illustrati in questo capitolo sia particolarmente adatto a questa circostanza. Motivate la risposta.
- Proponete un algoritmo di scheduling che offra prestazioni anche migliori sfruttando questo "punto caldo" del disco.

11.20 Considerate un sistema raid a livello 5, comprensivo di cinque dischi; nel quinto disco risiedono le parità per gli insiemi di quattro blocchi di quattro dischi. A quanti blocchi si deve accedere per effettuare le operazioni seguenti:

- scrittura di un blocco di dati;
- scrittura di sette blocchi contigui di dati.

11.21 Ponete a confronto il throughput ottenuto attraverso un sistema raid a livello 5 con quello ottenuto mediante un sistema raid a livello 1, per quel che riguarda:

- operazioni di lettura su blocchi singoli;

b. operazioni di lettura su blocchi multipli contigui.

11.22 Paragonate le prestazioni raggiunte da un sistema raid a livello 5 con quelle di un sistema raid a livello 1, relativamente alle operazioni di scrittura.

11.23 Assumete di avere una configurazione mista, che comprenda dischi strutturati secondo il sistema raid a livello 1, e altri dischi a livello raid 5. Supponete che il sistema, per memorizzare un determinato file, sia libero di optare per l'una o l'altra soluzione. Quali file dovrebbero essere memorizzati nei dischi a livello raid 1 e quali nei dischi a livello raid 5, allo scopo di ottimizzare le prestazioni?

11.24 L'affidabilità di un'unità a disco generalmente si quantifica usando il tempo medio fra due guasti (*mean time between failures*, mtbf); sebbene sia chiamata "tempo", questa quantità in effetti si misura in ore di funzionamento dell'unità per guasto.

- a. Dato un gruppo di 1000 unità a disco, ognuna delle quali ha un mtbf di 750.000 ore, dite con quale frequenza avverrà il guasto di un'unità del gruppo, scegliendo fra le seguenti possibilità quella che si adatta meglio alla situazione descritta: una volta ogni mille anni, una volta ogni cento anni, una volta ogni dieci anni, una volta al mese, una volta alla settimana, una volta al giorno, una volta ogni ora, una volta al minuto, una volta al secondo.
- b. Le statistiche di mortalità indicano che in media un individuo residente negli Stati Uniti d'America ha circa 1 probabilità su 1000 di morire fra i 20 e i 21 anni d'età. Deducete l'mtbf di un ventenne, e convertite il risultato da ore in anni. Spiegate che cosa dice questo mtbf sulle aspettative di vita di un ventenne.
- c. Un produttore asserisce che un certo modello di unità a disco abbia un mtbf di 1 milione di ore. Dite cosa si può concludere circa il numero di anni per cui una di queste unità a disco è coperta dalla garanzia.

11.25 Esponete vantaggi e svantaggi relativi all'accantonamento dei settori e alla traslazione dei settori.

11.26 Esponete i motivi per cui il sistema operativo potrebbe necessitare di informazioni accurate sulle modalità di memorizzazione dei blocchi sul disco. In termini di miglioramento delle prestazioni del file system, in che modo il sistema operativo può mettere a frutto queste informazioni?

11.27 Scrivete un programma che implementi i seguenti algoritmi di scheduling del disco:

- a. fcfs
- b. scan
- c. c-scan

Il vostro programma servirà un disco con 5000 cilindri numerati da 0 a 4.999. Il programma genererà una sequenza casuale di 1000 richieste e le servirà secondo ciascuno degli algoritmi sopra elencati. Al programma sarà passata, come parametro da riga di comando, la posizione iniziale della testina del disco. Il programma restituirà la quantità totale di movimenti della testina richiesti da ciascun algoritmo.

CAPITOLO 11

Memoria di massa

In questo capitolo vedremo com'è strutturata la memoria di massa, il sistema di memorizzazione non volatile di un computer. Il principale sistema di archiviazione di massa nei computer moderni è la memoria secondaria, che di solito è fornita da dischi rigidi (anche detti dischi fissi, hard disk, hdd, o semplicemente dischi) e da altri dispositivi di memoria non volatile (nvm). Alcuni sistemi dispongono inoltre di una memoria terziaria più lenta e più grande, generalmente costituita da nastri magnetici, dischi ottici o anche spazio di memorizzazione su cloud.

Poiché i dispositivi di archiviazione più comuni e importanti nei moderni sistemi elaborativi sono i dischi rigidi e i dispositivi nvm, la maggior parte di questo capitolo è dedicata alla trattazione di queste due tipologie di memoria di massa. Inizieremo descrivendo la loro struttura fisica e considereremo poi gli algoritmi di scheduling che pianificano l'ordine delle operazioni di i/o al fine di massimizzare le prestazioni. Successivamente, discuteremo della formattazione dei dispositivi e della gestione dei blocchi di avvio, dei blocchi danneggiati e dello spazio di swap. Infine, esamineremo la struttura dei sistemi raid.

Esistono molti tipi di memoria di massa e in questo testo utilizzeremo l'acronimo nvs (non-volatile storage), oppure il termine unità (o dispositivo) di memorizzazione, quando vogliamo includerli tutti, mentre sarà utilizzato il termine specifico appropriato nel caso in cui si stiano trattando particolari dispositivi, come i dischi rigidi e i dispositivi nvm.

11.1 Struttura dei dispositivi di memorizzazione

I dischi magnetici e i dispositivi nvm costituiscono i supporti fondamentali di memoria secondaria dei moderni computer. In questo paragrafo descriviamo i meccanismi di base di questi dispositivi e spieghiamo in che modo i sistemi operativi traducano le loro proprietà fisiche in memoria logica tramite la mappatura degli indirizzi.

11.1.1 Dischi rigidi

Concettualmente, i dischi (Figura 11.1) sono relativamente semplici: i piatti dei dischi hanno una forma piana e rotonda come quella dei cd, con un diametro che comunemente varia tra 1,8 e 3,5 pollici, e le due superfici ricoperte di materiale magnetico; le informazioni si memorizzano registrandole magneticamente sui piatti e vengono lette rilevando la configurazione magnetica memorizzata.

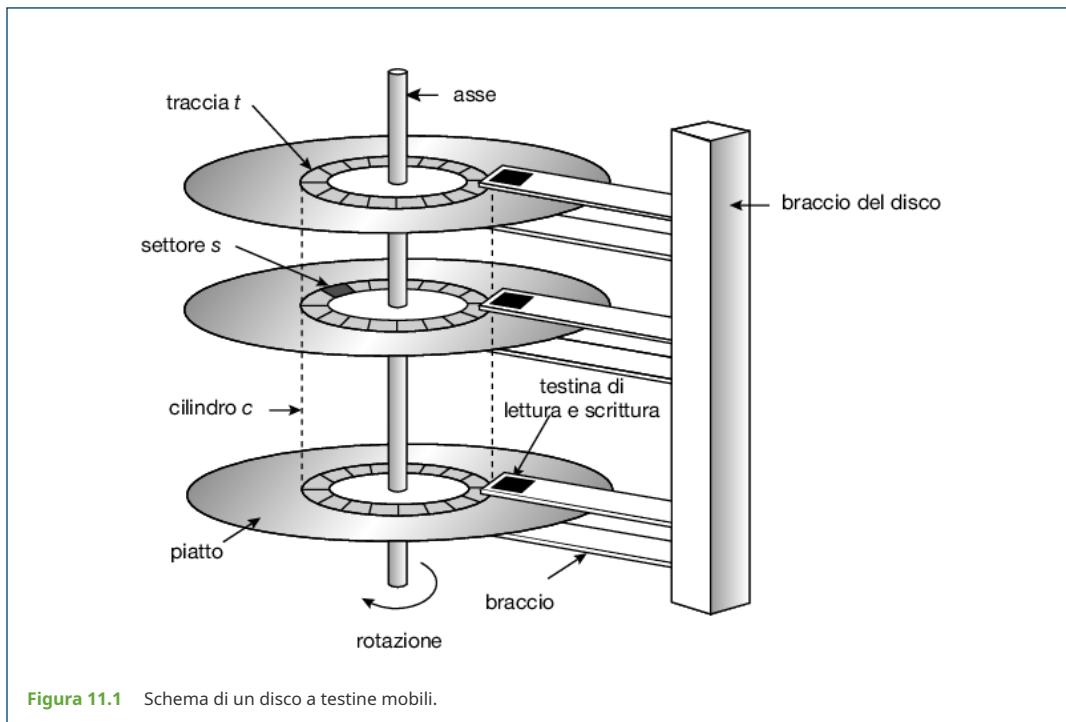


Figura 11.1 Schema di un disco a testine mobili.

Una testina di lettura e scrittura è sospesa su ciascuna superficie d'ogni piatto. Le testine sono attaccate al braccio del disco che le muove in blocco. La superficie di un piatto è divisa logicamente in tracce circolari a loro volta suddivise in settori; l'insieme delle tracce corrispondenti a una posizione del braccio costituisce un cilindro. In un'unità a disco possono esservi migliaia di cilindri concentrici e ogni traccia può contenere centinaia di settori. Ogni settore ha una dimensione fissa, che è la più piccola unità di trasferimento. Le dimensioni del settore erano in genere di 512 byte, ma dal 2010 molti produttori hanno iniziato a migrare verso settori di 4kb. La capacità di memorizzazione di una comune unità a disco si misura in gigabyte o terabyte. La Figura 11.2 mostra un disco rigido privato della sua copertura.

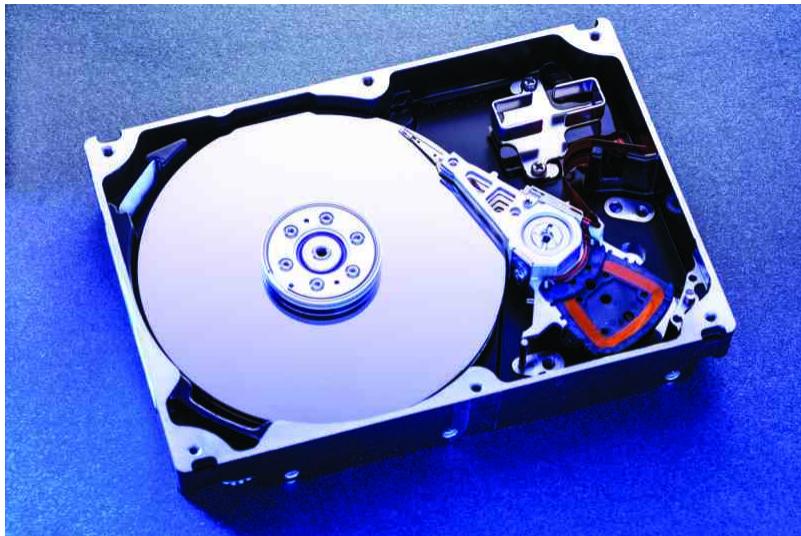


Figura 11.2 Disco da 3,5 pollici senza la sua copertura.

VELOCITÀ DI TRASFERIMENTO DEI DISCHI

Come per molti altri aspetti dell'informatica, le prestazioni pubblicate relative ai dischi non coincidono con le cifre reali: sono sempre inferiori delle **velocità di trasferimento effettive**, per esempio. La velocità di trasferimento può essere considerata la velocità con cui la testina legge i bit dal supporto magnetico, ma questa va distinta dalla velocità con cui i blocchi sono consegnati al sistema operativo.

Quando un disco è in funzione, un motore lo fa ruotare ad alta velocità; la maggior parte dei dischi ruota a velocità comprese tra 60 e 250 giri al secondo. Questa velocità viene espresa in termini di giri al minuto (rpm): i comuni dischi possono lavorare alle velocità di 5400, 7200, 10000 o 15000 rpm. Alcune unità si spengono quando non sono in uso e si accendono dopo aver ricevuto una richiesta di i/o. La velocità di rotazione è in relazione con la velocità di trasferimento (*transfer rate*), ovvero la velocità con cui i dati fluiscano dall'unità a disco al calcolatore. Un secondo aspetto relativo alle prestazioni è il tempo di posizionamento, detto anche tempo d'accesso casuale, che consiste di due componenti: il tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato, detto tempo di ricerca (*seek time*), e il tempo necessario affinché il settore desiderato si porti, tramite la rotazione del disco, sotto la testina, detto latenza di rotazione. In genere i dischi possono trasferire parecchi megabyte di dati al secondo e hanno un tempo di ricerca e una latenza di rotazione di diversi millisecondi. Le prestazioni possono aumentare grazie ai buffer dram presenti nel controllore dell'unità.

Poiché le testine di un disco sono sospese su un cuscino d'aria sottilissimo (dell'ordine dei micron), esiste il pericolo che la testina urti la superficie del disco; in tal caso, nonostante i piatti del disco siano ricoperti da un sottile strato protettivo, la testina può danneggiare la superficie magnetica. Tale incidente, detto urto della testina, che di solito non può essere riparato, comporta la sostituzione dell'intera unità a disco e la perdita dei dati sul disco, a meno che non siano stati salvati su un altro dispositivo o protetti mediante raid (il raid è discusso nel Paragrafo 11.8).

I dischi rigidi sono unità sigillate e alcuni *chassis* che li contengono consentono di rimuoverli senza spegnere il sistema. Ciò è utile quando un sistema ha bisogno di più spazio di archiviazione, perché lo si può aggiungere in qualsiasi momento, o quando è necessario sostituire un disco non funzionante con uno funzionante. Anche altre forme di supporti di memorizzazione sono rimovibili, inclusi i cd, i dvd e i dischi Blu-ray.

11.1.2 Dispositivi NVM

I dispositivi nvm stanno acquisendo un'importanza sempre maggiore. In sostanza, questi dispositivi sono elettrici invece che meccanici e, nella maggior parte dei casi, sono composti da un controllore e da diversi chip di memoria a semiconduttore nand flash utilizzati per memorizzare i dati. Esistono altre tecnologie nvm, come le dram dotate di una batteria di backup che permette di non perdere il contenuto o altre tecnologie basate su semiconduttori come le 3D XPoint, ma sono molto meno comuni e per questa ragione non verranno trattate in questo libro.

11.1.2.1 Panoramica dei dispositivi NVM

I dispositivi nvm basati su memoria flash vengono spesso inseriti in contenitori simili a unità disco, e per questa ragione sono chiamati dischi a stato solido, o ssd (Figura 11.3). In altri casi, un dispositivo nvm può assumere la forma di un'unità usb (nota anche come *pen drive* o *unità flash*) o di un modulo dram. In dispositivi come gli smartphone le nvm sono montate sulla superficie delle schede madri come dispositivo principale di archiviazione. In qualunque forma si presenti, un dispositivo nvm si comporta e può essere trattato allo stesso modo. La nostra discussione sui dispositivi nvm si concentra su questa tecnologia di memorizzazione.



Figura 11.3 Scheda di un ssd da 3,5 pollici.

I dispositivi nvm possono essere più affidabili dei dischi rigidi, perché non hanno parti mobili, e possono essere più veloci, perché non hanno tempo di posizionamento o latenza di rotazione. Inoltre, consumano meno energia. Per contro, sono più costosi (per megabyte) rispetto ai dischi rigidi tradizionali e hanno una capacità inferiore rispetto ai dischi rigidi più grandi presenti sul mercato. Nel corso del tempo, tuttavia, la capacità dei dispositivi nvm è aumentata più rapidamente rispetto a quella dei dischi rigidi e il loro prezzo è diminuito maggiormente; il loro utilizzo è quindi in continuo aumento e gli ssd, o dispositivi simili, sono ora utilizzati in alcuni computer portatili per renderli più piccoli, più veloci e più efficienti dal punto di vista energetico.

Poiché i dispositivi nvm possono essere molto più veloci rispetto ai dischi fissi, le interfacce bus standard possono costituire un limite importante al throughput e alcuni dispositivi nvm sono progettati per connettersi direttamente al bus di sistema (a un bus pcie, per esempio). La tecnologia nvm sta cambiando anche altri aspetti tradizionali del progetto di un'architettura. Alcuni sistemi usano dispositivi nvm come sostituto diretto delle unità disco, mentre altri li usano come un nuovo livello di cache, spostando i dati tra dischi magnetici, nvm e memoria principale per ottimizzare le prestazioni.

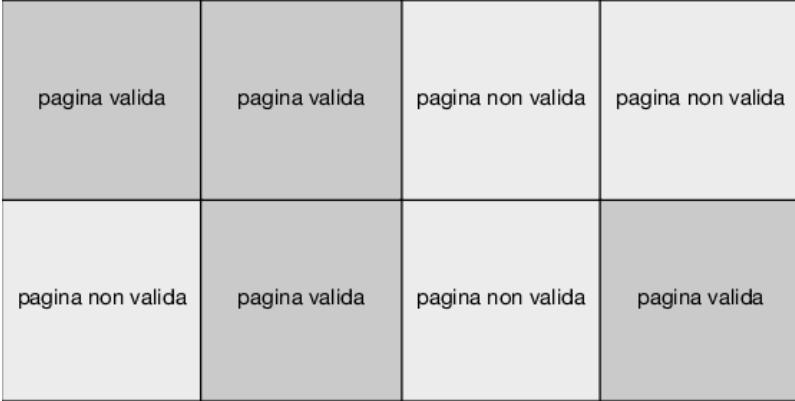
Le caratteristiche dei semiconduttori nand portano alcune nuove sfide che riguardano memorizzazione e affidabilità. Per esempio, possono essere letti e scritti a livello di pagina (l'analogo di un settore di un disco), ma i dati non possono essere sovrascritti senza che le celle nand siano prima cancellate. La cancellazione, che si verifica a livello di blocchi della dimensione di diverse pagine, richiede molto più tempo di una lettura (l'operazione più veloce) o di una scrittura (più lenta della lettura, ma molto più veloce della cancellazione). Il problema è però alleviato dal fatto che i dispositivi flash nvm sono composti da molte piastrine, dette die, ciascuna con molti percorsi dati (datapath), e le operazioni possono quindi avvenire in parallelo (ciascuna su un percorso dati). I semiconduttori nand, inoltre, si deteriorano a ogni ciclo di cancellazione e dopo circa 100.000 cicli (il numero specifico varia a seconda del supporto) le celle non sono più in grado di conservare i dati. A causa dell'usura nella scrittura, e poiché non vi sono parti in movimento, la durata delle nvm nand non viene misurata in anni, ma in numero di scritture per giorno sull'unità (dwpd, *Drive Writes per Day*). Tale numero indica quante volte al giorno l'intera capacità dell'unità può essere scritta prima che l'unità si guasti. Per esempio, su un dispositivo nand da 1 tb con una classificazione di 5 dwpd dovrebbe essere possibile scrivere 5 tb al giorno per tutto il periodo di garanzia senza che si verifichino errori.

Queste limitazioni hanno portato allo sviluppo di diversi algoritmi di miglioramento che, fortunatamente, vengono di solito implementati nel controllore del dispositivo nvm e non sono di interesse per il sistema operativo. Il sistema operativo si limita

semplicemente a leggere e scrivere blocchi logici, mentre il dispositivo si occupa dell'effettiva realizzazione delle operazioni (i blocchi logici sono discussi più dettagliatamente nel Paragrafo 11.1.5). Tuttavia, i dispositivi nvm sono soggetti a variazioni delle prestazioni dovute ai loro algoritmi operativi e sarà quindi indispensabile una breve discussione su ciò che fa il controllore.

11.1.2.2 Algoritmi del controllore delle NAND Flash

Poiché i semiconduttori nand non possono essere sovrascritti, sono di solito presenti pagine che contengono dati non validi. Considerate un blocco del file system scritto una volta e in seguito riscritto. Se nel frattempo non si è verificata alcuna cancellazione, la pagina scritta per prima contiene vecchi dati, che ora non sono più validi, mentre la seconda pagina ha la versione attuale e valida del blocco. Un blocco nand contenente pagine valide e non valide è mostrato nella Figura 11.4. Per tenere traccia di quali blocchi logici contengono dati validi, il controllore mantiene una tabella di traduzione flash (ftl, *flash translation layer*), che tiene traccia delle pagine fisiche contenenti blocchi logici validi e dello stato del blocco fisico, ovvero dei blocchi che contengono solo pagine non valide e pertanto cancellabili.



The diagram illustrates a 2x4 grid representing a NAND block. The top row contains four cells labeled "pagina valida" (valid page). The bottom row contains four cells labeled "pagina non valida" (invalid page). This visualizes how a physical page can be marked as invalid even if it still contains valid data.

Figura 11.4 Blocco nand con pagine valide e non valide.

Considerate ora un ssd pieno con una richiesta di scrittura in sospeso. Anche se l'ssd è pieno, ovvero tutte le pagine sono state scritte, potrebbe esserci un blocco contenente dati non validi. In tal caso, dopo aver aspettato la fine della cancellazione è possibile effettuare la scrittura. Ma che cosa succede se non ci sono blocchi liberi? Potrebbe ancora esserci spazio disponibile se alcune pagine contenessero dati non validi. In tal caso, può essere effettuata la garbage collection: i dati validi vengono copiati in altre posizioni, liberando così blocchi cancellabili e riutilizzabili per nuove scritture. Ma dove possono essere copiati i dati validi durante l'operazione di garbage collection? Per risolvere questo problema e migliorare le prestazioni in scrittura, il dispositivo di memoria non volatile utilizza un tecnica detta over-provisioning: un certo numero di pagine (solitamente il 20 percento del totale) viene messo da parte per fornire un'area sempre disponibile per la scrittura. I blocchi che diventano interamente non validi in seguito a una garbage collection o a operazioni di scrittura che invalidano le versioni precedenti dei dati vengono cancellati e inseriti nello spazio di over-provisioning se il dispositivo è pieno, o altrimenti nella lista dei blocchi liberi.

Lo spazio di over-provisioning può anche aiutare a rendere omogenea l'usura del dispositivo. Se alcuni blocchi vengono cancellati ripetutamente, mentre altri non lo sono, i blocchi frequentemente cancellati si logorano più velocemente degli altri e l'intero dispositivo avrà una vita più breve di quanto sarebbe se tutti i blocchi si consumassero contemporaneamente. Il controllore cerca di evitare questo problema mediante vari algoritmi che favoriscono la memorizzazione dei dati sui blocchi meno cancellati, in modo che le successive cancellazioni avvengano su quei blocchi piuttosto che sui blocchi più cancellati, livellando così l'usura dell'intero dispositivo.

In termini di protezione dei dati, i dispositivi nvm, come i dischi rigidi, forniscono codici di correzione degli errori che vengono calcolati e archiviati insieme ai dati durante la scrittura e letti insieme ai dati per rilevare gli errori e, se possibile, correggerli (i codici di correzione degli errori sono discussi nel Paragrafo 11.5.1). Se una pagina presenta spesso errori correggibili, può essere contrassegnata come non valida (bad page) in modo da non essere utilizzata nelle scritture successive. Su un singolo dispositivo nvm, come su un disco rigido, possono verificarsi gravi guasti che causano la corruzione dei dati o l'impossibilità di rispondere alle richieste di lettura o scrittura. Per consentire ai dati di essere ripristinati in tali casi viene utilizzata la protezione raid.

11.1.3 Memoria volatile

Può sembrare strano discutere di memoria volatile in un capitolo che tratta la memoria di massa, ma è utile farlo, perché la dram viene spesso utilizzata come dispositivo di archiviazione di massa. Nello specifico, le unità ram (chiamate in diversi modi, tra cui dischi ram) agiscono come memoria secondaria, ma sono create da driver di periferica che ritagliano una sezione della dram di sistema e la presentano al resto del sistema come se fosse una memoria di massa. Queste "unità" possono essere utilizzate come

dispositivi a blocchi non formattati, ma più comunemente su di esse vengono creati file system per permettere operazioni standard su file.

Visto che i computer sono già dotati di buffer e cache, qual è lo scopo di utilizzare anche la dram per l'archiviazione temporanea dei dati? Dopotutto, la dram è volatile e i dati su un'unità ram non sopravvivono in seguito a un arresto anomalo del sistema, o a un arresto volontario o un'interruzione di corrente. Mentre cache e buffer sono allocati dal programmatore o dal sistema operativo, le unità ram consentono all'utente (così come al programmatore) di salvare temporaneamente i dati in memoria utilizzando le normali operazioni sui file. A riprova del fatto che la funzionalità dei dischi ram è utile, osserviamo che tali unità sono disponibili in tutti i principali sistemi operativi: su Linux c'è `/dev/ram`, su macos il comando `diskutil` permette di creare unità ram, su Windows le unità ram sono rese disponibili tramite strumenti di terze parti; inoltre, su Solaris e Linux viene creata all'avvio la directory `/tmp` di tipo "tmpfs", che è un'unità ram.

Le unità ram sono utili come spazio di archiviazione temporaneo ad alta velocità. Sebbene i dispositivi nvm siano veloci, la dram è molto più veloce e le operazioni di i/o su unità ram sono il modo più veloce per creare, leggere, scrivere ed eliminare i file e il loro contenuto. Molti programmi utilizzano (o potrebbero trarre vantaggio dall'utilizzo) unità ram per la memorizzazione di file temporanei. Per esempio, i programmi possono condividere facilmente i dati scrivendo e leggendo i file su un'unità ram. Come ulteriore esempio, Linux crea all'avvio un file system temporaneo (`initrd`) che consente ad altre parti del sistema di accedere a un file system di root e al suo contenuto prima che vengano caricate le parti del sistema operativo che comprendono i dispositivi di memorizzazione.

Figura 11.5 AGNETICI

I nastri magnetici sono stati i primi supporti di memorizzazione secondaria. Pur avendo la capacità di memorizzare in modo permanente un'enorme quantità di dati, queste unità sono caratterizzate da un tempo d'accesso molto elevato rispetto a quello della memoria centrale e dei dischi magnetici. Inoltre il tempo d'accesso casuale dei nastri magnetici (essendo fisicamente ad accesso sequenziale) è un migliaio di volte maggiore di quello dei dischi magnetici, e circa centomila volte più lento dell'accesso casuale a ssd, e ciò li rende inadatti come supporto di memoria secondaria. Gli usi principali dei nastri sono la creazione di copie di backup dei dati, la registrazione di dati poco usati e il trasferimento di informazioni tra diversi sistemi elaborativi.

Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere alcuni minuti, anche se, una volta raggiunta la posizione desiderata, l'unità a nastro può leggere o scrivere informazioni a una velocità paragonabile a quella di un'unità a disco. La capacità varia secondo il particolare tipo di unità a nastro. I nastri di oggi hanno una capacità di diversi terabyte. Alcune unità sono dotate di funzionalità di compressione dei dati che permettono di più che raddoppiare la capacità effettiva. Le unità a nastro e i loro driver sono solitamente classificate per larghezza: misure tipiche sono 4, 8 o 19 millimetri, e 1/4 o 1/2 pollice. Alcune unità prendono il nome dalla tecnologia su cui si basano, come nel caso di lto-6 (Figura 11.5) e sdlt.



Unità a nastro lto-6 con cartuccia inserita.

11.1.4 Metodi di connessione alla memoria secondaria

Un dispositivo di memoria secondaria è collegato a un computer tramite il bus di sistema o tramite un bus di i/o. Sono disponibili diversi tipi di bus, tra cui ata (*Advanced Technology Attachment*), sata (*Serial ata*), esata, sas (*Serial Attached scsi*), usb (*Universal Serial Bus*) e fc (*Fibre Channel*); tra questi, il più comune è sata. Poiché i dispositivi nvm sono molto più veloci degli hdd, è stata creata un'interfaccia speciale e veloce per tali dispositivi chiamata nvme (*nvm Express*), che collega direttamente il dispositivo al bus pci di sistema, aumentando il throughput e riducendo la latenza rispetto ad altri metodi di connessione.

I trasferimenti di dati su un bus vengono eseguiti da speciali processori chiamati controlleri o adattatori bus-host (hba). L'host controller si trova alla fine del bus del lato del computer, mentre un controllore di periferica è incorporato in ogni dispositivo di memoria di massa. Per eseguire un'operazione di i/o su un dispositivo di memoria di massa, il computer inserisce un comando nell'host controller, in genere utilizzando le porte di i/o mappate in memoria, come descritto nel Paragrafo 12.2.1. Il controllore invia quindi il comando tramite messaggi al controllore del dispositivo che gestisce l'hardware dell'unità per eseguire il comando. I controlleri dei dispositivi hanno soffitamente una cache integrata; il trasferimento di dati sull'unità avviene tra la cache e il supporto di memorizzazione, mentre il trasferimento di dati verso l'host avviene, a velocità elevate, con la cache dram dell'host, tramite dma.

11.1.5 Mappatura degli indirizzi

I dispositivi di archiviazione sono indirizzati come grandi vettori unidimensionali di blocchi logici, in cui il blocco logico è la più piccola unità di trasferimento. Ogni blocco logico viene mappato su un settore fisico o su una pagina di un dispositivo a semiconduttore e l'intero vettore di blocchi logici è mappato sui settori o sulle pagine del dispositivo. Per esempio, il settore 0 può essere il primo settore della prima traccia sul cilindro più esterno di un hdd. La mappatura procede in ordine su quella traccia, quindi attraverso il resto delle tracce sullo stesso cilindro, e infine attraverso il resto dei cilindri, dall'esterno all'interno. Nei dispositivi nvm una tupla (una lista ordinata e finita) *<chip, blocco, pagina>* viene mappata su un vettore di blocchi logici. Un indirizzo di blocco logico (lba) può essere utilizzato dagli algoritmi in maniera più semplice rispetto a una tupla *<settore, cilindro, testina>* o a una tupla *<chip, blocco, pagina>*.

Usando questa mappatura su un disco rigido, possiamo, almeno in teoria, convertire un numero di blocco logico in un indirizzo "vecchio stile" del disco formato da un numero di cilindro, un numero di traccia all'interno di quel cilindro e un numero di settore all'interno di quella traccia. Nella pratica è difficile eseguire questa traduzione, per tre motivi. Innanzitutto, la maggior parte delle unità presenta alcuni settori difettosi, che la mappatura nasconde sostituendoli con settori di riserva collocati in altre posizioni sull'unità: in questo caso l'indirizzo del blocco logico rimane sequenziale, ma la posizione del settore fisico cambia. In secondo luogo, su alcune unità il numero di settori per traccia non è costante. Infine, i produttori di dischi gestiscono la mappatura di lba su indirizzi fisici internamente, pertanto nelle unità che si utilizzano ora esiste una scarsa relazione tra lba e settori fisici. Nonostante queste peculiarità degli indirizzi fisici, gli algoritmi che gestiscono gli hdd tendono a presumere che gli indirizzi logici siano relativamente correlati agli indirizzi fisici, ovvero ad associare la crescita dell'indirizzo logico con una crescita dell'indirizzo fisico.

Diamo un'occhiata più da vicino al secondo motivo. Sui dispositivi con velocità lineare costante (clv), la densità dei bit per traccia è uniforme. Più una traccia si trova lontana dal centro del disco, maggiore è la sua lunghezza e più settori può contenere, mentre spostandosi dalle zone esterne a quelle interne, il numero di settori per traccia diminuisce. Le tracce nella zona più esterna contengono in genere il 40 percento di settori in più rispetto alle tracce nella zona più interna. L'unità aumenta la sua velocità di rotazione quando la testina si sposta dalle tracce esterne a quelle interne, in modo da mantenere invariata la quantità di dati in movimento sotto la testina. Questo metodo è utilizzato nelle unità cd-rom e dvd-rom. In alternativa, la velocità di rotazione del disco può rimanere costante. In questo caso, la densità dei bit diminuisce spostandosi dalle tracce interne alle tracce esterne in modo da mantenere costante la velocità di trasmissione dei dati (e fare in modo che le prestazioni siano simili indipendentemente dalla posizione dei dati sull'unità). Questo metodo è utilizzato nei dischi rigidi ed è noto come *velocità angolare costante* (cav).

Il numero di settori per traccia è aumentato al migliorare della tecnologia dei dischi: oggi la zona esterna di un disco contiene generalmente diverse centinaia di settori per traccia. Analogamente, il numero di cilindri per disco è aumentato e i dischi di grandi dimensioni hanno ora decine di migliaia di cilindri.

Si noti che esistono più tipi di dispositivi di archiviazione di quelli che è ragionevole coprire in un testo di sistemi operativi. Per esempio, ci sono dischi rigidi con registrazione magnetica a strati, che offrono densità maggiore, ma prestazioni peggiori rispetto agli hdd tradizionali (si veda <http://www.tomshardware.com/articles/shingled-magnetic-recoding-smr-101-basics-2-933.html>). Esistono anche dispositivi che combinano le tecnologie nvm e hdd, o gestori di volumi (si veda il Paragrafo 11.5) che consentono di collegare dispositivi nvm e hdd in un'unica unità di archiviazione più veloce dell'hdd, ma con un costo inferiore rispetto a un dispositivo nvm. Questi dispositivi hanno caratteristiche diverse da quelli più comuni e possono richiedere algoritmi di memorizzazione nella cache e di scheduling differenti per consentire di massimizzare le prestazioni.

11.2 Scheduling dei dischi rigidi

Una delle responsabilità del sistema operativo è quella di fare un uso efficiente dell'hardware. Nel caso dei dischi rigidi, far fronte a questa responsabilità significa garantire tempi d'accesso contenuti e ampiezze di banda elevate.

Nel caso dei dischi e di altri dispositivi meccanici il tempo d'accesso si può scindere in due componenti principali, come menzionato nel Paragrafo 11.1: il tempo di ricerca (*seek time*), cioè il tempo necessario affinché il braccio dell'unità a disco sposti le testine fino al cilindro contenente il settore desiderato, e la latenza di rotazione (*rotational latency*), e cioè il tempo aggiuntivo necessario perché il disco ruoti finché il settore desiderato si trovi sotto la testina. L'ampiezza di banda (*bandwidth*) del disco è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell'ultimo trasferimento. Gestendo l'ordine delle richieste di i/o relative al disco si possono migliorare sia il tempo d'accesso sia l'ampiezza di banda.

Ogni volta che deve compiere operazioni di i/o con un'unità a disco, un processo effettua una chiamata di sistema.

La richiesta contiene diverse informazioni:

- se l'operazione è di input o di output;
- l'indirizzo nel disco per il trasferimento;
- l'indirizzo di memoria per il trasferimento;
- il numero di settori da trasferire.

Se l'unità a disco desiderata e il controllore sono disponibili, la richiesta si può immediatamente soddisfare; altrimenti le nuove richieste si aggiungono alla coda di richieste inattese relativa a quell'unità. La coda relativa a un'unità a disco in un sistema con multiprogrammazione può spesso essere piuttosto lunga.

La presenza di una coda di richieste verso un dispositivo su cui è possibile minimizzare i tempi di ricerca (*seek*) consente ai driver del dispositivo di migliorare le prestazioni ordinando opportunamente la coda.

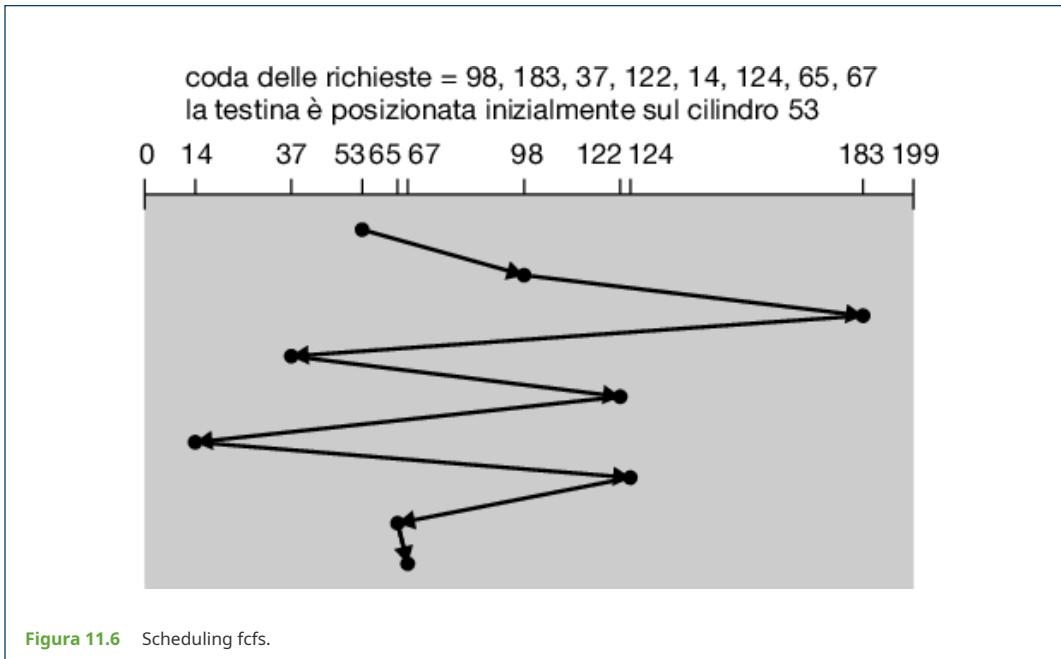
In passato, le interfacce delle unità disco richiedevano che l'host specificasse traccia e testina da utilizzare, e un notevole sforzo era rivolto agli algoritmi di scheduling del disco. Le unità prodotte a partire dagli inizi di questo secolo non solo non incaricano gli host di questi controlli, ma la mappatura di lba in indirizzi fisici è sotto il controllo dell'unità stessa. Gli obiettivi attuali dello scheduling del disco comprendono equità, tempestività e ottimizzazioni come, per esempio, il raggruppamento di letture o scritture che appaiono in sequenza, poiché le unità forniscono prestazioni migliori con i/o sequenziali. Alcuni sforzi nello scheduling sono pertanto ancora utili. Per ottenere i risultati desiderati è possibile utilizzare uno qualsiasi dei numerosi algoritmi di scheduling del disco che discuteremo in seguito. Si noti che la conoscenza assoluta della posizione della testina e delle posizioni fisiche del blocco/cilindro non è generalmente possibile sulle unità moderne, ma in maniera approssimativa gli algoritmi possono presupporre che l'aumento dell'lba implichi un incremento degli indirizzi fisici, e che lba vicini siano mappati in blocchi fisici vicini.

11.2.1 Scheduling in ordine d'arrivo – FCFS

La forma più semplice di scheduling è, naturalmente, l'algoritmo di servizio secondo l'ordine d'arrivo (*first come, first served, fcfs*). Si tratta di un algoritmo intrinsecamente equo, ma che in generale non garantisce la massima velocità del servizio. Si consideri, per esempio, una coda di richieste per l'unità a disco che riguardino blocchi sui seguenti cilindri (nell'ordine):

98, 183, 37, 122, 14, 124, 65, 67.

Se si trova inizialmente al cilindro 53, la testina dell'unità a disco dovrà prima spostarsi al cilindro 98, poi al 183, 37, 122, 14, 124, 65 e infine al 67, per un movimento totale della testina, misurato in numero di cilindri visitati, di 640 cilindri. La sequenza è rappresentata nella Figura 11.6.



Le defezioni di quest'algoritmo sono evidenziate dal grande salto da 122 a 14 e poi di nuovo a 124: se le richieste per i cilindri 37 e 14 si potessero soddisfare in sequenza, la distanza totale percorsa diminuirebbe notevolmente e le prestazioni migliorerebbero di conseguenza.

11.2.2 Scheduling – SCAN

Secondo l'algoritmo scan il braccio dell'unità a disco parte da un estremo del disco e si sposta verso l'altro estremo, servendo le richieste mentre attraversa i cilindri, finché non completa il tragitto: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni. L'algoritmo scan è a volte chiamato algoritmo dell'ascensore, perché il braccio dell'unità a disco si comporta proprio come un ascensore che serva prima tutte le richieste in salita e poi tutte quelle in discesa.

Si consideri ancora l'esempio precedente. Prima di poter applicare lo scheduling scan alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, e 67, oltre la posizione corrente (53), occorre conoscere la direzione del movimento delle testine. Se lo spostamento è nella direzione del cilindro 0, l'unità a disco servirà prima la richiesta 37 e poi la 14; una volta giunto al cilindro 0, il braccio invertirà il movimento verso l'altro estremo del disco, servendo le richieste 65, 67, 98, 122, 124 e 183 (Figura 11.7). Se arriva una nuova richiesta riferita a uno dei cilindri posti davanti alla testina essa sarà quasi immediatamente soddisfatta; ma se la richiesta è riferita a uno dei cilindri appena sorpassati, essa dovrà attendere fino a che la testina non giunga alla fine del disco, inverta la direzione del moto, e torni indietro.

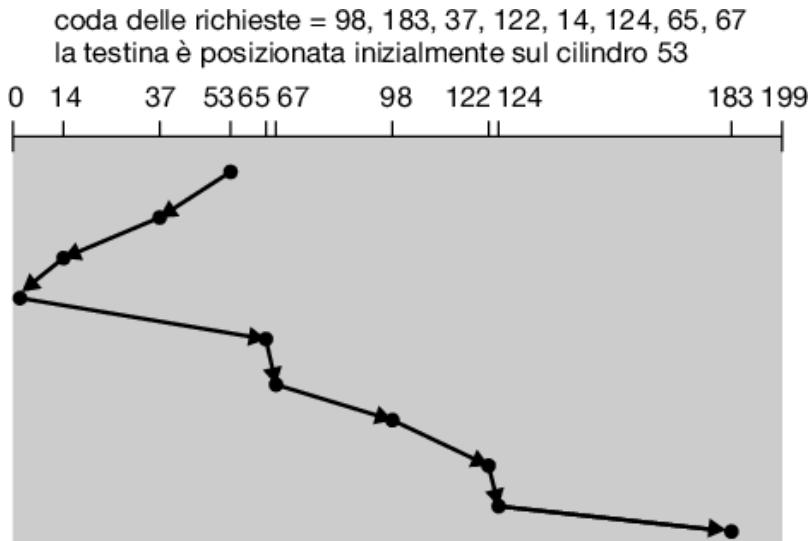


Figura 11.7 Scheduling scan.

Assumendo una distribuzione uniforme delle richieste per i cilindri da visitare, si consideri la densità di richieste quando il braccio giunge a un estremo e inverte la direzione del moto: in quel momento, relativamente poche richieste sono riferite a cilindri posti appena davanti alla testina, perché i cilindri in questione sono stati recentemente visitati. La massima densità di richieste si riferisce all'altro estremo del disco, e queste richieste sono anche quelle che hanno atteso più a lungo: sembra ragionevole spostarsi subito lì. Questa è l'idea dell'algoritmo seguente.

11.2.3 Scheduling – C-SCAN

L'algoritmo scan circolare (*circular scan*, c-scan) è una variante dello scheduling scan concepita per garantire un tempo d'attesa meno variabile. Anche l'algoritmo c-scan, come lo scan, sposta la testina da un estremo all'altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all'altro estremo del disco, ritorna immediatamente all'inizio del disco stesso, senza servire richieste durante il viaggio di ritorno.

Si consideri di nuovo l'esempio precedente. Prima di poter applicare lo scheduling c-scan alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, e 67, occorre conoscere la direzione di movimento delle testine secondo cui le richieste sono pianificate. Supponendo che le richieste siano schedulate quando il braccio del disco si muove da 0 a 199 e che la posizione iniziale della testina sia nuovamente 53, la richiesta verrà servita come illustrato nella Figura 11.8. L'algoritmo di scheduling c-scan, essenzialmente, tratta il disco come una lista circolare, cioè come se il primo e l'ultimo cilindro fossero adiacenti.

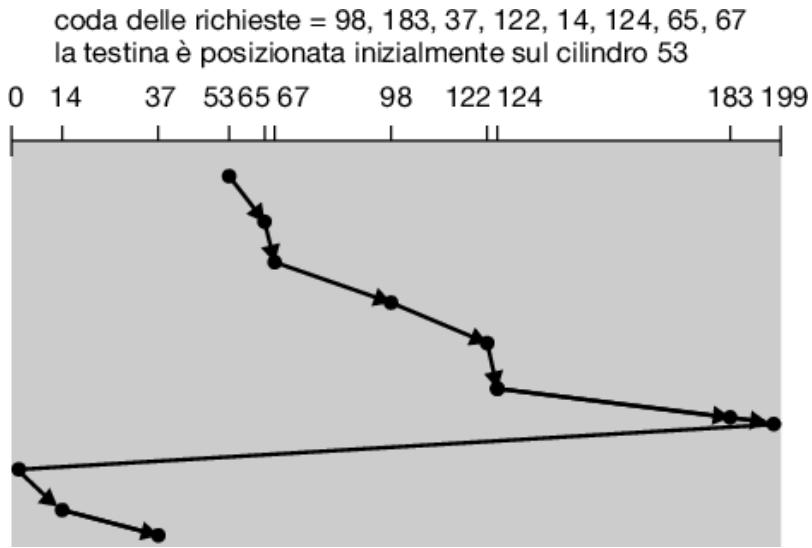


Figura 11.8 Scheduling c-scan.

11.2.4 Scelta di un algoritmo di scheduling

Molti algoritmi di scheduling del disco non sono stati inclusi in questo capitolo perché sono usati raramente. In che modo i progettisti dei sistemi operativi decidono quale algoritmo implementare e i responsabili di una installazione scelgono il migliore da utilizzare? Per una data lista di richieste si può definire un ordine ottimo di servizio, ma la computazione richiesta può non essere giustificata dal miglioramento in prestazioni rispetto all'algoritmo scan.

Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste. Per esempio, si supponga che la coda sia costituita in genere di una sola richiesta inesposta: tutti gli algoritmi danno allora luogo allo stesso comportamento, perché hanno una sola scelta possibile di spostamento della testina. In questo caso, tutti gli algoritmi si comportano come l'fcfs.

scan e c-scan forniscono prestazioni migliori in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità a situazioni d'attesa indefinita. Tuttavia, le possibilità di attesa indefinita permangono, e per questa ragione in Linux è stato implementato lo scheduler con scadenza (*deadline scheduler*). Questo algoritmo di scheduling conserva code di lettura e scrittura separate e dà priorità alla lettura, poiché i processi hanno più probabilità di bloccarsi in lettura che in scrittura. Le code sono ordinate secondo gli lba, implementando essenzialmente c-scan, e tutte le richieste di i/o vengono inviate secondo questo ordine. Lo scheduler con scadenza mantiene quattro code: due in lettura e due in scrittura, una in ordine di lba e l'altra in ordine fcfs. Dopo ogni lotto di operazioni l'algoritmo controlla se ci sono richieste nelle code fcfs più vecchie di un tempo preconfigurato (per impostazione predefinita 500ms). In tal caso, viene selezionata la coda lba (di lettura o di scrittura) contenente tale richiesta per il successivo lotto di i/o.

Lo scheduler di i/o con scadenza è lo scheduler predefinito nella distribuzione Linux RedHat 7, ma rhel 7 ne include altri due: noop, che viene preferito su sistemi cpu-bound che utilizzano dispositivi di i/o veloci, per esempio nvm, e lo scheduler cfq (*Completely Fair Queueing*), predefinito per le unità sata. cfq mantiene tre code (ordinate secondo gli lba mediante *insertion sort*): la coda real time, la coda best effort (default) e la coda idle. In questo ordine, ogni coda ha una priorità esclusiva sulle altre, con possibilità di attesa indefinita. Lo scheduler cfq utilizza i dati storici, cercando di prevedere se un processo genererà presto ulteriori richieste di i/o. In tal caso, resterà in attesa del nuovo i/o, ignorando altre richieste in coda, al fine di minimizzare il tempo di ricerca, assumendo la località dei riferimento delle richieste di i/o effettuate dallo stesso processo. Ulteriori dettagli su questi scheduler sono disponibili all'indirizzo https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Performance_Tuning_Guide/index.html.

11.3 Scheduling di dispositivi NVM

Gli algoritmi di scheduling discussi in precedenza si applicano a memorie basate su dischi meccanici e hanno come obiettivo principale la riduzione della quantità di movimento della testina del disco. I dispositivi nvm, che non contengono testine in movimento, usano solitamente una semplice politica fcfs. Per esempio, lo scheduler Noop di Linux utilizza una politica fcfs modificata per fondere richieste adiacenti. Il comportamento osservato dei dispositivi nvm indica che il tempo richiesto per le letture è uniforme, ma che, a causa delle proprietà della memoria flash, il tempo richiesto per le scritture non lo è. Alcuni scheduler per gli ssd sfruttano questa proprietà e fondono soltanto le richieste di scrittura adiacenti, mentre le richieste di lettura sono servite in ordine fcfs.

Come abbiamo visto, l'i/o può essere sequenziale o casuale. L'accesso sequenziale è ottimale per dispositivi meccanici come gli hdd e i nastri, poiché i dati da leggere o scrivere si trovano vicino alla testina di lettura/scrittura. L'i/o ad accesso casuale, misurato in operazioni di input/output al secondo (iops), provoca il movimento della testina del disco, e quindi è molto più veloce sui dispositivi nvm: un hdd può produrre centinaia di iops, mentre un ssd può arrivare a centinaia di migliaia di iops.

Il vantaggio dei dispositivi nvm è molto inferiore in caso di trasferimenti sequenziali, dove la ricerca su dischi rigidi è ridotta al minimo e vengono enfatizzate la lettura e la scrittura dei dati. In questi casi, per le letture, le prestazioni dei due tipi di dispositivi possono essere equivalenti o differire di un ordine di grandezza in favore dei dispositivi nvm. La scrittura su nvm è più lenta della lettura e ciò riduce ulteriormente il vantaggio rispetto agli hdd. Inoltre, mentre le prestazioni di scrittura su hdd restano costanti durante il ciclo di vita del dispositivo, le prestazioni in scrittura su dispositivi nvm variano a seconda di quanto il dispositivo è pieno (si ricordi la necessità di garbage collection e over-provisioning) e di quanto è usurato. Un dispositivo nvm vicino al termine del suo ciclo di vita ha generalmente prestazioni molto inferiori rispetto a un nuovo dispositivo, a causa dei molti cicli di cancellazione effettuati.

Un modo per migliorare la durata e le prestazioni dei dispositivi nvm nel tempo è che il file system informi il dispositivo quando i file vengono eliminati, in modo che il dispositivo possa cancellare i blocchi su cui sono stati memorizzati i file. Questo approccio è discusso in maniera più approfondita nel Paragrafo 14.5.6.

Diamo uno sguardo più ravvicinato all'impatto della garbage collection sulle prestazioni. Considerate un dispositivo nvm soggetto a letture e scritture casuali. Supponiamo che tutti i blocchi siano stati scritti, ma ci sia spazio disponibile. In questo caso è necessaria una garbage collection per recuperare lo spazio occupato da dati non validi, e una scrittura può quindi causare la lettura di una o più pagine, la scrittura dei dati validi presenti in quelle pagine sullo spazio di over-provisioning, la cancellazione del blocco contenente solo dati non validi e il posizionamento di quel blocco nello spazio di over-provisioning. In sintesi, una richiesta di scrittura causa una scrittura di pagina (per i dati), una o più letture di pagina (durante la garbage collection) e una o più scritture di pagina (di dati validi dai blocchi raccolti dalla garbage collection). La creazione di richieste di i/o non provenienti dalle applicazioni, ma dal dispositivo nvm che esegue la garbage collection e la gestione dello spazio di archiviazione è detta amplificazione di scrittura e può avere un notevole impatto sulle prestazioni in scrittura del dispositivo. Nel peggior dei casi, vengono attivate diverse operazioni di i/o aggiuntive per ogni richiesta di scrittura.

11.4 Rilevamento e correzione di errori

Il rilevamento e la correzione degli errori sono fondamentali in molti contesti, tra cui la gestione della memoria, delle reti e dei dispositivi di memoria di massa. Il rilevamento degli errori permette di determinare se si è verificato un problema, per esempio se un bit in una dram ha subito una modifica spontanea da 0 a 1, se il contenuto di un pacchetto di rete è cambiato durante la trasmissione, o se un blocco di dati ha subito una modifica tra quando è stato scritto e quando è stato letto. Rilevando il problema, il sistema può arrestare un'operazione prima che l'errore venga propagato, segnalare l'errore all'utente o all'amministratore, o notificare il malfunzionamento di un dispositivo.

Per molto tempo i sistemi di memoria hanno rilevato gli errori utilizzando i bit di parità. In questo scenario, a ogni byte in un sistema di memoria è associato un bit di parità che registra se il numero di bit nel byte impostato a 1 è pari (parità = 0) o dispari (parità = 1). Se uno dei bit nel byte è corrotto (un 1 diventa 0, oppure uno 0 diventa 1), la parità del byte cambia e non corrisponde più alla parità memorizzata. Allo stesso modo, se il bit di parità memorizzato è corrotto, non corrisponde alla parità calcolata. Ciò permette al sistema di memoria di rilevare tutti gli errori su un singolo bit, ma un errore su due bit potrebbe non essere rilevato. Si noti che la parità viene calcolata facilmente eseguendo un xor (o esclusivo) dei bit. Si noti inoltre che per ogni byte di memoria abbiamo bisogno di un bit aggiuntivo per memorizzare la parità.

La parità è una forma di checksum, che utilizza l'aritmetica modulare per calcolare, archiviare e confrontare valori su parole a lunghezza fissa. Un altro metodo di rilevamento degli errori, comune nelle reti, è il controllo di ridondanza ciclica (crc), che utilizza una funzione di hash per rilevare errori su più bit (si veda <http://www.mathpages.com/home/kmath458/kmath458.htm>).

Un codice di correzione degli errori (ecc) è in grado di correggere il problema, oltre a rilevarlo. La correzione viene eseguita utilizzando opportuni algoritmi e una certa quantità di spazio di memorizzazione aggiuntivo. I codici variano in base alla quantità di memoria aggiuntiva di cui hanno bisogno e al numero di errori che possono correggere. Le unità disco utilizzano codici di correzione per settore, mentre le unità flash utilizzano codici di correzione per pagina. Quando il controllore scrive un settore (una pagina) di dati durante il normale i/o, viene scritto anche un valore, l'ecc, calcolato a partire da tutti i byte dei dati che vengono scritti. Quando il settore (pagina) viene letto, l'ecc viene ricalcolato e confrontato con il valore memorizzato: un valore memorizzato diverso da quello calcolato indica che i dati sono corrotti e che il supporto di memorizzazione potrebbe essere danneggiato (si veda il Paragrafo 11.5.3). L'ecc permette di correggere errori quando solo pochi bit sono corrotti, perché contiene informazioni sufficienti per consentire al controllore di identificare quali bit sono stati modificati e calcolare quali sono i loro valori corretti. In tal caso viene segnalato un errore reversibile (*soft*). Se si verificano troppe modifiche e l'ecc non può correggere l'errore, viene segnalato un errore irreversibile (*hard*). Il controllore esegue automaticamente l'elaborazione necessaria ogni volta che un settore o una pagina viene letta o scritta.

Il rilevamento e la correzione degli errori costituiscono spesso fattori che distinguono i prodotti di consumo dai prodotti aziendali. L'ecc viene utilizzato, per esempio, su alcuni sistemi per la correzione degli errori nelle dram e per la protezione del data path.

11.5 Gestione delle unità di memoria secondaria

Il sistema operativo è anche responsabile di molti altri aspetti della gestione delle unità a disco. In questo paragrafo si discutono l'inizializzazione del dispositivo, l'avviamento del sistema da dispositivo, e la gestione dei blocchi difettosi.

11.5.1 Formattazione del disco, partizioni e volumi

Un dispositivo di storage nuovo è *tabula rasa*: è semplicemente un piatto ricoperto di materiale magnetico o un insieme di celle di memorizzazione a semiconduttore non inizializzate; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Nel caso dei dispositivi nvm, devono essere inizializzate le pagine e creata la ftl. Questo processo si chiama formattazione di basso livello, o formattazione fisica. La formattazione di basso livello riempie il dispositivo con una speciale struttura dati per ogni locazione di memorizzazione. La struttura dati per un settore o una pagina tipicamente consiste di un'intestazione, un'area per i dati e una coda. L'intestazione e la coda contengono informazioni usate dal controllore del disco, per esempio il numero del settore e un codice per la correzione degli errori (*error-correcting code*, ecc).

La formattazione fisica dei dispositivi è eseguita nella maggior parte dei casi dal costruttore come parte del processo produttivo; ciò permette al costruttore di provare il dispositivo, e di inizializzare la corrispondenza fra blocchi logici e settori o pagine correttamente funzionanti. La scelta della dimensione del settore è di solito ristretta a poche opzioni, come 512 byte o 4 kb. La formattazione in settori più grandi implica la presenza di meno settori su ogni traccia, ma anche meno intestazioni e code, e quindi maggior spazio per i dati utente. Alcuni sistemi operativi gestiscono solo una specifica dimensione dei settori.

Prima di usare un dispositivo come contenitore di file, il sistema operativo deve ancora registrare le proprie strutture dati nel disco, cosa che fa in tre passi.

Il primo passo è quello di suddividere il dispositivo in uno o più gruppi di blocchi o pagine, dette partizioni. Il sistema operativo può trattare ciascuna partizione come se fosse un'unità a sé stante: per esempio, una partizione può contenere un file system con una copia del codice eseguibile del sistema operativo, un'altra lo spazio di swap e un'altra ancora un file system contenente i file degli utenti. Alcuni sistemi operativi e file system eseguono automaticamente il partizionamento nel caso in cui un intero dispositivo debba essere gestito dal file system. Le informazioni sulle partizioni sono scritte in un formato stabilito e in una posizione fissa sul dispositivo di archiviazione. Il comando `fdisk` di Linux permette di gestire le partizioni sui dispositivi di archiviazione; una volta che il sistema operativo riconosce il dispositivo, vengono lette le informazioni sulle partizioni e il sistema operativo crea delle descrizioni per le partizioni stesse (in Linux, in `/dev`). Un file di configurazione, per esempio `/etc/fstab`, indica quindi al sistema operativo la posizione in cui montare ogni partizione contenente un file system e quali vincoli aggiuntivi adottare, per esempio la sola lettura. Quando un file system viene montato, diventa disponibile per l'uso da parte del sistema e dei suoi utenti.

Il secondo passo è la creazione e la gestione del volume. A volte questo passaggio è implicito, come quando un file system viene inserito direttamente all'interno di una partizione. Quel volume è quindi pronto per essere montato e usato. Altre volte, la creazione e la gestione del volume è esplicita, per esempio quando più partizioni o dispositivi verranno utilizzati insieme in raid (si veda il Paragrafo 11.8) con uno o più file system distribuiti sui dispositivi. Il volume manager `lvm2` di Linux può fornire queste funzionalità, così come strumenti commerciali di terze parti per Linux e altri sistemi operativi. zfs offre sia la gestione del volume sia un file system, integrati in un unico set di comandi e funzionalità. Si noti che il termine "volume" possa indicare qualsiasi file system montabile, incluso un singolo file contenente un file system, come l'immagine di un cd.

Il terzo passo è la formattazione logica, cioè la creazione di un file system: il sistema operativo registra nel dispositivo le strutture dati iniziali relative al file system. Le strutture dati in questione possono includere descrizioni dello spazio libero e dello spazio allocato e una directory iniziale vuota.

Le informazioni sulle partizioni indicano anche se una partizione contiene un file system avviabile (contenente il sistema operativo). La partizione etichettata per l'avvio viene utilizzata per stabilire la radice (root) del file system; una volta montata questa partizione è possibile creare collegamenti per tutti gli altri dispositivi e le relative partizioni. In generale, il "file system" di un computer è costituito da tutti i volumi montati. Su Windows, questi vengono denominati separatamente tramite una lettera (C:, D:, E:), mentre su altri sistemi, come Linux, al momento dell'avvio viene montato il file system principale e gli altri file system possono essere montati all'interno della stessa struttura ad albero (come discusso nel Paragrafo 13.3). In Windows l'interfaccia del file system rende chiaro quando viene utilizzato un determinato dispositivo, mentre in Linux un singolo accesso a un file può passare per l'attraversamento di molti dispositivi prima di arrivare al file system che lo contiene (all'interno di un volume). La Figura 11.9 mostra lo strumento `Gestione disco` di Windows 7 che visualizza tre volumi (C:, E: e F:). Si noti che E: ed F: sono ciascuno in una partizione del dispositivo "Disco 1" e che vi è spazio non allocato su quel dispositivo disponibile per ulteriori partizioni (eventualmente contenenti dei file system).

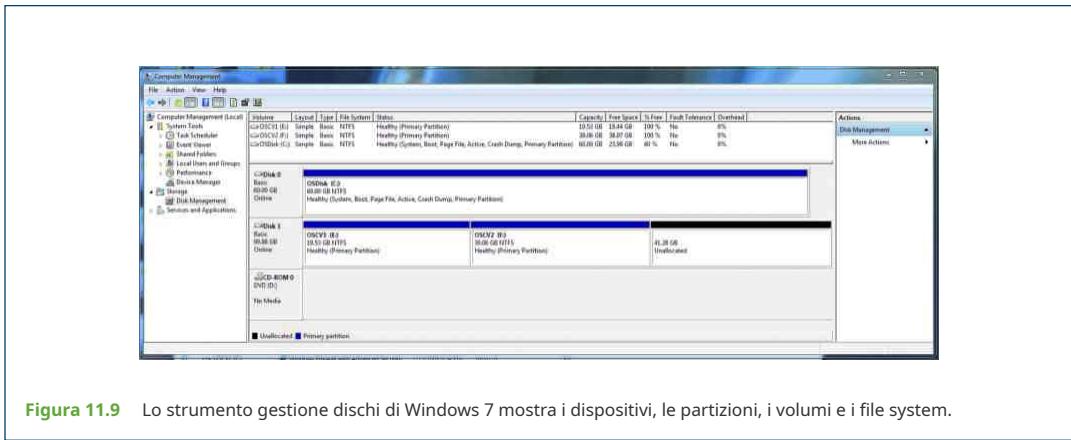


Figura 11.9 Lo strumento gestione dischi di Windows 7 mostra i dispositivi, le partizioni, i volumi e i file system.

Per una maggior efficienza, la maggior parte dei file system accorda i blocchi in gruppi, detti cluster. L'i/o del dispositivo procede per blocchi, ma l'i/o del file system procede invece per cluster, di modo che l'i/o abbia caratteristiche di accesso più sequenziale che casuale. I file system cercano anche di collocare il contenuto del file vicino ai suoi metadati, riducendo così i tempi di ricerca in un hdd quando si opera su un file.

Alcuni sistemi operativi danno l'opportunità a certi programmi speciali di impiegare una partizione del disco come un grande array sequenziale di blocchi logici, non contenente alcuna struttura relativa al file system. Questo array è detto a volte disco di basso livello (*raw disk*); l'i/o relativo si chiama i/o di basso livello (*raw i/o*). Un sistema di questo tipo può essere utilizzato come spazio di swap (si veda il Paragrafo 11.6.2), per esempio; inoltre, alcuni sistemi per la gestione di basi di dati preferiscono questo tipo di i/o perché permette di controllare l'esatta posizione nel disco di ogni record. Il raw i/o bypassa tutti i servizi del file system: gestione delle *buffer cache*, locking dei file, prefetching, l'allocazione dello spazio, la gestione dei nomi dei file e le directory. È possibile rendere certe applicazioni più efficienti permettendogli di implementare servizi di memorizzazione specializzati che usino una partizione di basso livello, ma la maggior parte delle applicazioni preferisce un file system già fornito piuttosto che prendersi carico della gestione dei dati. Si noti che Linux in generale non supporta l'i/o di basso livello, tuttavia è possibile ottenere un accesso di questo tipo utilizzando il flag *direct* nella chiamata di sistema *open()*.

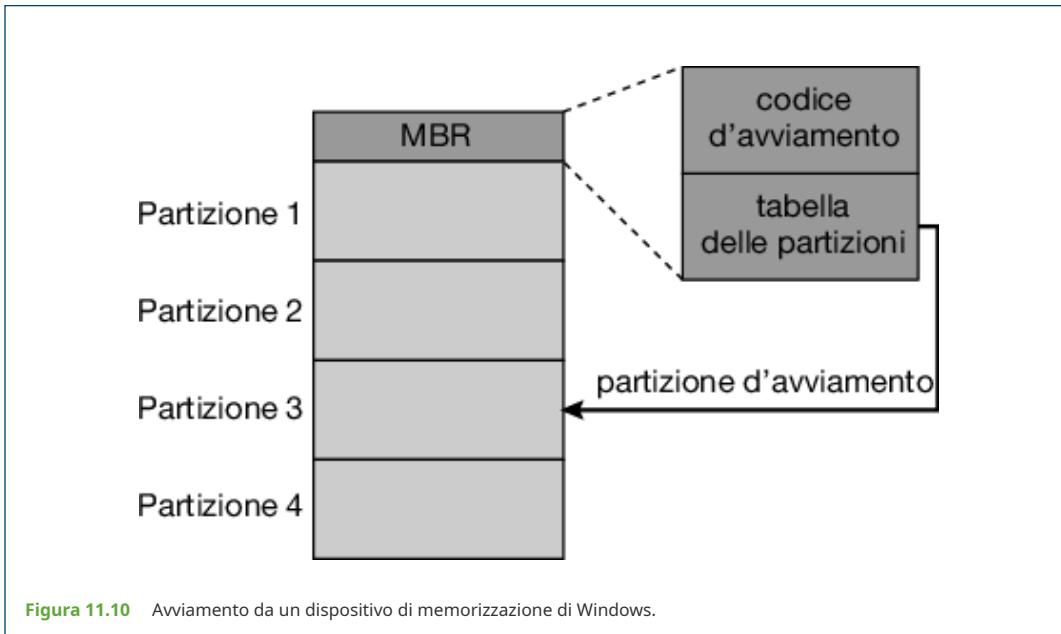
11.5.2 Blocco d'avviamento

Affinché un calcolatore possa entrare in funzione, per esempio quando viene acceso o riavviato, è necessario che esegua un programma iniziale; di solito, questo programma d'avviamento iniziale (*bootstrap*) è piuttosto semplice. Nella maggior parte dei calcolatori il bootstrap è memorizzato nel firmware (una memoria flash nvm) sulla scheda madre del sistema e mappato su una posizione di memoria nota. Il bootstrap può essere aggiornato dall'azienda produttrice secondo necessità, ma può anche essere sovrascritto da virus, infettando il sistema. Questo programma inizializza tutti gli aspetti del sistema, dai registri della cpu ai controllori dei dispositivi, al contenuto della memoria principale.

Il piccolo programma che carica il bootstrap (*bootstrap loader*) è anche in grado di caricare dalla memoria secondaria un programma di bootstrap completo, che è memorizzato nei blocchi di avvio in una posizione fissa del dispositivo. Il bootstrap loader predefinito di Linux è grub2 (<https://www.gnu.org/software/grub/manual/grub.html>). Un dispositivo contenente una partizione di avvio è chiamato disco di avvio o disco di sistema.

Il codice contenuto nella nvm d'avviamento istruisce il controllore dell'unità a disco affinché trasferisca il contenuto dei blocchi d'avviamento nella memoria (si noti che a questo fine non si carica alcun driver di dispositivo), quindi comincia a eseguire il codice. Il programma d'avviamento completo è più complesso del suo caricatore, ed è capace di trasferire nella memoria l'intero sistema operativo inizialmente residente in un dispositivo, in una locazione non fissata a priori, e di avviare il sistema operativo stesso.

Consideriamo come esempio il processo d'avviamento in Windows. Osserviamo innanzitutto che Windows consente di suddividere il disco rigido in una o più partizioni; in una di esse, detta partizione d'avviamento, sono contenuti il sistema operativo e i driver dei dispositivi. Windows colloca il proprio codice d'avviamento nel primo blocco logico del disco fisso o nella prima pagina del dispositivo nvm, denominato mbr (*master boot record*). La procedura d'avviamento inizia con l'esecuzione del codice residente nel firmware del sistema. Questo codice fa sì che il sistema legga il codice d'avviamento dall'mbr, conoscendo quanto basta sul controllore del dispositivo di archiviazione e sul dispositivo stesso per poter caricare un settore. Oltre al codice d'avviamento, l'mbr contiene una tabella che elenca le partizioni del dispositivo e un flag che indica da quale partizione si debba avviare il sistema, com'è illustrato nella Figura 11.10. Dopo aver identificato la partizione d'avviamento, il sistema legge da tale partizione il primo settore (chiamato settore d'avviamento) e svolge le restanti procedure d'avviamento, tra cui il caricamento dei vari sottosistemi e dei servizi del sistema.



11.5.3 Blocchi difettosi

Le unità a disco sono strutturalmente portate ai malfunzionamenti perché sono costituite da parti mobili con basse tolleranze (si ricordi che una testina è sospesa appena sopra la superficie del disco). A volte si può verificare un guasto irreparabile, e l'unità a disco deve essere sostituita: il suo contenuto dovrà essere recuperato da una copia di backup e trasferito nella nuova unità a disco. Più di frequente, uno o più settori divengono malfunzionanti; in effetti, la maggior parte dei dischi messi in commercio contiene già blocchi difettosi (*bad blocks*). Essi sono trattati in diversi modi a seconda del controllore e del tipo di disco.

Nel caso di dischi semplici come quelli gestiti da un controllore IDE, i blocchi difettosi sono gestiti "manualmente". Una possibile strategia consiste nell'effettuare la scansione del disco durante la formattazione, per rilevare la presenza di blocchi difettosi. Se si trova un blocco difettoso lo si marca come inutilizzabile al fine di segnalare alle procedure di allocazione del file system di non usarlo. Se qualche blocco diviene malfunzionante nel corso dell'ordinario uso del sistema, un programma speciale (per esempio il comando Linux `badblocks`) deve essere lanciato manualmente per individuare i blocchi difettosi e isolargli. Di solito, i dati residenti nei blocchi difettosi vanno perduti.

Unità a disco più complesse hanno strategie di recupero dei blocchi difettosi più raffinate. Il controllore mantiene una lista dei blocchi malfunzionanti dell'unità a disco che è inizializzata durante la formattazione fisica eseguita dal produttore, ed è aggiornata per tutto il periodo in cui l'unità a disco è operativa. La formattazione fisica mette anche da parte dei settori di riserva non visibili al sistema operativo: si può istruire il controllore affinché sostituisca da un punto di vista logico un settore difettoso con uno dei settori di riserva inutilizzati. Questa strategia è nota come accantonamento di settori (*sector sparing* o *sector forwarding*).

Un tipico esempio di attuazione di questa strategia è il seguente:

- il sistema operativo tenta di leggere il blocco logico 87;
- il controllore calcola l'ECC e scopre che il settore è difettoso; quindi segnala questo malfunzionamento al sistema operativo;
- la volta successiva che il sistema viene riavviato, si esegue un comando speciale al fine di comunicare al controllore la necessità di sostituire il settore difettoso con uno di riserva;
- dopo di ciò, ogni volta che il sistema tenta di leggere il contenuto del blocco 87, il controllore traduce la richiesta nell'indirizzo del settore di rimpiazzo.

Si noti che un tale reindirizzamento da parte del controllore potrebbe inficiare ogni ottimizzazione fornita dall'algoritmo di scheduling del disco del sistema operativo. Per questa ragione la maggior parte dei dischi viene formattata in modo tale da mantenere alcuni settori di riserva in ogni cilindro, e anche un intero cilindro di riserva. Quando un blocco difettoso viene rimappato, il controllore usa settori di riserva presenti nello stesso cilindro ogniqualvolta è possibile.

Un'alternativa all'accantonamento dei settori è data da quei controlleri capaci di sostituire i settori difettosi tramite la traslazione dei settori (*sector slipping*). Si supponga per esempio che il blocco logico 17 divenga malfunzionante, e che il primo settore di riserva disponibile sia quello successivo al settore 202. La traslazione dei settori sposta in avanti di una posizione tutti i settori dal 17 al 202: quindi, il settore 202 viene copiato sul settore di riserva, il settore 201 sul 202, il 200 sul 201, e così via, fino a che il settore 18 non viene copiato sul 19. Questa traslazione dei settori libera lo spazio del settore 18, e il settore 17 può essere mappato su quest'ultimo.

Gli errori reversibili possono attivare un processo in cui viene effettuata una copia dei dati del blocco e il blocco viene messo in riserva o traslato. Un *errore irreversibile*, tuttavia, causa una perdita di dati. Un file che usava quel blocco deve quindi essere riparato (per esempio, ripristinando da un nastro di backup), e ciò richiede un intervento manuale.

Anche i dispositivi nvm possono contenere bit, byte e persino pagine che sono difettose già al momento della produzione o che si degradano nel tempo. La gestione di queste aree difettose è più semplice rispetto agli hdd, perché non incide sul tempo di ricerca. È possibile in questi casi mettere da parte più pagine da utilizzare in sostituzione di quelle difettose, oppure utilizzare spazio dell'area di over-provisioning (diminuendo in questo caso la capacità di quest'area). In entrambi i casi, il controllore mantiene una tabella delle pagine non valide e non imposta mai quelle pagine come disponibili per la scrittura, in modo che non siano mai accessibili.

11.6 Gestione dell'area d'avvicendamento

L'avvicendamento (*swapping*) è stato introdotto, inizialmente, nel Paragrafo 9.5, dove abbiamo trattato lo spostamento di interi processi tra disco e memoria centrale. In quel contesto, l'avvicendamento interviene quando l'ammontare della memoria fisica si abbassa fino al punto di raggiungere la soglia critica e i processi vengono trasferiti dalla memoria all'area d'avvicendamento, per liberare memoria. Nella pratica, pochissimi sistemi operativi moderni realizzano l'avvicendamento nel modo descritto: essi, infatti, combinano l'avvicendamento con tecniche di memoria virtuale (Capitolo 10) ed effettuano lo swapping di pagine, e non necessariamente interi processi. Infatti alcuni sistemi considerano *avvicendamento* e *paginazione* termini intercambiabili, a riprova della moderna tendenza a convergere di questi due concetti.

La gestione dell'area d'avvicendamento è un altro compito di basso livello del sistema operativo. La memoria virtuale usa lo spazio dei dischi come estensione della memoria centrale: poiché l'accesso alle unità a disco è molto più lento dell'accesso alla memoria centrale, l'uso dell'area d'avvicendamento riduce notevolmente le prestazioni del sistema. L'obiettivo principale nella progettazione e realizzazione di un'area di avvicendamento è di fornire il migliore throughput per il sistema di memoria virtuale. In questo paragrafo sono trattati l'uso, la collocazione nei dischi e la gestione dell'area d'avvicendamento.

11.6.1 Uso dell'area d'avvicendamento

L'area d'avvicendamento (*area di swapping*) è usata in modi diversi da sistemi operativi diversi, in funzione degli algoritmi di gestione della memoria utilizzati. I sistemi che adottano l'avvicendamento dei processi nella memoria, per esempio, possono usare l'area d'avvicendamento per mantenere l'intera immagine del processo, inclusi i segmenti dei dati e del codice; i sistemi a paginazione, invece, possono semplicemente memorizzarvi pagine non contenute nella memoria centrale. Lo spazio richiesto dall'area d'avvicendamento per un sistema può quindi variare da pochi megabyte a alcuni gigabyte, a seconda della quantità di memoria fisica, della quantità di memoria virtuale che esso deve sostenere, e del modo in cui quest'ultima è usata.

Si noti che una stima per eccesso delle dimensioni dell'area d'avvicendamento è più prudente di una per difetto, perché un sistema che esaurisce l'area d'avvicendamento potrebbe essere costretto a terminare forzatamente i processi o ad arrestarsi completamente: una stima per eccesso spreca spazio dei dischi che si potrebbe usare per i file, ma non provoca altri danni. Alcuni sistemi consigliano la quantità da riservare per l'area d'avvicendamento. Solaris, per esempio, raccomanda di riservare a tal fine uno spazio uguale alla quantità di memoria virtuale che eccede la memoria fisica paginabile. Linux ha in passato suggerito di raddoppiare l'area d'avvicendamento rispetto alla memoria fisica, ma oggi questa limitazione è scomparsa e la maggior parte dei sistemi Linux usa un'area d'avvicendamento notevolmente minore.

Alcuni sistemi operativi, fra i quali Linux, permettono l'uso di aree d'avvicendamento multiple, che includono sia file che partizioni dedicate. Queste aree d'avvicendamento sono poste di solito in unità a disco distinte per distribuire su più dispositivi il carico della paginazione e dell'avvicendamento dei processi gravante sul sistema per l'i/o.

11.6.2 Collocazione dell'area d'avvicendamento

Le possibili collocazioni per un'area d'avvicendamento sono due: all'interno del normale file system, o in una partizione del disco a sé stante. Se l'area d'avvicendamento è semplicemente un grande file all'interno del file system, si possono usare le ordinarie funzioni del file system per crearla, assegnargli un nome, e allocare spazio per essa.

In alternativa, l'area d'avvicendamento si può creare in un'apposita partizione del disco non formattata (*raw partition*): in essa non è presente alcuna struttura relativa al file system e alle directory, ma si usa uno speciale gestore dell'area d'avvicendamento per allocare e rimuovere i blocchi. Esso adotta algoritmi ottimizzati rispetto alla velocità di accesso, e non rispetto allo spazio impiegato su disco, dato che all'area d'avvicendamento (quando viene utilizzata) si accede molto più frequentemente che al file system. La frammentazione interna può aumentare, ma questo prezzo da pagare è ragionevole perché i dati nell'area d'avvicendamento hanno una vita media molto più breve dei file ordinari. Poiché l'area d'avvicendamento è reinizializzata all'avvio del sistema, la frammentazione ha vita breve. Il metodo della raw partition assegna una dimensione fissa all'area d'avvicendamento al momento della creazione delle partizioni del disco, e l'aumento delle dimensioni dell'area d'avvicendamento deve quindi passare attraverso il ripartizionamento del disco (che implica lo spostamento o l'eliminazione e la sostituzione con copie di riserva delle altre partizioni del disco), oppure attraverso la creazione di un'altra area d'avvicendamento in qualche altra unità a disco del sistema.

Alcuni sistemi operativi non adottano una strategia rigida e possono costruire aree d'avvicendamento sia su partizioni specifiche sia all'interno del file system: Linux ne è un esempio. I metodi di gestione e la loro implementazione sono diversi nei due casi, e la scelta fra le due soluzioni è lasciata all'amministratore del sistema: sul piatto della bilancia pesano da un lato la facilità dell'allocazione e della gestione dell'area d'avvicendamento all'interno del file system, e dall'altro le migliori prestazioni ottenibili grazie all'uso di una partizione non formattata.

11.6.3 Gestione dell'area d'avvicendamento: un esempio

Si può comprendere come l'area d'avvicendamento venga utilizzata ripercorrendo l'evoluzione dello swapping e della paginazione nei vari sistemi unix. Il kernel tradizionale di unix implementava inizialmente una tecnica d'avvicendamento che spostava interi processi tra regioni contigue del disco e la memoria. Più tardi, con la disponibilità dell'hardware per la paginazione, unix progredi verso una combinazione d'avvicendamento e paginazione.

Per migliorare l'efficienza e sfruttare le innovazioni tecnologiche, in Solaris 1 (Sunos) i progettisti cambiarono le tecniche tradizionali di unix. Quando un processo va in esecuzione, le pagine contenenti il codice eseguibile sono prelevate dal file system, poste in memoria centrale e, qualora vengano selezionate per essere sovrascritte, eliminate. Rileggere una pagina dal file system è più efficiente che scriverla nell'area d'avvicendamento e rileggerla da lì: l'area d'avvicendamento è adoperata esclusivamente come area

di stoccaggio per le pagine della memoria anonima, di cui fanno parte la memoria allocata per lo stack, quella per lo heap e i dati non inizializzati dei processi.

Ulteriori cambiamenti sono stati introdotti nelle versioni più recenti di Solaris. Il più importante prevede che l'area d'avvicendamento sia allocata solo quando una pagina è portata fuori dalla memoria fisica, anziché quando la pagina è creata per la prima volta in memoria virtuale. Questa regola produce un miglioramento delle prestazioni nei calcolatori moderni, i quali, rispetto ai sistemi più vecchi, possono contare su una maggiore quantità di memoria fisica e limitare il ricorso alla paginazione.

Così come per Solaris, l'area d'avvicendamento di Linux è utilizzata soltanto per la memoria anonima, ovvero per la memoria che non corrisponde ad alcun file. Linux permette di istituire una o più aree d'avvicendamento, sia in file di avvicendamento (*swap file*) del file system regolare, sia in una partizione dedicata. Un'area d'avvicendamento è formata da una serie di moduli di 4 kb detti slot delle pagine, la cui funzione è di conservare le pagine avvicendate. Ogni area d'avvicendamento ha una mappa d'avvicendamento (*swap map*) associata, ovvero un array di contatori interi, ciascuno dei quali corrisponde a uno slot nell'area d'avvicendamento. Se un contatore segna il valore 0, la pagina che gli corrisponde è disponibile. Valori superiori a 0 indicano che lo slot è occupato da una delle pagine avvicendate. Il valore del contatore indica il numero di collegamenti alla pagina; se esso è 3, per esempio, allora la pagina avvicendata è associata a tre processi differenti, eventualità possibile nel caso che la pagina rappresenti una regione di memoria condivisa da tre processi. Le strutture dati relative all'avvicendamento nei sistemi Linux sono rappresentate dalla Figura 11.11.

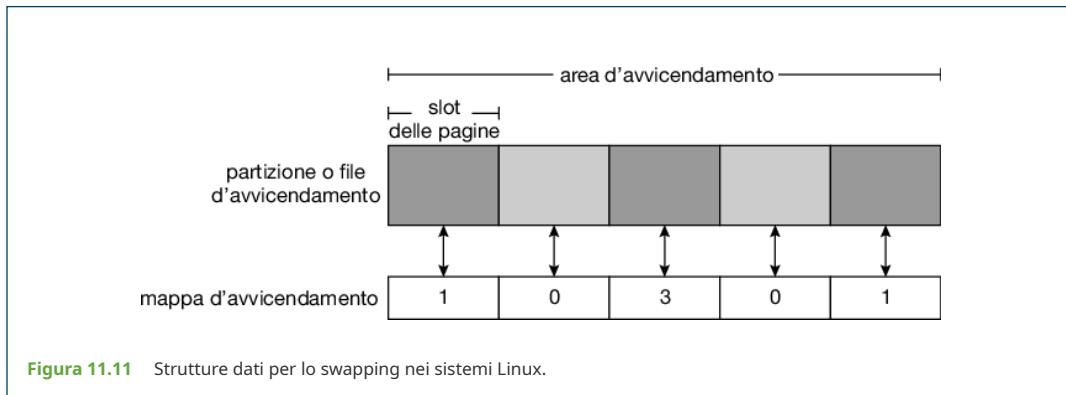


Figura 11.11 Strutture dati per lo swapping nei sistemi Linux.

11.7 Connessione dei dispositivi di memorizzazione

I calcolatori accedono alla memoria secondaria in tre modi: tramite un dispositivo collegato alla macchina, tramite un dispositivo connesso alla rete o tramite un dispositivo cloud.

11.7.1 Memoria secondaria connessa alla macchina

Alla memoria secondaria connessa alla macchina (*host-attached storage*) si accede dalle porte locali di i/o. Queste porte utilizzano diverse tecnologie, la più comune delle quali è sata, come accennato in precedenza. Un sistema tipico è dotato di una o più porte sata.

Per consentire a un sistema di accedere a un maggior spazio di archiviazione è possibile collegare un singolo dispositivo di archiviazione, un dispositivo installato in uno chassis oppure più unità installate in uno stesso chassis tramite porte e cavi usb, FireWire o Thunderbolt.

Le stazioni di lavoro di fascia alta e i server hanno in genere bisogno di più spazio di memorizzazione o della possibilità di condividere la memoria secondaria, quindi impiegano architetture più raffinate come fc (*fibre channel*), un'architettura seriale ad alta velocità che può funzionare sia su fibra ottica sia su un cavo con 4 conduttori di rame. Grazie al vasto spazio d'indirizzi e alla natura commutata della comunicazione, si possono connettere più macchine e dispositivi di memorizzazione alla struttura a commutazione, permettendo una notevole flessibilità nella comunicazione di i/o.

C'è un gran numero di dispositivi utilizzabili come memoria secondaria connessa alla macchina, tra questi le unità a disco, i dispositivi nvm, le unità a cd, dvd e Blu Ray, i nastri magnetici e le storage area network (san), descritte nel Paragrafo 11.7.4. I comandi di i/o che avviano trasferimenti di dati a un dispositivo di memoria connessa alla macchina sono letture e scritture di blocchi logici di dati, dirette a unità di memorizzazione specificamente identificate (per esempio, tramite bus id e unità logica del dispositivo).

11.7.2 Memoria secondaria connessa alla rete

Un dispositivo nas (*network-attached storage*) fornisce l'accesso allo spazio di archiviazione attraverso una rete (Figura 11.12) e può essere costituito da un sistema di memoria specializzato oppure da un normale computer che fornisce il suo spazio di archiviazione ad altri host attraverso la rete. I client accedono alla memoria connessa alla rete tramite un'interfaccia rpc, come l' nfs nel caso dei sistemi unix e Linux o cifs nel caso di sistemi Windows. Le chiamate di procedura remota (rpc) sono realizzate per mezzo dei protocolli tcp o udp sopra una rete ip (di solito la stessa rete locale che porta tutto il traffico dati ai client). Può quindi risultare più semplice pensare a nas come a un ulteriore protocollo di accesso alla memoria secondaria. L'unità di memoria è normalmente realizzata come una batteria di dispositivi di memorizzazione (storage array) con programmi di controllo che implementano l'interfaccia per le rpc.

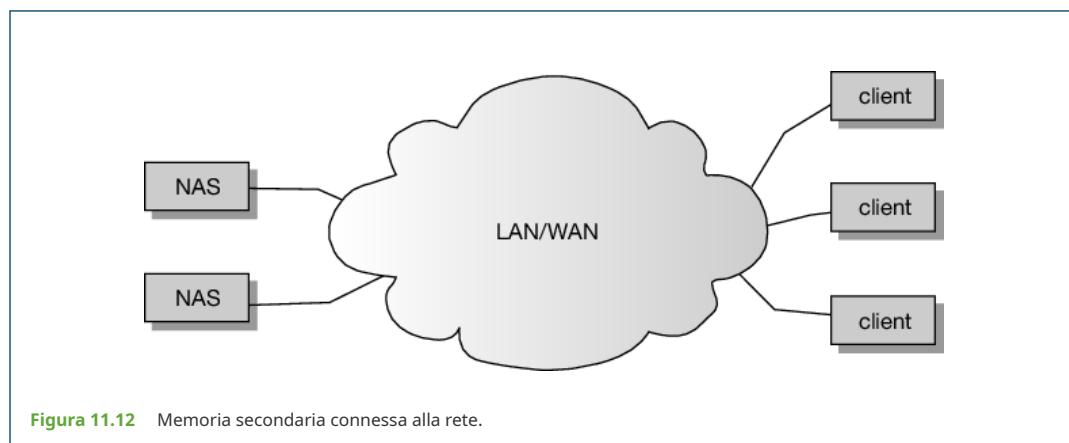


Figura 11.12 Memoria secondaria connessa alla rete.

cifs e nfs offrono varie funzioni di lock, consentendo la condivisione di file tra host che accedono a un nas mediante tali protocolli. Per esempio, un utente che ha effettuato l'accesso a più client di un nas può accedere alla sua home directory da tutti questi client contemporaneamente.

La memoria secondaria connessa alla rete fornisce un modo semplice per condividere spazio di storage per tutti i calcolatori di una lan, con la stessa facilità di gestione dei nomi e degli accessi caratteristica della memoria secondaria connessa alla macchina. Tuttavia, un sistema di questo genere tende a essere meno efficiente e ad avere prestazioni inferiori rispetto ad alcuni sistemi con connessione diretta alla macchina.

iscsi è il più recente protocollo per la memoria connessa alla rete. Essenzialmente sfrutta il protocollo ip della rete per il trasporto del protocollo scsi. Ne consegue la possibilità di usare la rete invece che cavi scsi per connettere le diverse macchine alla memoria secondaria. Uno dei vantaggi di questa tecnica è che le macchine sono in grado di trattare la memoria secondaria come se fosse

direttamente collegata, sebbene possa essere collocata a distanza. Mentre nfs e cifs presentano un file system e inviano parti di file attraverso la rete, iscsi invia sulla rete blocchi logici e lascia al client la possibilità di utilizzare direttamente i blocchi o di creare con essi un file system.

11.7.3 Memoria secondaria su cloud

Nel Paragrafo 1.10.5 è stato trattato il cloud computing. Una delle offerte dei fornitori di servizi cloud è lo spazio di archiviazione su cloud (*cloud storage*). Come nel caso della memoria secondaria collegata alla rete, quella su cloud offre accesso allo spazio di archiviazione attraverso una rete ma, a differenza dei nas, l'accesso allo spazio di archiviazione avviene tramite Internet o tramite un'altra wan verso un data center remoto che fornisce spazio a pagamento (o talvolta gratuitamente).

Un'altra differenza tra nas e cloud storage è il modo in cui lo spazio di archiviazione è accessibile e presentato agli utenti. Utilizzando i protocolli cifs o nfs è possibile accedere a un nas come si accede a un qualsiasi altro file system, mentre se viene utilizzato il protocollo iscsi vi si accede come a una unità di memoria a blocchi. La maggior parte dei sistemi operativi integra questi protocolli e presenta lo storage nas allo stesso modo degli altri sistemi di memorizzazione. Il cloud storage è invece basato su api, e i programmi utilizzano le api per accedere allo spazio di memorizzazione. Amazon S3 è un'offerta leader di cloud storage, Dropbox, un esempio di azienda che fornisce app per connettersi allo storage, Microsoft OneDrive e Apple iCloud sono altri esempi di archiviazione su cloud.

Uno dei motivi per cui vengono utilizzate le api al posto dei protocolli esistenti è la latenza e gli scenari di errore di una wan. I protocolli nas sono stati progettati per essere utilizzati nelle reti lan, che hanno una latenza inferiore rispetto alle wan e sono meno soggette alla perdita di connettività tra l'utente e il dispositivo di memorizzazione. Se una connessione lan viene interrotta, un sistema che utilizza nfs o cifs potrebbe bloccarsi fino alla ripresa del normale funzionamento. Quando si utilizzano servizi cloud errori del genere sono più probabili e un'applicazione interrompe semplicemente l'accesso fino al ripristino della connettività.

11.7.4 Storage-area network e storage array

Uno svantaggio dei sistemi di memoria secondaria connessa alla rete è che le operazioni di i/o sulla memoria secondaria impegnano banda della rete e quindi aumentano la latenza della comunicazione nella rete. Questo problema può essere particolarmente grave per sistemi client-server di grandi dimensioni: l'ordinaria comunicazione tra i server e i client compete per la banda con la comunicazione tra i server e i dispositivi di memorizzazione.

Una storage-area network (san) è una rete privata (che impiega protocolli specifici per la memorizzazione anziché protocolli di rete) tra i server e le unità di memoria secondaria (Figura 11.13). La potenza di una san sta nella sua flessibilità: si possono connettere alla stessa san molte macchine e molti storage array; la memoria può essere allocata alle macchine dinamicamente. Gli storage array possono essere protetti da raid o essere costituiti da unità non protette (jbod, *Just a Bunch of Disks*). Uno switch san permette di consentire o proibire l'accesso degli host allo spazio di archiviazione. Per esempio, la san può essere configurata per allocare più spazio di archiviazione a un host che sta esaurendo lo spazio su disco. Le san consentono ai cluster di server di condividere lo stesso storage e agli storage array di avere più connessioni dirette con gli host. Le san in genere hanno più porte e costano di più rispetto agli storage array. La connettività san avviene a breve distanza e in genere senza routing, quindi un nas può avere molti più host connessi rispetto a una san.

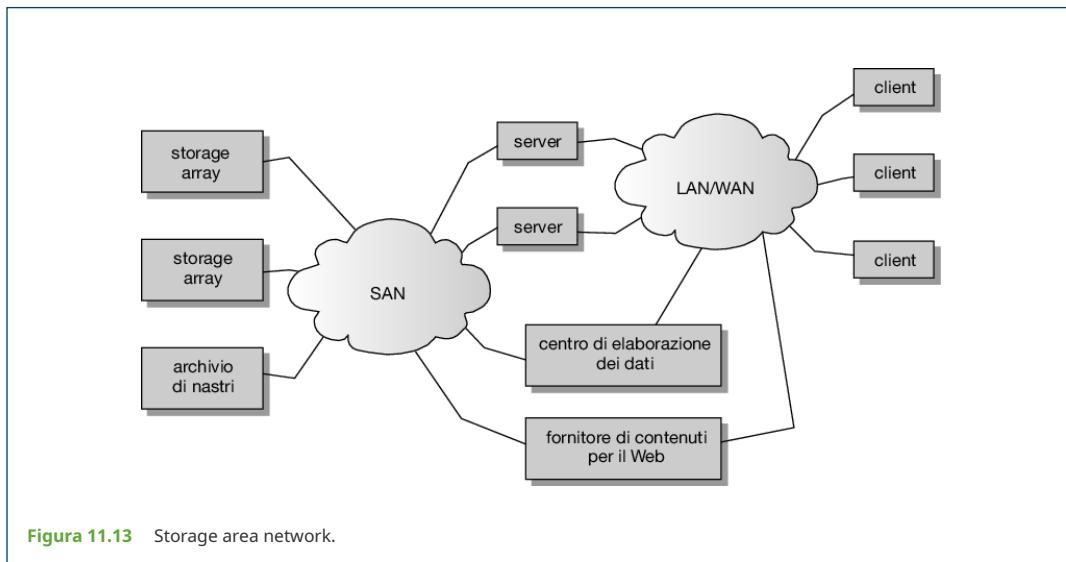


Figura 11.13 Storage area network.

Uno storage array (si veda la Figura 11.14) è un dispositivo costruito appositamente che può includere porte san, porte di rete o entrambe. Contiene inoltre unità per memorizzare dati e un controllore (o un set ridondante di controllori) per gestire l'archiviazione e consentire l'accesso storage attraverso le reti. I controllori sono composti da cpu, memoria e software che implementano le

funzionalità dell'array, che possono comprendere protocolli di rete, interfacce utente, protezione raid, snapshot, replica, compressione, deduplicazione e crittografia. Alcune di queste funzionalità sono discusse nel Capitolo 14.



Figura 11.14 Batteria di dispositivi di memorizzazione (storage array).

Alcuni storage array includono dischi ssd: in alcuni casi sono composti esclusivamente da ssd, con prestazioni massime, ma capacità inferiore di archiviazione, in altri contengono un mix di ssd e hdd, con il software dell'array (o l'amministratore) che seleziona il supporto migliore per un dato utilizzo oppure con l'utilizzo degli ssd come cache e degli hdd come memoria di massa.

La soluzione più comune per il collegamento tramite san è il fc, anche se la diffusione di iscsi è in crescita, grazie alla sua semplicità. Una possibile alternativa è InfiniBand, un'architettura di bus specifica per san, che fornisce supporto hardware e software per reti d'interconnessione ad alta velocità tra server e unità di memoria secondaria.

11.8 Strutture RAID

L'evoluzione tecnologica ha reso le unità di memorizzazione progressivamente più piccole e meno costose, tanto che oggi è economicamente possibile equipaggiare un sistema elaborativo con molti dischi. La presenza di più dischi, qualora si possano usare in parallelo, rende possibile l'aumento della frequenza a cui i dati si possono leggere o scrivere. Inoltre, una configurazione di questo tipo permette di migliorare l'affidabilità della memoria secondaria, poiché diventa possibile memorizzare le informazioni in più dischi in modo ridondante. In questo caso, un guasto a uno dei dischi non comporta la perdita di dati. Ci sono varie tecniche per l'organizzazione dei dischi, note col nome comune di batterie ridondanti di dischi (*redundant array of independent disks, raid*), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

Nel passato, strutture raid composte da piccoli dischi economici erano viste come un'alternativa economicamente vantaggiosa rispetto a costosi dischi di grande capacità; oggi, le strutture raid s'impiegano per la loro maggiore affidabilità e velocità di trasferimento dei dati, piuttosto che per ragioni economiche. Quindi, la *I* in raid viene attualmente letta *independent* anziché *inexpensive* com'era originariamente.

STRUTTURA DEI DISPOSITIVI RAID

Le memorie raid si prestano a essere strutturate con modalità diverse. Un sistema, per esempio, può collegare direttamente i dischi ai propri bus, nel qual caso la funzionalità raid può essere realizzata dal sistema operativo o dai programmi di sistema. In alternativa, un controllore intelligente può gestire diversi dischi collegati alla macchina e implementare una struttura raid per quei dischi a livello hardware. Infine, si può ricorrere a una **batteria raid** (*raid array*), un'unità a sé stante, dotata di un controllore, di una cache (nella maggioranza dei casi) e di dischi autonomi. L'array è collegato alla macchina attraverso uno o più controlleri (ad esempio fc). Questa diffusa organizzazione consente a programmi e sistemi operativi di per sé privi della funzionalità raid di usufruirne comunque.

11.8.1 Miglioramento dell'affidabilità tramite la ridondanza

Consideriamo in primo luogo l'affidabilità dei raid di dischi. La possibilità che uno dei dischi in un insieme di n dischi si guasti è molto più alta della possibilità che uno specifico disco presenti un guasto. Si supponga che il tempo medio di guasto (*mean time between failures, mtbf*) di un singolo disco sia 100.000 ore. In questo caso, il tempo medio di guasto per un qualsiasi disco in una batteria di 100 dischi sarebbe $100.000/100 = 1000$ ore, o 41,66 giorni: non molto tempo! Se si memorizzasse una sola copia dei dati, allora ogni guasto di un disco comporterebbe la perdita di una notevole quantità di dati; una frequenza di perdita di dati così alta sarebbe inaccettabile.

La soluzione al problema dell'affidabilità sta nell'introdurre una certa ridondanza, cioè nel memorizzare informazioni che non sono normalmente necessarie, ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse. Il raid può essere applicato anche ai dispositivi nvm, sebbene i dispositivi nvm non abbiano parti mobili e siano quindi meno soggetti a guasti rispetto agli hdd.

Il metodo più semplice (ma anche il più costoso) di introduzione di ridondanza è quello della duplicazione di ogni disco. Questo metodo è detto *mirroring* (*copiatura speculare*): ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Si ottiene un disco duplicato (detto *mirrored volume*). Se uno dei dischi si guasta, i dati si possono leggere dall'altro. I dati si perdono solo se il secondo disco si guasta prima della sostituzione del disco già guasto.

L'*mtbf* di un disco duplicato, dove per *guasto* s'intende ora la perdita di dati, dipende da due fattori: il tempo medio tra due guasti di un singolo disco e il tempo medio di riparazione, cioè il tempo richiesto (in media) per sostituire un disco guasto e ripristinarvi i dati. Supponendo che i possibili guasti dei due dischi siano indipendenti, vale a dire che il guasto di un disco non sia mai legato a quello dell'altro, se il tempo medio di guasto di un singolo disco è 100.000 ore e il tempo medio di riparazione è di 10 ore, allora il tempo medio di perdita di dati di un sistema con mirroring è $100.000^2/(2 * 10) = 500 * 10^6$ ore, che corrispondono a 57.000 anni!

Occorre però notare che non è possibile assumere l'ipotesi di indipendenza tra i guasti dei dischi. Improvvisi cali di tensione e disastri naturali, quali terremoti, incendi e alluvioni, danneggerebbero con tutta probabilità entrambi i dischi. Inoltre, difetti di fabbricazione in una partita di dischi possono causare guasti correlati. Con l'invecchiamento del disco, la probabilità di un guasto aumenta, accrescendo la probabilità che un secondo disco si guasti mentre il primo è in riparazione. Tuttavia, nonostante tutte queste considerazioni, i sistemi con mirroring offrono un'affidabilità assai più alta dei sistemi a disco singolo.

I casi di caduta di alimentazione elettrica costituiscono un problema particolarmente sentito, poiché avvengono con una frequenza molto più alta dei disastri naturali. Anche impiegando il mirroring, se si sta svolgendo un'operazione di scrittura nello stesso blocco in entrambi i dischi e si verifica una caduta di alimentazione prima che sia completata la scrittura dell'intero blocco, i due blocchi

possono ritrovarsi in uno stato incerto. Una soluzione prevede la scrittura di una delle due copie e solo successivamente la scrittura della seconda. Un'altra soluzione è aggiungere una memoria non volatile a stato solido (nvram, *non-volatile ram*) alla batteria raid, protetta dalla perdita di dati causata dalle cadute di alimentazione: se è dotata di forme di correzione d'errore come ecc o mirroring, la scrittura dei dati nella cache può essere considerata completa anche in quei casi.

11.8.2 Miglioramento delle prestazioni tramite il parallelismo

Vediamo come l'accesso in parallelo a più dischi può migliorare le prestazioni. Con il mirroring, la frequenza con la quale si possono gestire le richieste di lettura raddoppia, poiché ciascuna richiesta si può inviare indifferentemente a uno dei due dischi (sempre che entrambi i dischi siano funzionanti, condizione che è quasi sempre soddisfatta). La capacità di trasferimento di ciascuna lettura è la stessa di quella di un sistema a singolo disco, ma il numero di letture per unità di tempo raddoppia.

Attraverso l'uso di più unità di memorizzazione è possibile anche (o alternativamente) migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dispositivi. La forma più semplice di questa distribuzione (*data striping*), consiste nel distribuire i bit di ciascun byte su più dispositivi; in questo caso si parla di sezionamento o striping a livello dei bit. Per esempio, se il sistema impiega un array di otto dispositivi, si scriverà il bit i di ciascun byte nel disco i . L'array di otto dispositivi si può trattare come un unico dispositivo avente settori che hanno una dimensione otto volte superiore a quella normale e, soprattutto, che hanno una capacità di trasferimento otto volte superiore. In un'organizzazione di questo tipo, ogni dispositivo è coinvolto in ogni accesso (lettura o scrittura che sia), così che il numero di accessi che si possono gestire nell'unità di tempo è circa lo stesso di quello per un sistema a dispositivo singolo, ma ogni accesso permette di leggere una quantità di dati pari a otto volte quella che si può leggere con un singolo dispositivo.

Lo striping a livello dei bit si può generalizzare a un numero di dispositivi multiplo di 8 o che divide 8. Per esempio, se un sistema adopera un array di quattro dispositivi, i bit i e $i + 4$ di ciascun byte si memorizzano nel disco i . Inoltre, lo striping non si deve realizzare necessariamente a livello dei bit di un byte: nello striping a livello di blocco, per esempio, i blocchi di un file si distribuiscono su più dispositivi; con n dispositivi, il blocco i di un file si memorizza nel dispositivo $(i \bmod n) + 1$. Sono possibili anche altri livelli di striping, come quelli basati sui byte di un settore o sui settori di un blocco, ma lo striping a livello di blocco è il più comune.

Riassumendo, gli obiettivi principali del parallelismo mediante striping in un sistema di dispositivi sono due:

1. l'aumento, tramite il bilanciamento del carico, del throughput per accessi multipli a piccole porzioni di dati (cioè accessi a pagine);
2. la riduzione del tempo di risposta relativo ad accessi a grandi quantità di dati.

11.8.3 Livelli raid

La tecnica di mirroring offre un'alta affidabilità ma è costosa; la tecnica di striping (sezionamento) offre un'alta capacità di trasferimento dei dati, ma non migliora l'affidabilità. Sono stati proposti numerosi schemi per fornire ridondanza a basso costo usando l'idea dello striping combinata con i bit di parità (che vedremo a breve). Questi schemi realizzano diversi compromessi tra costi e prestazioni e sono stati classificati in livelli chiamati livelli raid, che la Figura 11.15 mostra graficamente (nella figura, la lettera P indica i bit di correzione degli errori, la lettera C indica una seconda copia dei dati). In tutti i casi riportati nella figura, sono presenti quattro dischi di dati, mentre i dischi supplementari s'impiegano per memorizzare le informazioni ridondanti per il ripristino dai guasti.

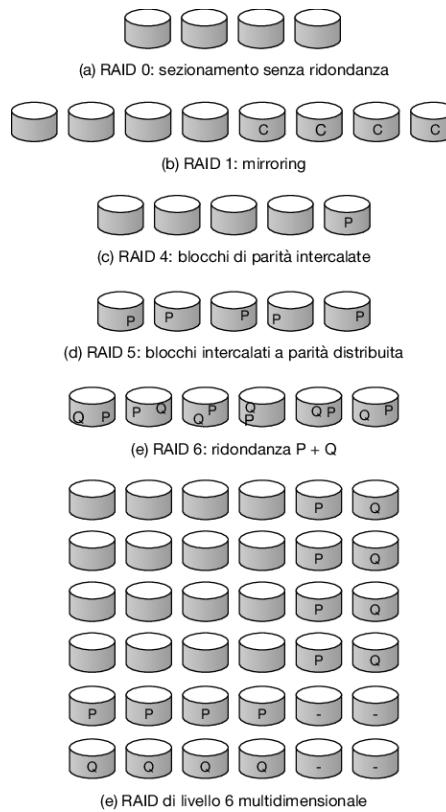


Figura 11.15 Livelli raid.

- raid di livello 0. Il livello 0 si riferisce ad array di dischi con striping a livello di blocchi, ma senza ridondanza (come il mirroring o i bit di parità), come illustrato nella Figura 11.15 (a).
- raid di livello 1. Il livello 1 si riferisce alla tecnica di mirroring. La Figura 11.15(b) mostra un'organizzazione basata sul mirroring.
- raid di livello 4. Il livello 4 è anche noto come organizzazione con codici per la correzione degli errori di memoria (ecc). Tale organizzazione è anche usata nei raid di livello 5 e 6.

La stessa idea alla base degli ecc (*error-correcting codes*) si può usare immediatamente negli array di memorizzazione eseguendo lo striping dei byte presenti nelle unità di memorizzazione. Per esempio, il primo bit di ogni byte si potrebbe memorizzare nell'unità 1, il secondo bit nell'unità 2, e così via fino alla memorizzazione dell'ottavo bit nell'unità 8 e alla memorizzazione dei bit di correzione degli errori in ulteriori unità. Questo schema è rappresentato graficamente nella Figura 11.15(c), dove i dispositivi etichettati con la lettera *P* contengono i bit di correzione. Se una delle unità si guasta, il ricalcolo del codice di correzione degli errori lo rileva e impedisce il passaggio dei dati al processo richiedente, generando un errore.

Il livello 4 può effettivamente correggere gli errori, anche se c'è soltanto un blocco ecc, considerando che, a differenza dei sistemi di memoria centrale, i controlleri dei dischi possono rilevare se un settore è stato letto correttamente, così che un unico bit di parità si può usare sia per individuare gli errori sia per correggerli. L'idea è la seguente: se uno dei settori è danneggiato, si conosce esattamente di quale settore si tratta e, per ogni bit nel settore, è possibile determinare se debba avere valore 1 o 0 calcolando la parità dei bit corrispondenti dai settori negli altri dischi. Se la parità dei rimanenti bit è uguale a quella memorizzata, il bit mancante è 0, altrimenti è 1.

La lettura di un blocco richiede l'accesso a un solo dispositivo, permettendo la gestione di altre richieste da parte di altri dischi. Quindi, la capacità di trasferimento dei dati per ciascun accesso è minore, ma gli accessi in lettura possono procedere in modo parallelo ottenendo una velocità complessiva di i/o più alta. La capacità di trasferimento per la lettura di molti dati è alta, poiché si possono leggere in modo parallelo tutti i dischi e anche le operazioni di scrittura di grandi quantità di dati presentano un'alta capacità di trasferimento, poiché i dati e i bit di parità si possono scrivere in parallelo.

Scritture indipendenti di modeste entità non si possono eseguire in parallelo. La scrittura da parte del sistema operativo di una quantità di dati inferiore a un blocco richiede la lettura del blocco, la sua modifica, e la scrittura del blocco modificato; anche il blocco di parità deve essere aggiornato. Si parla a questo proposito del ciclo lettura-modifica-scrittura. Una singola richiesta di scrittura comporta pertanto quattro accessi al disco, due in lettura e due in scrittura.

Nel Capitolo 14 sarà presentato wafl: esso adotta raid di livello 4, che permette l'aggiunta di dischi al sistema senza soluzione di continuità. Inizializzando i nuovi dischi a zero, la parità non cambia, e l'array raid è ancora nello stato corretto.

Il raid di livello 4 presenta due vantaggi rispetto al livello 1, fornendo allo stesso tempo un'uguale protezione dei dati. Il primo vantaggio è che si usa un solo disco per la parità dei dati memorizzati in diversi dischi di dati, anziché un disco di mirroring per ciascun disco di dati come nel livello 1. Il secondo vantaggio è che, essendo le letture e le scritture dei byte distribuite su più dischi con uno striping a n vie, la velocità di trasferimento di un singolo blocco è pari a n volte quella del raid di livello 1.

Un altro problema di prestazioni riguardante il raid di livello 4 (come per tutti i livelli raid basati sui bit di parità) è il tempo richiesto dal calcolo e dalla scrittura della parità. Questo tempo aggiuntivo determina operazioni di scrittura significativamente più lente rispetto ad array raid senza parità. Tuttavia, le moderne cpu general-purpose sono molto veloci rispetto all'i/o sui dispositivi e l'impatto in termini di prestazioni può essere minimo. Inoltre, molti array raid dispongono di un controllore capace di gestire il calcolo della parità. Questo sposta il carico dovuto al calcolo della parità dalla cpu all'array di dischi. L'array ha anche una cache nvram per memorizzare i blocchi mentre viene calcolata la parità e per memorizzare transitoriamente le scritture dal controllore alle unità. Questa tecnica può evitare la maggior parte dei cicli di lettura-modifica-scrittura raccogliendo i dati da scrivere in una sezione e scrivendo contemporaneamente su tutte le unità della sezione. Questa combinazione di accelerazione hardware e buffering può rendere la tecnica raid con parità altrettanto veloce di quella senza parità; infatti, un array raid con cache e bit di parità può avere prestazioni migliori di un'organizzazione raid senza cache e senza parità.

- raid di livello 5. Il livello 5, o organizzazione con blocchi intercalati a parità distribuita, differisce dal livello 4 per il fatto che, invece di memorizzare i dati in n unità e la parità in un disco separato, i dati e le informazioni di parità sono distribuite tra le $n + 1$ unità. Per ogni blocco, una delle unità memorizza la parità e le altre i dati. Per esempio, considerando una batteria di cinque unità, la parità per il blocco m -esimo si memorizza nell'unità $(m \bmod 5) + 1$, mentre i blocchi m -esimi delle altre quattro unità contengono i dati effettivi per quel blocco. Questo schema è illustrato nella Figura 11.15(d), dove i simboli P sono distribuiti su tutte le unità. Un blocco di parità non può contenere informazioni di parità per blocchi che risiedono nella stessa unità, poiché un guasto all'unità provocherebbe sia la perdita di dati sia la perdita dell'informazione di parità e quindi i dati non sarebbero ripristinabili. Con la distribuzione della parità sui diverse unità, il raid di livello 5 evita un uso intensivo dell'unità dove risiede la parità, che invece si ha con il raid di livello 4. raid 5 è il più comune sistema di parità raid.
- raid di livello 6. Il livello 6, detto anche schema di ridondanza $P + Q$, è molto simile al raid di livello 5, ma memorizza ulteriori informazioni ridondanti per poter gestire guasti contemporanei di più dischi. La parità xor non può essere utilizzata su entrambi i blocchi di parità perché sarebbero identici e non potrebbero fornire informazioni di ripristino. Invece della parità, per calcolare Q vengono utilizzati codici di correzione degli errori basati sulla matematica dei campi di Galois. Nello schema mostrato nella Figura 11.15 (e) sono memorizzati 2 blocchi di dati ridondanti per ogni 4 blocchi di dati (a differenza di 1 blocco di parità usato nel livello 5) e il sistema risultante può tollerare due guasti delle unità.
- raid di livello 6 multidimensionale. Alcuni sofisticati storage array ampliano il raid di livello 6. Si consideri un array contenente centinaia di unità. L'inserimento di tali unità in una sezione raid di livello 6 determinerebbe la presenza di molte unità dati e solo due unità di parità logica. Il raid di livello 6 multidimensionale organizza logicamente le unità in righe e colonne (in array di dimensione due o superiore) e implementa il raid di livello 6 sia orizzontalmente lungo le righe che verticalmente lungo le colonne. Il sistema è in grado di ripristinare qualsiasi errore, o anche più errori, utilizzando blocchi di parità in una di queste posizioni. Questo livello di raid è mostrato nella Figura 11.15(f). Per semplicità, la figura mostra la parità raid su unità dedicate, ma in realtà i blocchi raid sono sparsi su righe e colonne.
- raid di livello $0 + 1$ e $1 + 0$. Il livello $0 + 1$ consiste in una combinazione dei livelli raid 0 e 1. Il livello 0 fornisce le prestazioni, mentre il livello 1 l'affidabilità. Di solito, questo schema porta a prestazioni migliori rispetto al livello 5 e si usa prevalentemente negli ambienti in cui sono importanti sia le prestazioni sia l'affidabilità. Sfortunatamente, questo schema richiede, come il raid di livello 1, un raddoppio del numero di dischi necessario per memorizzare i dati, quindi è anche relativamente costoso. Nel raid di livello $0 + 1$, si effettua lo striping su un insieme di dischi e si duplica ogni sezione con la tecnica del mirroring.

Un altro metodo che sta diventando disponibile commercialmente è il raid di livello $1 + 0$, in cui si fa prima il mirroring dei dischi a coppie, e poi lo striping di queste coppie. Questo schema raid ha alcuni vantaggi teorici rispetto al raid $0 + 1$. Per esempio, se si guasta una singola unità nel raid $0 + 1$, l'intera sezione di dati diventa inaccessibile, lasciando disponibile solo l'altra sezione. Con un guasto nel raid $1 + 0$, la singola unità diventa inaccessibile, ma il suo duplice è ancora disponibile, come tutte le altre unità (Figura 11.16).

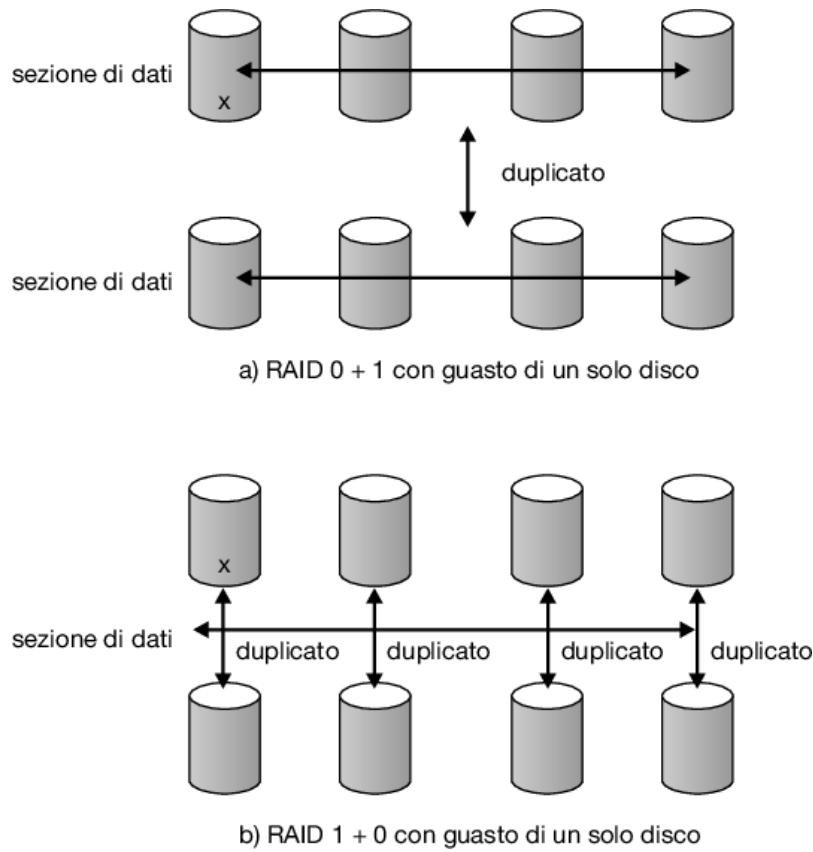


Figura 11.16 raid 0 + 1 e 1 + 0 con guasto di un solo disco.

Sono state proposte numerose altre varianti agli schemi raid di base illustrati sopra e questo ha portato anche una certa confusione nelle precise definizioni dei diversi livelli raid.

Un altro aspetto soggetto a molte varianti è l'implementazione del raid. Esaminiamo i diversi strati architetturali a cui è possibile implementare un sistema raid.

- Il software per la gestione dei volumi può implementare un sistema raid all'interno del kernel o a livello dei programmi di sistema. In questo caso, nonostante i dispositivi per la memorizzazione possano fornire funzionalità minime, è possibile ottenere un sistema raid completo.
- Il raid può essere implementato a livello hardware dall'adattatore del bus della macchina (*host bus adapter*, hba). Solo i dischi connessi direttamente all'hba possono costituire parte integrante di una dato array raid. Questa soluzione è a basso costo, ma non è molto flessibile.
- Il raid può essere implementato a livello hardware dall'array di dischi. È così possibile creare sistemi raid a vari livelli, e persino ricavare da essi volumi più piccoli, che sono quindi presentati al sistema operativo, che avrà solo da realizzare il file system su ciascuno dei volumi. Gli array possono disporre di connessioni multiple o far parte di una rete di memorizzazione secondaria (san), consentendo a varie macchine di sfruttare le funzionalità dell'array.
- Il raid può essere implementato da dispositivi di virtualizzazione del disco a livello di interconnessione san. In questo caso, un dispositivo funge da intermediario tra le macchine e l'area di memorizzazione, accettando istruzioni dai server e gestendo l'accesso alla memoria secondaria. Esso potrebbe, per esempio, attuare il mirroring, trascrivendo ciascun blocco su due distinti dispositivi di memorizzazione.

Ulteriori funzionalità, come quella di istantanea e di replica, possono essere implementate a ognuno di questi livelli. Un'istantanea (*snapshot*) è un'immagine del file system così com'era prima dell'ultimo aggiornamento (le istantanee saranno trattate più in dettaglio nel Capitolo 14). La replica prevede la duplicazione automatica di scritture su siti diversi, per finalità di ridondanza, o di ripristino in caso di eventi catastrofici (*disaster recovery*). La replica può essere sincrona o asincrona. Se è sincrona, ciascun blocco deve essere scritto sia localmente, sia in remoto, prima che la scrittura sia considerata completa; se è asincrona, si effettuano periodicamente scritture a gruppi. La replica asincrona espone al rischio di perdere i dati, se il sito principale fallisce, ma è più veloce e non ha limiti di distanza.

L'implementazione di queste funzionalità varia a seconda dello strato scelto per realizzare il sistema raid. Qualora raid, per esempio, sia implementato a livello software, ciascuna macchina può aver necessità di implementare e gestire la replica per proprio conto.

Tuttavia, se la replica avviene a livello dell'array di dischi o dell'interconnessione san, si possono replicare i dati della macchina a prescindere dal suo sistema operativo e dalle relative funzionalità.

Un'altra caratteristica spesso presente nei sistemi raid è la previsione di dischi di scorta (*hot spare*), che possono sostituire quelli normali in caso di guasti. Per esempio, un disco di scorta può essere usato per sostituire un disco danneggiato, ricostruendo l'integrità di una coppia in mirroring. In questo modo, si può ristabilire automaticamente lo stato corretto del livello raid, senza attendere che il disco difettoso sia sostituito. È possibile riparare più di un guasto, senza l'intervento di un operatore, con l'allocazione di più dischi di scorta.

11.8.4 Scelta di un livello RAID

Viste le svariate possibilità esistenti, ci si potrebbe chiedere quali siano i criteri di scelta dei progettisti nei confronti del livello raid. Una considerazione è il tempo di ricostruzione. Se un disco si guasta, il tempo necessario a ricostruire i dati che contiene può essere rilevante. Questo fattore può essere importante nel caso in cui venga richiesto un flusso continuo di dati, come nei database ad alte prestazioni o interattivi. Inoltre il tempo di ricostruzione influenza il tempo medio di guasto.

Il tempo di ricostruzione varia a seconda del livello raid utilizzato. La ricostruzione più semplice si ha per raid di livello 1, poiché i dati possono essere copiati da un'altra unità; per gli altri livelli, per ricostruire i dati in un'unità guasta è necessario accedere a tutte le altre unità dell'array. Il tempo necessario per la ricostruzione dei dati può essere di ore nel caso di sistemi raid di livello 5 con molte unità.

raid di livello 0 si usa nelle applicazioni ad alte prestazioni in cui le perdite di dati non sono critiche. Per esempio, nel calcolo scientifico, dove un data set viene caricato e analizzato, il raid di livello 0 funziona bene perché qualsiasi guasto all'unità richiederebbe solo una riparazione e il ricaricamento dei dati dalla fonte. Il raid di livello 1 si usa comunemente nelle applicazioni che richiedono un'alta affidabilità e un rapido ripristino. I livelli raid 0 + 1 e 1 + 0 si usano dove sia le prestazioni sia l'affidabilità sono importanti, per esempio per piccole basi di dati. A causa dell'elevata richiesta di spazio del raid di livello 1, per la memorizzazione di grandi quantità di dati, spesso si preferisce impiegare il raid di livello 5. Il livello 6 e il livello 6 multidimensionale sono i formati più comuni negli storage array, perché offrono buone prestazioni e una buona protezione senza la necessità di un eccessivo overhead.

Progettisti e amministratori di sistemi raid devono prendere anche altre decisioni importanti, per esempio riguardo al numero ottimale di dischi in un array e al numero di bit che ciascun bit di parità deve proteggere. Maggiore è il numero di dischi in un array, maggiore sarà la capacità di trasferimento dei dati, ma il sistema sarà anche più costoso. Maggiore è il numero di bit protetti da un singolo bit di parità, minore sarà lo spazio richiesto dai bit di parità; sarà però maggiore anche la probabilità che un secondo disco si guasti prima che un disco guasto sia riparato e questo porterebbe alla perdita di dati.

L'ARRAY INSERV

L'innovazione, grazie al quale sono di continuo introdotte soluzioni migliori, più veloci e meno costose, ridefinisce spesso i confini che separano le tecnologie già esistenti. Si consideri, per esempio, l'array InServ di 3Par. A differenza di molti altri, esso non richiede la configurazione di un insieme di dischi a un livello raid specifico, ma scomponete invece ogni disco in porzioni da 256 mb. Il metodo raid è pertanto applicato a livello di queste porzioni. Di conseguenza, vari livelli raid possono interessare lo stesso disco, e le sue porzioni vengono utilizzate per formare diversi volumi.

InServ mette inoltre a disposizione le istantanee, una funzionalità simile a quella del file system wafL. Le istantanee di InServ prevedono sia il formato lettura-scrittura sia il formato a sola lettura, consentendo a utenti multipli di montare copie di un dato file system senza doverne possedere una copia completa. Le modifiche eventualmente apportate dagli utenti alla propria copia sono di copiatura su scrittura, ragion per cui non hanno effetto sulle altre copie.

Un'altra innovazione è il cosiddetto **utility storage**. Alcuni file system non possono essere espansi, né compressi. I file system di questo genere mantengono costantemente le dimensioni originali: per qualsiasi modifica è necessaria la copiatura di dati. Un amministratore può configurare InServ per fornire a una macchina cospicue quantità di memoria logica, che all'inizio occupano solamente un piccolo spazio di memoria fisica. Mentre la macchina comincia a usare lo spazio di memorizzazione, dischi non ancora utilizzati sono assegnati alla macchina, fino a raggiungere il livello logico originale. In tal modo, la macchina è indotta a credere di possedere un vasto spazio di memorizzazione permanente, dove creare i propri file system, e così via. InServ può aggiungere o rimuovere dischi dal file system senza che il file system se ne accorga. Questa caratteristica può ridurre il numero complessivo di unità a disco necessarie alle macchine, o perlomeno ritardare l'acquisizione di nuovi dischi finché non divengano realmente necessari.

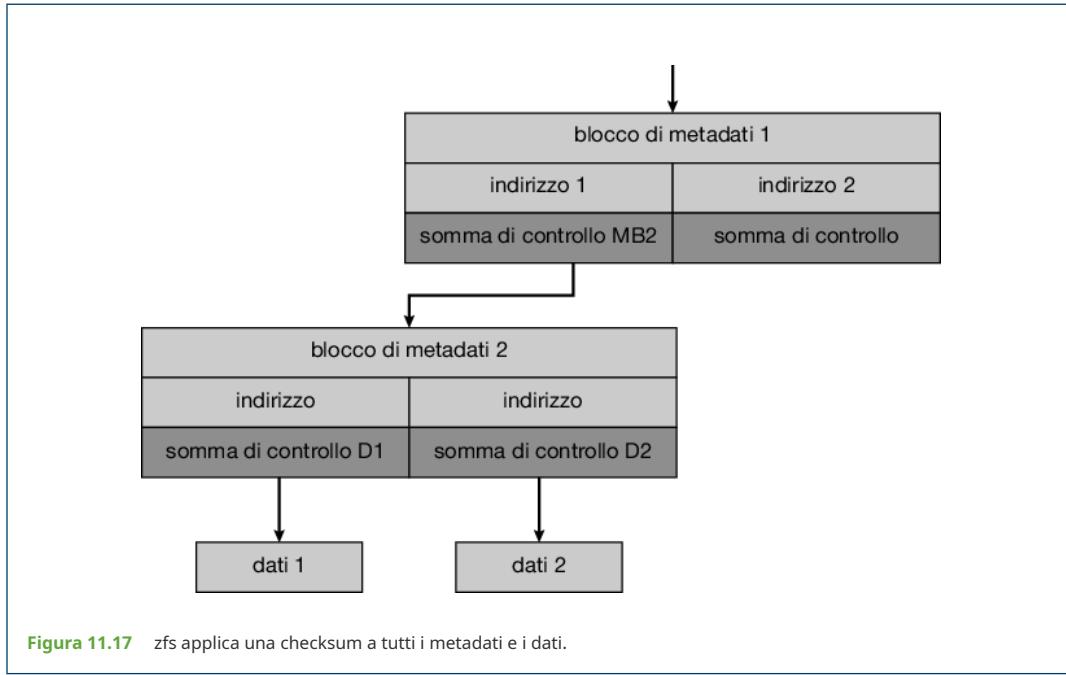
11.8.5 Estensioni

I concetti relativi ai sistemi raid sono stati generalizzati ad altri dispositivi di memorizzazione, comprese le unità a nastri e anche alla diffusione dei dati tramite sistemi senza fili (*wireless*). Con strutture raid applicate alle unità a nastri si possono ripristinare i dati anche se uno dei nastri della batteria è danneggiato. Se applicate alla trasmissione dei dati, si divide ogni blocco di dati in unità più piccole che si trasmettono insieme a un'unità di parità; se per qualsiasi ragione una delle unità non viene ricevuta, può essere ricostruita dalle altre. Di solito, con l'uso di unità automatiche dotate di molte unità a nastro si esegue lo striping dei dati su tutte le unità per aumentare il throughput e diminuire il tempo di backup.

11.8.6 Problemi connessi a RAID

I sistemi raid, purtroppo, non assicurano sempre la disponibilità dei dati al sistema operativo e agli utenti. Un puntatore a un file potrebbe essere errato, per esempio, e lo stesso potrebbe accadere ai puntatori nella struttura interna dei file. Le operazioni incomplete di scrittura, se non ripristinate in maniera adeguata, possono alterare i dati. Altri processi, inoltre, potrebbero scrivere accidentalmente sulle strutture del file system. raid protegge dagli errori derivanti dai supporti fisici per la memorizzazione, ma non da altri tipi di errori dovuti all'hardware e ai programmi. I pericoli potenziali, per i dati di un sistema, si estendono alla totalità degli errori derivanti dal software e dall'hardware.

Per risolvere tali problemi, il file system Solaris zfs ricorre a una strategia innovativa per verificare l'integrità dei dati. Esso applica una checksum (*somma di controllo*) interna a ogni blocco, dati e metadati inclusi. Le checksum non risiedono nel blocco sottoposto a controllo: ciascuna di esse, invece, è memorizzata insieme al puntatore a quel blocco (Figura 11.17). Si consideri un inode – una struttura dati per memorizzare i metadati del file system - con puntatori ai propri dati. All'interno dell'inode si trova la checksum per ciascun blocco di dati. Se si verifica un problema con i dati, la checksum darà un valore errato e il file system verrà a conoscenza del problema. Qualora sia attivo il mirroring, il sistema zfs, in presenza di un blocco con una checksum corretta e di uno con una checksum errata, sostituirà automaticamente il blocco errato con quello valido. In maniera simile, l'elemento della directory che punta all'inode possiede una checksum relativa all'inode. Qualunque problema riguardi l'inode, quindi, è rilevato al momento dell'accesso alla directory. Queste checksum, che sono applicate a tutte le strutture di zfs, producono risultati molto più efficaci degli ambienti raid o dei file system tradizionali, per livello di coerenza, rilevazione degli errori e capacità di correggerli. Il sovraccarico di gestione determinato dal calcolo delle checksum e dai cicli supplementari di lettura-modifica-scrittura dei blocchi non condizionano il funzionamento complessivo di zfs, che mantiene un'alta velocità nelle prestazioni. Una funzionalità di checksum simile è presente nel file system btrfs di Linux (si veda https://btrfs.wiki.kernel.org/index.php/Btrfs_design).



Un altro problema della maggior parte delle implementazioni raid è la mancanza di flessibilità. Considerate un array di memorizzazione dotato di venti unità divise in quattro insiemi da cinque unità. Ogni gruppo di cinque unità è un insieme raid di livello 5. Ne risultano quattro volumi separati, ciascuno contenente un proprio file system. Ma che cosa succede se un file system ha una dimensione troppo grande per un insieme raid di livello 5 a cinque unità? E se un altro file system necessita di un'area molto

ridotta? Se tali fattori sono noti in anticipo, allora i dischi e i volumi possono essere allocati adeguatamente. Frequentemente, però, l'utilizzo delle unità e le richieste variano nel tempo.

Anche se l'array di memorizzazione ha permesso all'intero insieme di venti unità di essere creato come un grande insieme raid, potrebbero insorgere altre problematiche. Nell'insieme potrebbero essere costruiti diversi volumi di dimensioni differenti. Alcuni gestori del volume non ci permettono però di cambiare la dimensione di un volume. In quel caso si ripresenterebbe la stessa situazione descritta in precedenza, ovvero dimensioni non adatte al file system. Alcuni gestori di volume permettono cambiamenti di dimensione, ma alcuni file system non permettono l'aumento o la diminuzione della dimensione del file system stesso. I volumi potrebbero cambiare dimensione, ma i file system dovrebbero essere ricreati per poter usufruire di quei cambiamenti.

zfs combina la gestione dei file system e quella dei volumi in una unità in grado di offrire una maggiore funzionalità rispetto a quella permessa dalla tradizionale separazione di tali funzioni. I dischi, o le partizioni di dischi, sono riuniti in gruppi di memorizzazione (*pools of storage*) attraverso insiemi raid. Un gruppo o pool può contenere uno o più file system zfs. Tutta l'area libera di un pool è a disposizione di tutti i file system contenuti all'interno di quel pool. zfs utilizza il modello di gestione della memoria basato su `malloc()` e `free()` per allocare e rilasciare memoria nei file system quando i blocchi vengono utilizzati e liberati all'interno del file system. Di conseguenza non ci sono limiti artificiali all'utilizzo della memoria e non sussiste la necessità di ridistribuire i file system tra i volumi né di ridimensionare i volumi. zfs stabilisce le quote per limitare la dimensione di un file system e definisce dei meccanismi di prenotazione per assicurarsi che il file system possa aumentare all'interno di una dimensione specificata, ma queste variabili possono essere sempre cambiate dal proprietario del file system. Altri sistemi, come Linux, hanno gestori di volumi che consentono l'unione logica di più dischi per creare volumi più grandi dei dischi stessi, in modo da poter contenere file system di grandi dimensioni. La Figura 11.18(a) illustra i volumi e i file system tradizionali, mentre la Figura 11.18(b) rappresenta il modello zfs.

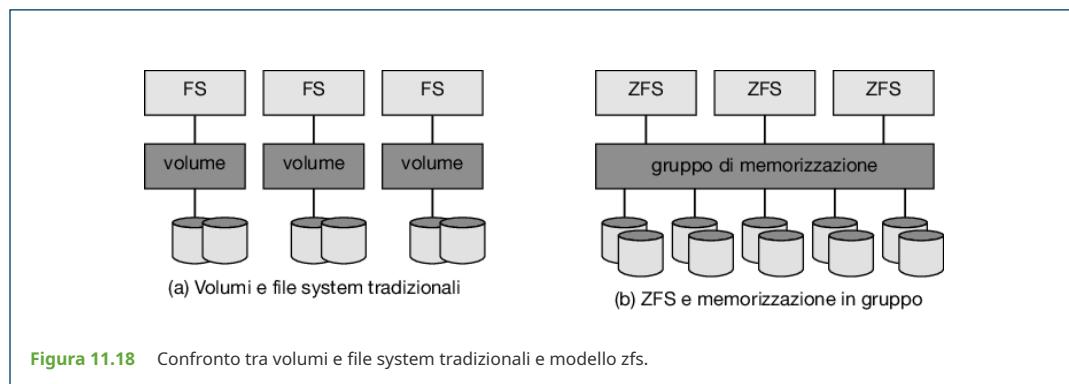


Figura 11.18 Confronto tra volumi e file system tradizionali e modello zfs.

11.8.7 Object storage

I computer general-purpose usano in genere i file system per archiviare i contenuti degli utenti. Un altro approccio all'archiviazione dei dati è iniziare con un gruppo (pool) di archiviazione e posizionare gli oggetti in quel gruppo. Questo approccio differisce dai file system, perché non fornisce la possibilità di navigare all'interno del gruppo alla ricerca degli oggetti. Piuttosto che essere orientati all'utente, possiamo dire che gli archivi di oggetti (detti anche object storage) sono orientati al computer e progettati per essere utilizzati dai programmi. Una sequenza di operazioni tipica su un archivio di oggetti è la seguente.

1. Creare un oggetto nel gruppo di archiviazione e ricevere un id dell'oggetto.
2. Accedere all'oggetto quando necessario, tramite il suo id.
3. Eliminare l'oggetto, tramite il suo id.

I software di gestione degli archivi di oggetti, come Hadoop file system (hdfs) e Ceph, determinano dove archiviare gli oggetti e gestiscono la protezione dei dati. In genere, la gestione è effettuata su hardware di base piuttosto che sugli array raid. Per esempio, hdfs può memorizzare N copie di un oggetto su N diversi computer. Questo approccio può essere meno costoso rispetto agli storage array e può fornire un accesso rapido all'oggetto (almeno sugli N sistemi dove è salvato). Tutti i sistemi in un cluster Hadoop possono accedere all'oggetto, ma solo i sistemi che dispongono di una copia hanno accesso rapido all'oggetto, tramite la copia. I calcoli sui dati vengono effettuati su tali sistemi e i risultati sono inviati attraverso la rete, per esempio, solo ai sistemi che li richiedono. Gli altri sistemi necessitano di connettività di rete per leggere e scrivere sull'oggetto. Gli archivi di oggetti vengono solitamente utilizzati per la memorizzazione di grandi quantità di dati, non per accesso casuale a velocità elevate, e hanno il vantaggio della scalabilità orizzontale, perché mentre uno storage array ha una capacità massima fissata, per aggiungere capacità a un archivio di oggetti aggiungiamo semplicemente più computer dotati di dischi interni o esterni al pool. I pool di archiviazione di oggetti possono avere dimensioni dell'ordine dei petabyte.

Un'altra caratteristica fondamentale degli archivi di oggetti è che ogni oggetto è auto-descrittivo, ovvero include la descrizione dei suoi contenuti. L'archiviazione di oggetti è dunque anche nota come content-addressable storage, poiché gli oggetti possono essere recuperati in base al loro contenuto. Non esiste un formato predefinito per i contenuti, quindi ciò che il sistema memorizza sono dati non strutturati.

Anche se l'archiviazione di oggetti non è comune nei computer general-purpose, enormi quantità di dati sono memorizzati in archivi di oggetti, inclusi i contenuti di ricerca su Internet di Google, i contenuti Dropbox, i brani di Spotify e le foto di Facebook. Il cloud

computing (come Amazon aws) generalmente utilizza gli archivi di oggetti (in Amazon S3) per conservare i file system e gli oggetti per le applicazioni dei clienti in esecuzione su computer nel cloud.

Per conoscere la storia degli archivi di oggetti si faccia riferimento a
http://www.theregister.co.uk/2016/07/15/the_history_boys_cas_and_object_storage_map.

CAPITOLO 12

Sistemi di I/O

I due compiti principali di un calcolatore sono l'i/o e l'elaborazione. In molti casi il compito principale è costituito dall'i/o mentre l'elaborazione è semplicemente accessoria: quando si consulta una pagina web, o quando si modifica un file, ciò che più direttamente interessa l'utente è la lettura o l'immissione di informazioni, non l'elaborazione di qualche risposta.

Il ruolo di un sistema operativo nell'i/o di un calcolatore è quello di gestire e controllare le operazioni e i dispositivi di i/o. Sebbene in altri capitoli compaiano argomenti collegati, in questo capitolo sono raccolti tutti gli elementi utili alla composizione di un quadro d'insieme dell'i/o. Poiché il genere delle interfacce hardware stabilisce i requisiti che le funzioni interne del sistema operativo devono possedere, si descrivono innanzitutto i fondamenti dell'hardware di i/o. Quindi si discutono i servizi di i/o che il sistema operativo fornisce e come questi sono inclusi nell'interfaccia di i/o per le applicazioni; inoltre si spiega come il sistema operativo colmi il divario tra le interfacce hardware e quelle per le applicazioni. Si prende in esame anche il meccanismo streams di unix System V, che consente a un'applicazione di comporre dinamicamente catene di codice di driver. Infine, si trattano gli aspetti riguardanti le prestazioni dell'i/o e i principi di progettazione dei sistemi operativi utili al miglioramento delle prestazioni dell'i/o.

12.1 Introduzione

Il controllo dei dispositivi connessi a un calcolatore è una delle questioni più importanti che riguardano i progettisti di sistemi operativi. Poiché i dispositivi di i/o sono così largamente diversi per funzioni e velocità (si considerino per esempio un mouse, un disco, un'unità flash e un archivio di nastri), altrettanto diversi devono essere i metodi di controllo. Tali metodi costituiscono il *sottosistema di i/o* del kernel; questo sottosistema separa il resto del kernel dalla complessità di gestione dei dispositivi di i/o.

La tecnologia dei dispositivi di i/o mostra due tendenze tra loro in conflitto. Da una parte, si osserva la crescente uniformazione a standard delle interfacce fisiche e logiche, e ciò semplifica l'introduzione nei calcolatori e nei sistemi operativi già esistenti di più avanzate generazioni di dispositivi. D'altra parte, però, si assiste a una crescente varietà di dispositivi di i/o; alcuni di loro sono tanto diversi dai dispositivi precedenti da rendere molto difficile il compito di integrarli nei calcolatori e nei sistemi operativi esistenti. Questo problema si affronta con una combinazione di tecniche hardware e software. Gli elementi di base dell'hardware di i/o – porte, bus e controllori di dispositivi – si possono connettere con un'ampia varietà di dispositivi di i/o. Il kernel del sistema operativo è strutturato in moduli di driver dei dispositivi allo scopo di incapsulare i dettagli e le particolarità dei diversi dispositivi. I driver dei dispositivi offrono al sottosistema di i/o un'interfaccia uniforme per l'accesso ai dispositivi, così come le chiamate di sistema forniscono un'interfaccia uniforme tra le applicazioni e il sistema operativo.

12.2 Hardware di I/O

I calcolatori fanno funzionare un gran numero di tipi di dispositivi. La maggior parte rientra nella categoria dei dispositivi di memorizzazione (dischi, nastri), dispositivi di trasmissione (connessioni di rete, Bluetooth), interfacce uomo-macchina (schermi, tastiere, mouse, ingressi e uscite audio). Altri dispositivi sono più specializzati, come i dispositivi di pilotaggio di un caccia a reazione. In questi velivoli il pilota interagisce col calcolatore di bordo tramite la *cloche* e la pedaliera, il calcolatore invia comandi per l'attivazione dei motori che azionano timoni, *flap* e propulsori. Nonostante l'incredibile varietà dei dispositivi di i/o, bastano alcuni concetti per capire come siano connessi e come il sistema operativo li controlli.

Un dispositivo comunica con un sistema elaborativo inviando segnali attraverso un cavo o attraverso l'etere e comunica con il calcolatore tramite un punto di connessione (porta), per esempio una porta seriale. Se più dispositivi condividono un insieme di fili, la connessione è detta *bus*. Un bus è un insieme di fili e un protocollo rigorosamente definito che specifica l'insieme dei messaggi che si possono inviare attraverso i fili. In termini elettronici, i messaggi si inviano tramite configurazioni di livelli di tensione elettrica applicate ai fili con una definita scansione temporale. Quando un dispositivo *A* ha un cavo che si connette a un dispositivo *B* e il dispositivo *B* ha un cavo che si connette a un dispositivo *C* che a sua volta è collegato a una porta di un calcolatore, si ottiene il cosiddetto collegamento in daisy chain, che di solito funziona come un bus.

I bus sono ampiamente usati nell'architettura dei calcolatori e differiscono tra loro per formato dei segnali, velocità, throughput e metodo di connessione. La Figura 12.1 mostra una tipica struttura di bus di pc; si tratta di un bus pcie (il comune bus di sistema dei pc) che connette il sottosistema cpu-memoria ai dispositivi veloci, e di un bus d'espansione cui si connettono i dispositivi relativamente lenti come la tastiera e le porte seriali e usb. Nella parte inferiore sinistra della figura quattro dischi sono collegati a un bus sas (*serial-attached scsi*) inserito nel relativo controllore. Il bus pcie è un bus flessibile che invia i dati su uno o più canali, ciascuno composto da due coppie di fili, una coppia per la ricezione dei dati e l'altra per la trasmissione. Ogni canale è quindi formato da quattro fili e viene utilizzata come flusso di byte full duplex, che trasporta pacchetti di dati a gruppi di otto bit contemporaneamente in entrambe le direzioni. Fisicamente, i collegamenti pcie possono contenere 1, 2, 4, 8, 12, 16 o 32 canali e il numero di canali viene indicato col prefisso "x". Per esempio, una scheda o un connettore pcie che utilizza 8 canali viene indicato con x8. Inoltre, il bus pcie ha attraversato più "generazioni", e nuove generazioni del bus arriveranno in futuro. Quindi, per esempio, una scheda potrebbe essere denominata "pcie gen3 x8", per indicare che funziona con la generazione 3 di pcie e utilizza 8 canali. Un dispositivo del genere ha un throughput massimo di 8 gigabyte al secondo. I dettagli su pcie sono disponibili su <https://pcisig.com>.

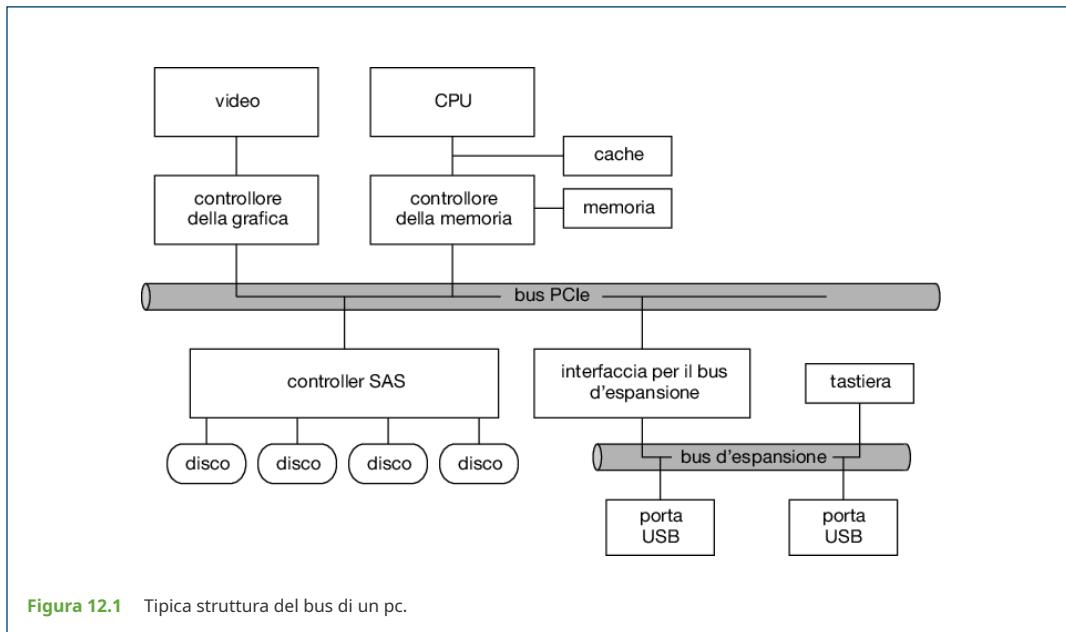


Figura 12.1 Tipica struttura del bus di un pc.

Un controllore è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo. Un controllore di porta seriale è un semplice controllore di dispositivo; si tratta di un singolo circuito integrato (o di una sua parte) nel calcolatore che controlla i segnali presenti nei fili della porta seriale. Per contro un controllore fc (*fibre channel*) non è semplice; poiché il protocollo fc è complesso e utilizzato nei data center anziché nei pc, il controllore del bus fc è spesso realizzato come una scheda hardware separata o come un adattatore bus-host (hba, *host bus adapter*) che si collega a un bus nel computer. Esso contiene generalmente un'unità d'elaborazione, microcodice, e memoria privata che gli consentono di elaborare i messaggi del protocollo fc. Alcuni dispositivi sono dotati di propri controllori incorporati. Osservando un'unità a disco si vede, da un lato, una scheda elettronica a essa agganciata; si tratta del controllore che attua la parte lato disco del protocollo di qualche tipo di connessione, per esempio sas o sata

(*serial advanced technology attachment*). Ha un'unità d'elaborazione e microcodice per l'esecuzione di molti compiti, come localizzazione dei settori difettosi, prelievo anticipato (*prefetching*), gestione del buffer e della cache.

12.2.1 I/O memory mapped

L'unità d'elaborazione fornisce comandi e dati al controllore per portare a termine trasferimenti di i/o tramite uno o più registri per dati e segnali di controllo. La comunicazione con il controllore avviene attraverso la lettura e la scrittura, da parte dell'unità d'elaborazione, di configurazioni di bit in questi registri. Un modo in cui questa comunicazione può avvenire è tramite l'uso di speciali istruzioni di i/o che specificano il trasferimento di un byte o una parola a un indirizzo di porta di i/o. L'istruzione di i/o attiva le linee di bus per selezionare il giusto dispositivo e trasferire bit dentro o fuori dal registro di dispositivo. In alternativa, il controllore di dispositivo può supportare l'i/o memory mapped (*mappato in memoria*). In questo caso i registri di controllo del dispositivo sono mappati in un sottoinsieme dello spazio d'indirizzi della cpu, che esegue le richieste di i/o usando le ordinarie istruzioni di trasferimento di dati per leggere e scrivere i registri di controllo del dispositivo alle locazioni di memoria fisica in cui sono mappati.

In passato, i pc usavano spesso istruzioni di i/o per controllare alcuni dispositivi e l'i/o memory mapped per controllarne altri. Nella Figura 12.2 sono riportati gli usuali indirizzi delle porte di i/o dei pc. Il controllore della grafica utilizza alcune porte di i/o per le operazioni di controllo di base, ma dispone di un'ampia regione mappata in memoria che serve a mantenere i contenuti dello schermo. Un thread scrive sullo schermo inserendo i dati nella regione mappata in memoria; il controllore genera l'immagine dello schermo sulla base del contenuto di questa regione di memoria. Questa tecnica è semplice da usare; inoltre la scrittura di milioni di byte nella memoria grafica è più veloce dell'invio di milioni di istruzioni di i/o. Per questa ragione, nel tempo, i sistemi si sono spostati verso l'i/o memory mapped e oggi la maggior parte dell'i/o viene eseguita da controllori dei dispositivi che utilizzano questa tecnica.

indirizzi per l'I/O (in esadecimale)	dispositivo
000-00F	controllore DMA
020-021	controllore delle interruzioni
040-043	timer
200-20F	controllore dei giochi
2F8-2FF	porta seriale (secondaria)
320-32F	controllore del disco
378-37F	porta parallela
3D0-3DF	controllore della grafica
3F0-3F7	controllore dell'unità a dischetti
3F8-3FF	porta seriale (principale)

Figura 12.2 Indirizzi delle porte dei dispositivi di i/o nei pc (elenco parziale).

Il controllo di un dispositivo di i/o consiste in genere di quattro registri: `status`, `control`, `data-in` e `data-out`.

- La cpu legge dal registro `data-in` per ricevere dati.
- La cpu scrive nel registro `data-out` per emettere dati.
- Il registro `status` contiene alcuni bit che possono essere letti dalla cpu e indicano lo stato della porta; per esempio indicano se è stata portata a termine l'esecuzione del comando corrente, se un byte è disponibile per essere letto dal registro `data-in`, se si è verificato un errore del dispositivo.
- Il registro `control` può essere scritto per attivare un comando o per cambiare il modo di funzionamento del dispositivo. Per esempio, un certo bit nel registro `control` della porta seriale determina il tipo di comunicazione tra *half-duplex* e *full-duplex*, un altro abilita il controllo di parità, un terzo imposta la lunghezza delle parole a 7 o 8 bit, altri selezionano una tra le velocità che la porta seriale può sostenere.

La tipica dimensione dei registri di dati varia tra 1 e 4 byte. Certi controllori hanno circuiti integrati fifo che possono contenere parecchi byte per l'invio e la ricezione di dati, in modo da espandere la capacità del controllore oltre la dimensione del registro di dati. Un circuito integrato fifo può contenere una piccola sequenza di dati finché il dispositivo o la cpu non sono in grado di riceverli.

12.2.2 Polling

Il protocollo completo per l'interazione fra la cpu e un controllore può essere intricato, ma la fondamentale nozione di handshaking (*negoziazione*) è semplice, ed è illustrata con un esempio. Si assuma l'uso di due bit per coordinare la relazione di tipo produttore-consumatore fra il controllore e la cpu. Il controllore specifica il suo stato per mezzo del bit `busy` del registro `status`: pone a 1 il bit `busy` quando è impegnato in un'operazione, e lo pone a 0 quando è pronto a eseguire il comando successivo. La cpu comunica le sue

richieste tramite il bit `command-ready` nel registro `command`: pone questo bit a 1 quando il controllore deve eseguire un comando. In questo esempio, la cpu scrive in una porta coordinandosi con un controllore per mezzo del seguente handshaking.

1. La cpu legge ripetutamente il bit `busy` finché questo non vale 0.
2. La cpu pone a 1 il bit `write` del registro dei comandi e scrive un byte nel registro `data-out`.
3. La cpu pone a 1 il bit `command-ready`.
4. Quando il controllore si accorge che il bit `command-ready` è posto a 1, pone a 1 il bit `busy`.
5. Il controllore legge il registro dei comandi e trova il comando `write`; legge il registro `data-out` per ottenere il byte da scrivere, e compie l'operazione di i/o sul dispositivo.
6. Il controllore pone a 0 il bit `command-ready`, pone a 0 il bit `error` nel registro `status` per indicare che l'operazione di i/o ha avuto esito positivo, e pone a 0 il bit `busy` per indicare che l'operazione è terminata.

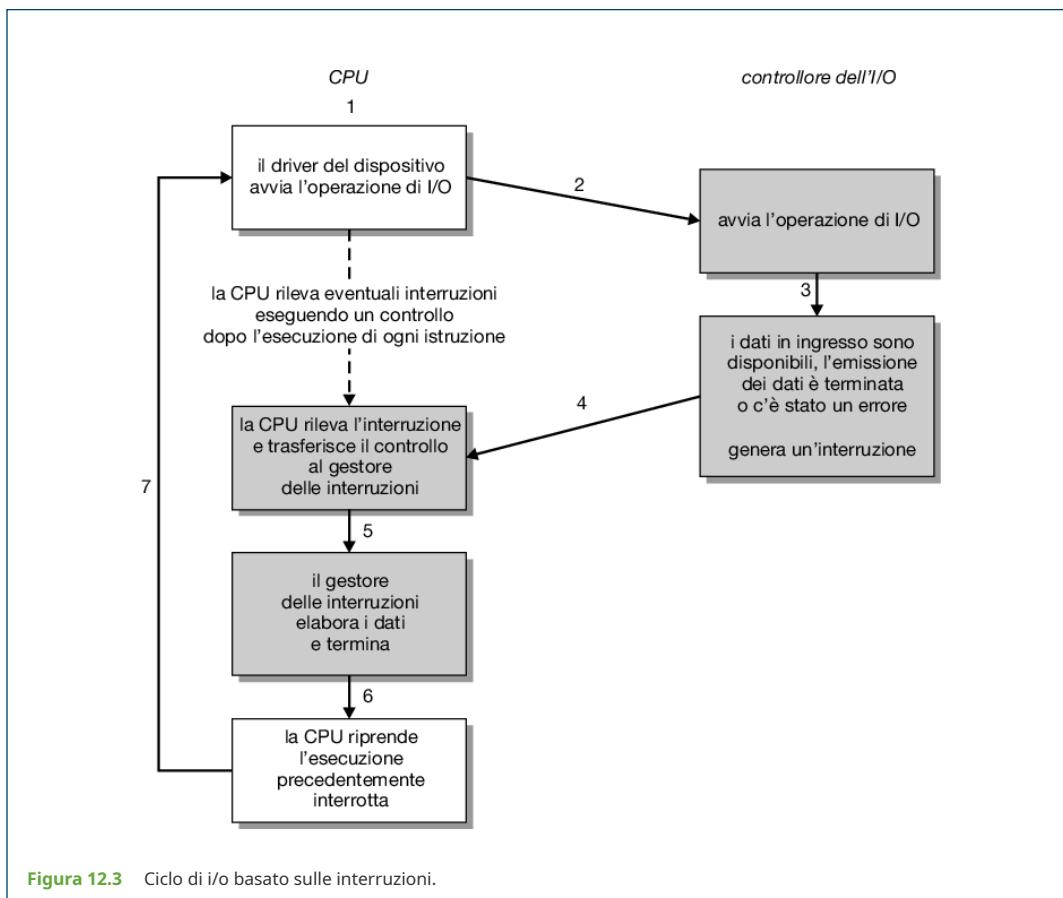
La sequenza appena descritta si ripete per ogni byte.

Durante l'esecuzione del passo 1, la cpu è in attesa attiva (*busy-waiting*) o in interrogazione ciclica (*polling*): itera la lettura del registro `status` finché il bit `busy` assume il valore 0. Se il controllore e il dispositivo sono veloci, questo metodo è ragionevole, ma se l'attesa rischia di prolungarsi, sarebbe probabilmente meglio se la cpu si dedicasse a un'altra operazione. In questo caso si pone il problema di come la cpu possa sapere quando il controllore è tornato libero. È necessario che la cpu serva certi tipi di dispositivi rapidamente, o si potrebbero perdere alcuni dati. Quando, per esempio, i dati affluiscono in una porta seriale o dalla tastiera, il piccolo buffer del controllore diverrà presto pieno, e se la cpu attende troppo a lungo prima di riprendere la lettura dei byte, si perderanno informazioni.

In molte architetture di calcolatori sono sufficienti tre istruzioni della cpu per effettuare il polling di un dispositivo: `read`, lettura di un registro del dispositivo; `logical-and`, usata per estrarre il valore di un bit di stato, e `branch`, salto a un altro punto del codice se l'argomento è diverso da zero. Chiaramente, il polling è in sé un'operazione efficiente; tale tecnica diviene però inefficiente se le ripetute interrogazioni trovano raramente un dispositivo pronto per il servizio, mentre altre utili elaborazioni attendono la cpu. In tali casi, anziché richiedere alla cpu di eseguire il polling, può essere più efficiente far sì che il controllore comunichi alla cpu che il dispositivo è pronto. Il meccanismo hardware che permette tale comunicazione si chiama interruzione (*interrupt*).

12.2.3 Interruzioni

Il meccanismo di base dell'interruzione funziona come segue. L'hardware della cpu ha un input, detto linea di richiesta dell'interruzione, del quale la cpu controlla lo stato dopo l'esecuzione di ogni istruzione. Quando rileva il segnale di un controllore sulla linea di richiesta dell'interruzione, la cpu salva lo stato corrente e salta alla routine di gestione dell'interruzione (*interrupt-handler routine*), che si trova a un indirizzo prefissato di memoria. Questa procedura determina le cause dell'interruzione, porta a termine l'elaborazione necessaria, ripristina lo stato ed esegue un'istruzione `return from interrupt` per far sì che la cpu ritorni nello stato in cui si trovava prima della sua interruzione. Il controllore del dispositivo genera un'interruzione della cpu sulla linea di richiesta delle interruzioni, che la cpu rileva e recapita al gestore delle interruzioni, che a sua volta gestisce l'interruzione corrispondente servendo il dispositivo. Nella Figura 12.3 è riassunto il ciclo di i/o causato da un'interruzione della cpu. In questo capitolo daremo molta importanza alla gestione delle interruzioni, perché persino un sistema a singola utenza ne gestisce centinaia al secondo, mentre un server ne può gestire centinaia di migliaia.



Il peso dato in questo capitolo alla gestione delle interruzioni è dovuto al fatto che anche i moderni sistemi monoutente gestiscono centinaia di interruzioni al secondo e i server ne gestiscono persino centinaia di migliaia al secondo. Per esempio, la Figura 12.4 mostra l'output del comando `latency` su macos, rivelando che in dieci secondi un computer desktop senza particolari carichi di lavoro ha eseguito quasi 23.000 interrupt.

	SCHEDULER	INTERRUPTS	0:00:10
<hr/>			
total_samples	13	22998	
delays < 10 usecs	12	16243	
delays < 20 usecs	1	5312	
delays < 30 usecs	0	473	
delays < 40 usecs	0	590	
delays < 50 usecs	0	61	
delays < 60 usecs	0	317	
delays < 70 usecs	0	2	
delays < 80 usecs	0	0	
delays < 90 usecs	0	0	
delays < 100 usecs	0	0	
total < 100 usecs	13	22998	

Figura 12.4 Il comando latency di Mac os X.

Il meccanismo di base delle interruzioni che abbiamo appena descritto permette alla cpu di rispondere a un evento asincrono, come quello di un controllore di un dispositivo che divenga pronto per essere servito. Nei sistemi operativi moderni sono necessarie capacità di gestione delle interruzioni più raffinate.

1. Si deve poter posporre la gestione delle interruzioni durante le fasi critiche dell'elaborazione.
2. Si deve disporre di un meccanismo efficiente per passare il controllo all'appropriato gestore delle interruzioni, senza dover esaminare ciclicamente tutti i dispositivi (*polling*) per determinare quale abbia generato l'interruzione.
3. Si deve disporre di più livelli d'interruzione, di modo che il sistema possa distinguere le interruzioni ad alta priorità da quelle a priorità inferiore, servendo le richieste con la celerità appropriata al caso.
4. Abbiamo bisogno di un modo per permettere a un'istruzione di ottenere l'attenzione del sistema operativo direttamente (in maniera distinta rispetto alle richieste di i/o), per attività come la gestione di page fault e per errori come, per esempio, la divisione per zero. Come vedremo, questo ruolo è svolto dalle trap.

In un calcolatore moderno queste caratteristiche sono fornite dalla cpu e dal controllore hardware delle interruzioni.

La maggior parte delle cpu ha due linee di richiesta delle interruzioni. Una è quella delle interruzioni non mascherabili, riservata a eventi quali gli errori di memoria irrecuperabili. La seconda linea è quella delle interruzioni mascherabili: può essere disattivata dalla cpu prima dell'esecuzione di una sequenza critica di istruzioni che non deve essere interrotta. L'interruzione mascherabile è usata dai controllori dei dispositivi per richiedere un servizio.

Il meccanismo delle interruzioni accetta un indirizzo – un numero che seleziona una specifica procedura di gestione delle interruzioni da un insieme ristretto. Nella maggior parte delle architetture questo indirizzo è uno scostamento relativo in una tabella detta vettore delle interruzioni, contenente gli indirizzi di memoria degli specifici gestori delle interruzioni. Lo scopo di un meccanismo vettorizzato di gestione delle interruzioni è di ridurre la necessità che un singolo gestore debba individuare tutte le possibili fonti d'interruzione per determinare quale di loro abbia richiesto un servizio. In pratica, tuttavia, i calcolatori hanno più dispositivi (e quindi, più gestori delle interruzioni) che elementi nel vettore delle interruzioni. Una maniera diffusa di risolvere questo problema consiste nel concatenamento delle interruzioni (*interrupt chaining*), in cui ogni elemento del vettore delle interruzioni punta alla testa di una lista di gestori delle interruzioni. Quando si verifica un'interruzione, si chiamano uno alla volta i gestori nella lista corrispondente finché non se ne trova uno che può soddisfare la richiesta. Questa struttura è un compromesso fra l'overhead di una tabella delle interruzioni enorme e l'inefficienza dell'uso di un solo gestore delle interruzioni.

Nella Figura 12.5 è descritto il vettore delle interruzioni della cpu Intel Pentium. Gli eventi da 0 a 31, non mascherabili, si usano per segnalare varie condizioni d'errore (che sono causa di arresti anomali del sistema), errori di pagina (che richiedono un'azione immediata) e richieste di debug (arresto delle normali operazioni per passare all'esecuzione di un debugger); quelli dal 32 al 255, mascherabili, si usano, per esempio, per le interruzioni generate dai dispositivi.

indice del vettore	descrizione
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19-31	(Intel reserved, do not use)
32-255	maskable interrupts

Figura 12.5 Vettore delle interruzioni della cpu Intel Pentium.

Il meccanismo delle interruzioni realizza anche un sistema di livelli di priorità delle interruzioni. Esso permette alla cpu di differire la gestione delle interruzioni di bassa priorità senza mascherare tutte le interruzioni, e permette a un'interruzione di priorità alta di

sospendere l'esecuzione della procedura di servizio di un'interruzione di priorità bassa.

Un sistema operativo moderno interagisce con il meccanismo delle interruzioni in vari modi. All'accensione della macchina esamina i bus per determinare quali dispositivi siano presenti, e installa gli indirizzi dei corrispondenti gestori delle interruzioni nel vettore delle interruzioni. Durante l'i/o, i vari controllori di dispositivi generano le interruzioni della cpu quando sono pronti per un servizio. Queste interruzioni significano che è stato completato un output, o che sono disponibili dati in ingresso, o che un'operazione non è andata a buon fine. Il meccanismo delle interruzioni si usa anche per gestire un'ampia gamma di eccezioni, come la divisione per 0, l'accesso a indirizzi di memoria protetti o inesistenti o il tentativo di eseguire un'istruzione privilegiata in modalità utente. Gli eventi che producono le interruzioni hanno una proprietà in comune: inducono il sistema operativo a eseguire urgentemente una procedura autonoma.

La gestione delle interruzioni ha in molti casi vincoli di tempo e risorse ed è quindi complicata da implementare. Per questa ragione, i sistemi suddividono spesso la gestione delle interruzioni tra un gestore di primo livello (flih) e un gestore di secondo livello (slih): il primo esegue un cambio di contesto, memorizza lo stato e accoda un'operazione di gestione, mentre il secondo, schedulato separatamente, esegue la gestione dell'operazione richiesta.

Un sistema operativo può fare altri usi proficui di un efficiente meccanismo hardware e software che memorizza una piccola quantità d'informazioni sullo stato della cpu e poi richiama una procedura del kernel. Per esempio, molti sistemi operativi usano il meccanismo delle interruzioni per la gestione della memoria virtuale. Un'eccezione di pagina mancante genera un'interruzione che sospende il processo corrente e trasferisce il controllo dell'esecuzione al relativo gestore nel kernel. Tale gestore memorizza le informazioni sullo stato del processo, lo sposta nella coda d'attesa, compie le necessarie operazioni di gestione della cache delle pagine, avvia un'operazione di i/o per prelevare la pagina giusta, schedula la ripresa dell'esecuzione di un altro processo e ritorna dall'interruzione.

Un altro esempio è dato dall'implementazione delle *chiamate di sistema*. Solitamente i programmi sfruttano routine di libreria per eseguire chiamate di sistema. La routine controlla i parametri passati dall'applicazione, li assembla in una struttura dati appropriata da passare al kernel, e infine esegue una particolare istruzione detta interruzione software o trap. Questa istruzione ha un operando che identifica il servizio del kernel desiderato. Quando un processo esegue l'istruzione di eccezione, l'hardware delle interruzioni salva lo stato del codice utente, passa al modo kernel e recapita l'interruzione alla procedura del kernel che realizza il servizio richiesto. A una trap si assegna una priorità di interruzione relativamente bassa rispetto a quelle date alle interruzioni dei dispositivi – eseguire una chiamata di sistema per conto di un'applicazione è meno urgente di quanto non sia servire un controllore prima che la sua coda fifo trabocchi causando la perdita di informazioni.

Le interruzioni si possono inoltre usare per gestire il flusso di controllo all'interno del kernel. Si consideri un esempio di elaborazione richiesta per completare una lettura da un disco. Un passo necessario è quello di copiare dati dallo spazio del kernel al buffer dell'utente. Questa azione richiede tempo, ma non è urgente, e non dovrebbe bloccare la gestione delle interruzioni con priorità più alta. Un altro passo è quello di avviare il successivo i/o in attesa relativo a quell'unità a disco. Questo passo ha priorità più alta: se le unità a disco si devono usare in modo efficiente, è necessario avviare l'evasione della successiva richiesta di i/o non appena la precedente sia stata soddisfatta. Di conseguenza una coppia di gestori delle interruzioni realizza il codice del kernel che compie le letture dai dischi. Il gestore ad alta priorità mantiene le informazioni sullo stato dell'i/o, risponde al segnale d'interruzione del dispositivo, avvia il prossimo i/o in attesa e genera un'interruzione a bassa priorità per completare il lavoro. Più tardi, in un momento in cui la cpu non è occupata in compiti ad alta priorità, si serve l'interruzione a bassa priorità. Il gestore corrispondente completa l'i/o a livello utente copiando i dati dal buffer del kernel a quello dell'applicazione, e richiamando poi lo scheduler per aggiungere l'applicazione alla coda dei processi pronti.

Un'architettura del kernel basata su thread è adatta alla realizzazione di più livelli di priorità delle interruzioni, e a dare la precedenza alla gestione delle interruzioni rispetto alle elaborazioni in background delle procedure del kernel e delle applicazioni. Questo concetto si può esemplificare considerando il kernel del sistema operativo Solaris. In questo sistema i gestori delle interruzioni si eseguono come thread del kernel cui si riservano valori elevati di priorità. Tali priorità garantiscono la precedenza dei gestori delle interruzioni rispetto al codice delle applicazioni e al lavoro ordinario del kernel, e inoltre realizzano le necessarie relazioni di priorità fra i diversi gestori delle interruzioni. Le priorità fanno sì che lo scheduler dei thread di Solaris sospenda i gestori delle interruzioni di bassa priorità a vantaggio di quelli di priorità più alta, e la realizzazione basata su thread permette alle architetture multiprocessore di eseguire parallelamente diversi gestori delle interruzioni. L'architettura delle interruzioni dei sistemi Linux è descritta nel Capitolo 20, di Windows 10 nel Capitolo 21 e di unix nell'Appendice C (reperibili sulla piattaforma MyLab).

Riassumendo, le interruzioni sono usate diffusamente dai sistemi operativi moderni per gestire eventi asincroni e per eseguire procedure in modalità supervisore nel kernel. Per far sì che i compiti più urgenti siano portati a termine per primi, i calcolatori moderni usano un sistema di priorità delle interruzioni. I controllori dei dispositivi, i guasti hardware e le chiamate di sistema generano interruzioni al fine di innescare l'esecuzione di procedure del kernel. Poiché le interruzioni sono usate in modo massiccio per affrontare situazioni in cui il tempo è un fattore critico, è necessario avere un'efficiente gestione delle interruzioni per ottenere buone prestazioni del sistema.

12.2.4 Accesso diretto alla memoria (DMA)

Quando un dispositivo compie trasferimenti di grandi quantità di dati, come nel caso di un'unità a disco, l'uso di una costosa cpu per il controllo dei bit di stato e per la scrittura di dati nel registro del controllore un byte alla volta, detto i/o programmato (*programmed i/o*, pio), sembra essere uno spreco. In molti calcolatori si evita di sovraccaricare la cpu assegnando una parte di questi compiti a un processore specializzato, detto controllore dell'accesso diretto alla memoria (*direct memory-access*, dma). Per dar avvio a un trasferimento dma, la cpu scrive in memoria un blocco di comando per il dma. Esso contiene un puntatore alla locazione dei dati da trasferire, un altro puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La cpu scrive l'indirizzo di questo blocco di comando nel controllore del dma, e prosegue con altre attività. Il controllore dma agisce quindi direttamente sul bus della memoria, presentando al bus gli indirizzi di memoria necessari per eseguire il trasferimento senza l'aiuto della cpu. Un semplice controllore dma è un componente standard in tutti i sistemi moderni, dagli smartphone ai mainframe.

È più ragionevole che l'indirizzo di destinazione debba trovarsi nello spazio di indirizzamento del kernel. Se fosse nello spazio utente l'utente potrebbe, per esempio, modificare il contenuto di quello spazio durante il trasferimento, perdendo alcuni dati. Per ottenere il

trasferimento dei dati dma nello spazio utente per l'accesso da parte dei thread, tuttavia, è necessaria una seconda operazione di copia, questa volta dalla memoria del kernel alla memoria dell'utente. Questo doppio buffering è inefficiente e per tale ragione, nel corso del tempo, i sistemi operativi sono passati all'utilizzo della mappatura della memoria (si veda il Paragrafo 12.2.1) per eseguire trasferimenti i/o direttamente tra i dispositivi e lo spazio utente.

L'handshaking tra il controllore del dma e il controllore del dispositivo si svolge grazie a una coppia di fili detti dma-request e dma-acknowledge. Il controllore del dispositivo manda un segnale sulla linea dma-request quando una word di dati è disponibile per il trasferimento. Questo segnale fa sì che il controllore dma prenda possesso del bus di memoria, presenti l'indirizzo desiderato ai fili d'indirizzamento della memoria e mandi un segnale lungo la linea dma-acknowledge. Quando il controllore del dispositivo riceve questo segnale, trasferisce in memoria la word di dati e rimuove il segnale dalla linea dma-request.

Quando l'intero trasferimento termina, il controllore del dma interrompe la cpu. Nella Figura 12.6 è rappresentato questo processo. Quando il controllore del dma prende possesso del bus di memoria, la cpu è temporaneamente impossibilitata ad accedere alla memoria centrale, sebbene abbia accesso ai dati contenuti nella sua cache primaria e secondaria. Questo fenomeno, noto come sottrazione di cicli, può rallentare le computazioni della cpu; ciononostante l'assegnamento del lavoro di trasferimento di dati a un controllore dma migliora in generale le prestazioni complessive del sistema. In alcune architetture per realizzare la tecnica dma si usano gli indirizzi della memoria fisica, mentre in altre s'impiega l'accesso diretto alla memoria virtuale (*direct virtual-memory access*, dvma); in questo caso si usano indirizzi virtuali che poi si traducono in indirizzi fisici. La tecnica dvma permette di compiere i trasferimenti di dati tra due dispositivi che eseguono i/o memory mapped senza far intervenire la cpu o accedere alla memoria centrale.

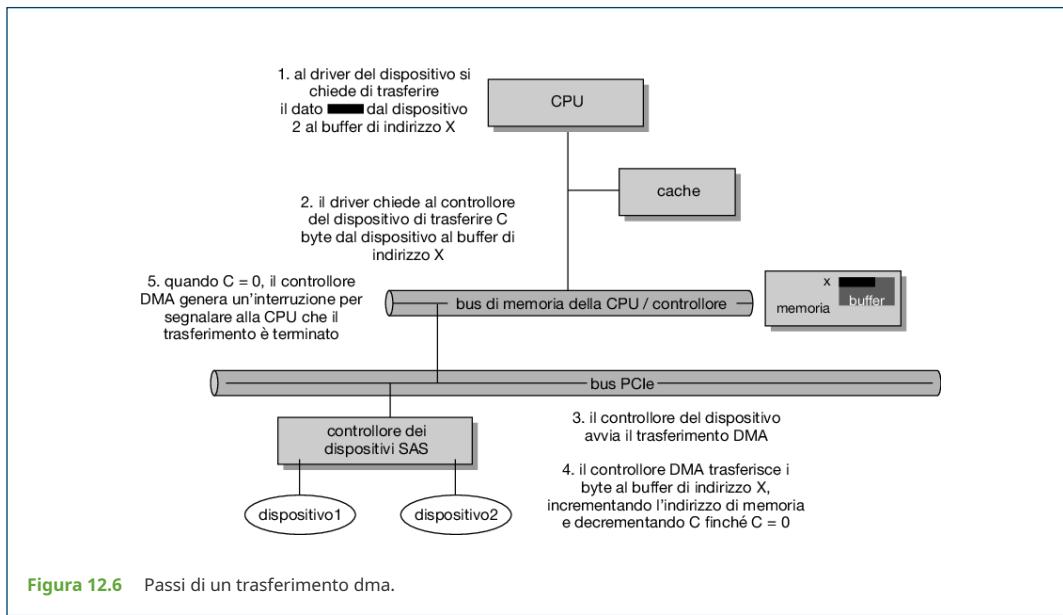


Figura 12.6 Passi di un trasferimento dma.

Nei kernel che operano in modalità protetta, in genere il sistema operativo non permette ai processi di impartire direttamente comandi ai dispositivi. Ciò protegge i dati dalle violazioni dei controlli d'accesso e il sistema da un eventuale uso scorretto dei controllori dei dispositivi che potrebbe portare a una caduta del sistema stesso. Il sistema operativo invece esporta delle funzioni di i/o che possono essere eseguite da processi sufficientemente privilegiati per effettuare operazioni di basso livello sull'hardware sottostante. Quando invece il kernel non può garantire la protezione della memoria, i processi hanno accesso diretto ai controllori dei dispositivi. Questo accesso diretto si può utilizzare in modo da ottenere buone prestazioni, perché evita la comunicazione col kernel, i cambi di contesto e l'interazione fra diversi livelli del kernel. Purtroppo interferisce con la stabilità e la sicurezza del sistema. La tendenza comune per i sistemi operativi d'uso generale è quella di proteggere la memoria e i dispositivi in modo da salvaguardarli da applicazioni accidentali o volutamente dannose.

12.2.5 Concetti principali dell'hardware di I/O

Sebbene gli aspetti dell'i/o che riguardano i dispositivi siano complessi se si analizzano tanto dettagliatamente quanto farebbe un progettista elettronico, i concetti appena descritti sono sufficienti per comprendere molti aspetti dell'i/o per ciò che concerne i sistemi operativi. Ecco un sommario dei concetti principali:

- bus;
- controllore;
- porta di i/o e suoi registri;
- procedura di handshaking tra la cpu e il controllore di un dispositivo;
- esecuzione dell'handshaking per mezzo del polling o delle interruzioni;
- delega dell'i/o a un controllore dma nel caso di trasferimenti di grandi quantità di dati.

Precedentemente in questo paragrafo è stato fornito un esempio dell'*handshaking* che avviene tra un controllore di dispositivo e un host. In realtà, la grande varietà di dispositivi esistenti pone un problema a chi voglia realizzare concretamente un sistema operativo. Ogni tipo di dispositivo ha proprie funzionalità, proprie definizioni dei bit di controllo, e un proprio protocollo per l'interazione con la macchina – e tutto ciò varia da dispositivo a dispositivo. Come deve essere progettato un sistema operativo affinché sia possibile collegare al calcolatore nuovi dispositivi senza che sia necessario riscrivere il sistema operativo stesso? E inoltre, vista la grande varietà di dispositivi, come può il sistema operativo fornire alle applicazioni un'interfaccia per l'i/o uniforme ed efficace? Discuteremo questi aspetti nel seguito.

12.3 Interfaccia di I/O delle applicazioni

In questo paragrafo si discutono le tecniche e le interfacce di un sistema operativo che permettono un trattamento standardizzato e uniforme dei dispositivi di i/o. Si spiega, per esempio, come un'applicazione possa aprire un file residente in un disco senza sapere di che tipo di disco si tratti, e come si possano aggiungere al calcolatore nuove unità a disco e altri dispositivi senza che si debba modificare il sistema operativo.

I metodi qui esposti coinvolgono l'astrazione, l'incapsulamento e la stratificazione del software. In particolare, si può effettuare un'astrazione rispetto ai dettagli delle differenze tra i dispositivi per l'i/o identificandone alcuni tipi generali. A ognuno di questi tipi si accede per mezzo di un insieme standardizzato di funzioni – un'interfaccia. Le differenze sono incapsulate in moduli del kernel detti driver dei dispositivi, specializzati internamente per gli specifici dispositivi, ma che comunicano con l'esterno per mezzo delle interfacce uniformi. Nella Figura 12.7 è illustrata la divisione in strati software di quelle parti del kernel che riguardano la gestione dell'i/o.

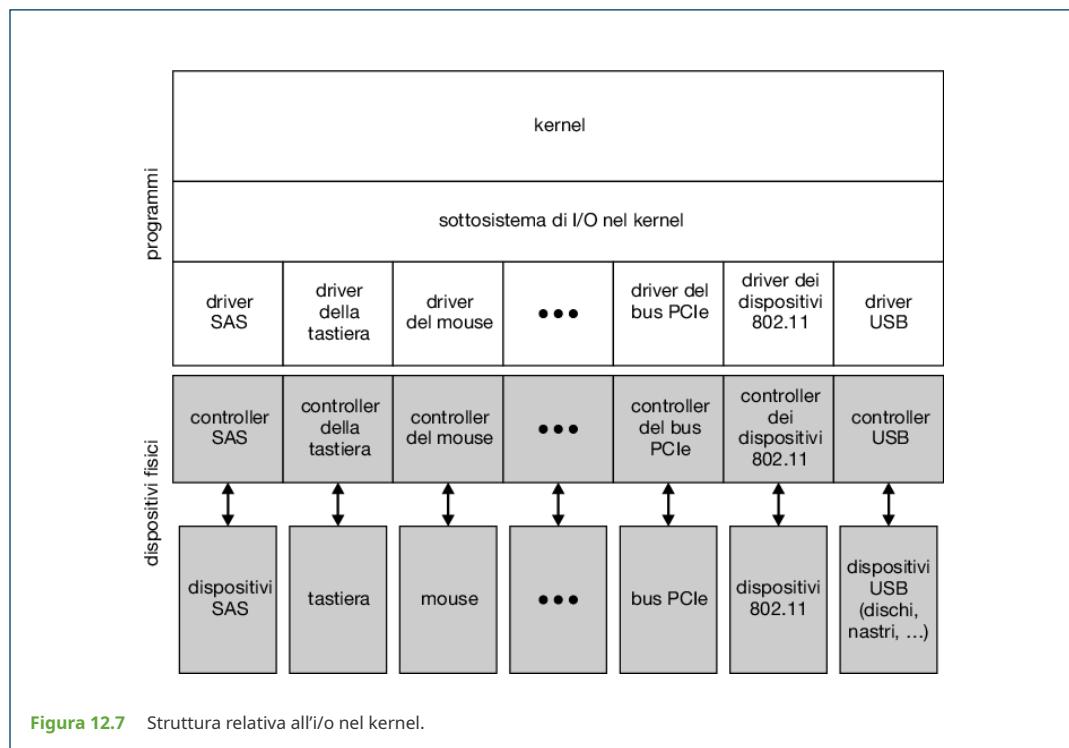


Figura 12.7 Struttura relativa all'i/o nel kernel.

Lo scopo dello strato dei driver dei dispositivi è di nascondere al sottosistema di i/o del kernel le differenze fra i controllori dei dispositivi, in modo simile a quello con cui le chiamate di sistema di i/o incapsulano il comportamento dei dispositivi in alcune classi generiche che nascondono le differenze hardware alle applicazioni. Il fatto che così il sottosistema di i/o sia reso indipendente dall'hardware semplifica il lavoro di chi sviluppa il sistema operativo, e va inoltre a vantaggio dei costruttori dei dispositivi. Questi, infatti, o progettano i nuovi dispositivi in modo tale che siano compatibili con un'interfaccia host-controllore già esistente (per esempio sata), oppure scrivono driver che permettano ai nuovi dispositivi di essere gestiti dai sistemi operativi più diffusi. In questo modo, nuovi dispositivi sono utilizzabili da un calcolatore senza che occorra attendere lo sviluppo del codice di supporto da parte del produttore del sistema operativo.

Sfortunatamente per i produttori di dispositivi, ogni tipo di sistema operativo ha le sue convenzioni riguardanti l'interfaccia dei driver dei dispositivi. Così, un dato dispositivo potrà essere venduto con molti driver diversi – per esempio, driver per Windows, Linux, aix e macos. I dispositivi (Figura 12.8) possono differire in molti aspetti.

aspetto	variazione	esempio
modalità di trasferimento dei dati	a caratteri a blocchi	terminale unità a disco
modalità d'accesso	sequenziale casuale	modem lettore di CD-ROM
prevedibilità dell'I/O	sincrono asincrono	unità a nastro tastiera
condivisione	dedicato condiviso	unità a nastro tastiera
velocità	latenza tempo di ricerca velocità di trasferimento attesa fra le operazioni	
direzione dell'I/O	solo lettura solo scrittura lettura e scrittura	lettore di CD-ROM controllore della grafica unità a disco

Figura 12.8 Caratteristiche dei dispositivi per l'i/o.

- Trasferimento a flusso di caratteri o a blocchi. Un dispositivo a flusso di caratteri trasferisce dati un byte alla volta, mentre uno a blocchi trasferisce un blocco di byte in un'unica soluzione.
- Sequenziale o accesso diretto. Un dispositivo sequenziale trasferisce dati secondo un ordine fisso dipendente dal dispositivo, mentre l'utente di un dispositivo ad accesso diretto può richiedere l'accesso a una qualunque delle possibili locazioni di memorizzazione.
- Dispositivi sincroni o asincroni. Un dispositivo sincrono trasferisce dati con un tempo di risposta prevedibile, in maniera coordinata rispetto al resto del sistema. Un dispositivo asincrono ha tempi di risposta irregolari o non prevedibili, non coordinati con gli altri eventi del computer.
- Condivisibili o dedicati. Un dispositivo condivisibile può essere usato in modo concorrente da diversi processi o thread, mentre ciò è impossibile se il dispositivo è dedicato.
- Velocità di funzionamento. Può variare da alcuni byte al secondo fino a qualche gigabyte al secondo.
- Lettura e scrittura, sola lettura o sola scrittura. Alcuni dispositivi possono emettere e ricevere dati, ma altri possono trasferire dati in una sola direzione.

Per ciò che riguarda l'accesso delle applicazioni ai dispositivi, molte di queste differenze sono nascoste dal sistema operativo, e i dispositivi sono raggruppati in poche classi convenzionali. Si è riscontrato che i modi d'accesso ai dispositivi che ne risultano sono utili e largamente applicabili. Anche se la forma precisa delle chiamate di sistema può variare nei diversi sistemi operativi, le classi di dispositivi sono abbastanza regolari. Le convenzioni d'accesso principali includono l'i/o a blocchi, l'i/o a flusso di caratteri, l'accesso ai file mappati in memoria, e le socket di rete. I sistemi operativi forniscono anche chiamate di sistema speciali per l'accesso a qualche dispositivo aggiuntivo, per esempio un orologio o un timer. Qualche sistema operativo mette a disposizione un insieme di chiamate di sistema per display grafici, video e audio.

La maggior parte dei sistemi ha anche una "via di fuga" (*escape*) o *back door* che permette il passaggio trasparente di comandi arbitrari da un'applicazione a un driver di dispositivo. In unix tale funzione è svolta dalla chiamata di sistema `ioctl()` (che sta per *i/o control*) e consente a un'applicazione di impiegare qualsiasi funzionalità fornita da qualsiasi driver di dispositivo, senza che per questo sia necessario creare nuove chiamate di sistema. Gli argomenti di `ioctl()` sono tre: il primo è un descrittore di file che collega l'applicazione al driver desiderato facendo riferimento a un dispositivo gestito da quel driver; il secondo è un numero intero che seleziona uno dei comandi forniti dal driver; il terzo argomento, infine, è un puntatore a un'arbitraria struttura dati in memoria, tramite la quale l'applicazione e il driver si scambiano le informazioni di controllo o i dati necessari.

L'identificativo del dispositivo in unix e Linux è una coppia formata da un major number e un minor number. Il major number rappresenta il tipo di dispositivo, mentre il minor number è l'istanza di quel dispositivo. Per esempio, considerate un sistema dotato di alcuni dispositivi ssd. Se si immette il comando:

```
% ls -l /dev/sda*
```

si ottiene il seguente output:

```
brw-rw---- 1 root disk 8, 0 Mar 16 09:18 /dev/sda
```

```
brw-rw---- 1 root disk 8, 1 Mar 16 09:18 /dev/sda1  
brw-rw---- 1 root disk 8, 2 Mar 16 09:18 /dev/sda2  
brw-rw---- 1 root disk 8, 3 Mar 16 09:18 /dev/sda3
```

Il numero 8 è il major number del dispositivo. Il sistema operativo utilizza questa informazione per indirizzare le richieste di i/o al driver di periferica appropriato. I minor number 0, 1, 2 e 3 indicano l'istanza del dispositivo e consentono alle richieste di i/o di selezionare il corretto dispositivo per indirizzare la richiesta.

12.3.1 Dispositivi con trasferimento a blocchi e a caratteri

L'interfaccia per i dispositivi a blocchi sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati. Ci si aspetta che il dispositivo comprenda istruzioni come `read()` e `write()` e, nel caso sia ad accesso casuale, anche un comando `seek()` per specificare il blocco successivo da trasferire. Di solito le applicazioni comunicano con questi dispositivi tramite un'interfaccia di file system. Si vede che `read()`, `write()` e `seek()` catturano l'essenza del comportamento dei dispositivi con trasferimento a blocchi, in modo che le applicazioni non vedano le differenze di basso livello fra questi dispositivi.

Il sistema operativo e certe applicazioni particolari come quelle per la gestione delle basi di dati possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo caso si parla di i/o a basso livello (*raw i/o*). L'uso del file system da parte delle applicazioni che gestiscono già in proprio un buffer per l'i/o comporta l'inutile intervento di un buffer aggiuntivo. Analogamente, l'uso dei lock da parte del sistema operativo nei confronti di applicazioni che già implementano meccanismi per la mutua esclusione relativi ai blocchi dei file risulta ridondante, o peggio contraddittorio. Per superare tali potenziali conflitti, l'i/o a basso livello passa il controllo del dispositivo direttamente all'applicazione, togliendo così di mezzo il sistema operativo. Purtroppo, ciò significa anche che non risulta disponibile sul dispositivo in questione alcun servizio del sistema operativo. Un compromesso che si sta diffondendo sempre più è fornire una modalità d'accesso ai file che disabiliti il buffer e i meccanismi di gestione dei lock; nel mondo unix si parla di i/o diretto.

L'accesso ai file mappato in memoria può costituire uno strato software sopra i driver dei dispositivi a blocchi. Piuttosto che offrire funzioni di lettura e scrittura, un'interfaccia di mappaggio in memoria fornisce la possibilità di accedere a un'unità a disco tramite un vettore di byte della memoria centrale. La chiamata di sistema che associa un file a una regione di memoria restituisce l'indirizzo di memoria virtuale di una copia del file. Gli effettivi trasferimenti di dati sono eseguiti solo quando necessari per soddisfare una richiesta d'accesso all'immagine in memoria. Poiché i trasferimenti si trattano nello stesso modo in cui si gestisce l'accesso su richiesta a una pagina di memoria virtuale, l'i/o memory mapped è efficiente. Esso è inoltre conveniente per i programmati perché l'accesso a un file memory mapped è semplice tanto quanto la lettura e la scrittura in memoria. I sistemi operativi che gestiscono la memoria virtuale utilizzano comunemente l'interfaccia di mappaggio in memoria per i servizi del kernel. Per esempio, quando il sistema operativo deve eseguire un programma, mappa l'eseguibile in memoria, quindi trasferisce il controllo all'indirizzo iniziale. Questo tipo d'interfaccia è spesso usato anche per l'accesso del kernel all'area di swapping nei dischi.

La tastiera è un esempio di dispositivo al quale si accede tramite un'interfaccia a flusso di caratteri. Le chiamate di sistema fondamentali per le interfacce di questo tipo permettono a un'applicazione di acquisire (`get()`) o inviare (`put()`) un carattere. Basandosi su quest'interfaccia è possibile costruire librerie che offrono l'accesso riga per riga, con buffering ed editing (per esempio, quando l'utente preme il tasto backspace, si rimuove il carattere precedente dalla sequenza di caratteri da inserire). Questo tipo d'accesso è conveniente per dispositivi come tastiere, mouse, modem, che producono dati "spontaneamente", cioè in momenti che non possono essere sempre previsti dalle applicazioni. Lo stesso tipo d'accesso è adatto anche ai dispositivi che emettono dati organizzati in modo naturale come sequenza lineare di byte, per esempio le stampanti o le schede audio.

12.3.2 Dispositivi di rete

Poiché i modi di indirizzamento e le prestazioni tipiche dell'i/o di rete sono notevolmente differenti da quelli dell'i/o delle unità a disco, la maggior parte dei sistemi operativi fornisce un'interfaccia per l'i/o di rete diversa da quella caratterizzata dalle operazioni `read()`, `write()` e `seek()` usata per i dischi. Un'interfaccia disponibile in molti sistemi operativi, tra i quali unix e Windows, è l'interfaccia di rete socket (*presa di corrente*).

Si pensi a una presa di corrente elettrica a muro: vi si può collegare qualunque apparecchiatura elettrica; per analogia, le chiamate di sistema di un'interfaccia socket permettono a un'applicazione di creare una socket, collegare una socket locale all'indirizzo di un altro punto della rete (ciò ha l'effetto di collegare questa applicazione alla socket creata da un'altra applicazione), controllare se un'applicazione si inserisce nella socket locale, e inviare o ricevere pacchetti di dati lungo la connessione. Per supportare lo sviluppo di server, l'interfaccia socket fornisce anche una funzione chiamata `select()` che gestisce un insieme di socket. Essa restituisce informazioni sulle socket per le quali sono presenti pacchetti che attendono d'essere ricevuti, e su quelle che hanno spazio per accettare un pacchetto da inviare. L'uso della funzione `select()` elimina il polling e il busy waiting altrimenti necessari per l'i/o di rete. Queste funzioni incapsulano il comportamento essenziale delle reti, facilitando notevolmente la creazione di applicazioni distribuite che possano sfruttare qualsiasi hardware di rete e stack di protocolli.

Sono stati sviluppati molti altri metodi per affrontare il problema della comunicazione di rete e fra i processi. Il sistema operativo Windows, per esempio, fornisce un'interfaccia per la scheda di rete e un'altra per i protocolli di rete. Il sistema operativo unix, banco di prova storico delle tecnologie di rete, offre pipe *half-duplex*, code fifo *full-duplex*, streams *full-duplex*, code di messaggi e socket. Maggiori informazioni sulla gestione delle reti in unix sono disponibili nel Paragrafo C.9 dell'Appendice C (reperibile sulla piattaforma MyLab).

12.3.3 Orologi e timer

La maggior parte dei calcolatori ha timer e orologi hardware che forniscono tre funzioni essenziali:

- segnare l'ora corrente;
- segnalare il tempo trascorso;
- regolare un timer in modo da avviare l'operazione x al tempo t .

Queste funzioni sono spesso usate sia dal sistema operativo sia da applicazioni per cui il tempo è un fattore importante. Purtroppo, le chiamate di sistema che realizzano queste funzioni non sono standardizzate da un sistema operativo all'altro.

Il dispositivo che misura la durata di un lasso di tempo e che può avviare un'operazione si chiama timer programmabile; si può regolare in modo da attendere un certo tempo e poi generare un'interruzione, e può anche ripetere questo processo continuamente, generando così interruzioni periodiche. Lo scheduler usa questo meccanismo per generare un'interruzione che sospende un processo quando il suo quanto di tempo è scaduto. Il sottosistema dell'i/o delle unità a disco lo usa per riversare periodicamente nei dischi il contenuto della *buffer cache*, e il sottosistema di rete lo usa per annullare operazioni che procedono troppo lentamente a causa di congestioni di rete o guasti. Il sistema operativo può inoltre fornire un'interfaccia per permettere ai processi utenti di usare i timer. Simulando orologi virtuali, il sistema operativo può anche gestire un numero di richieste d'uso dei timer maggiore del numero dei timer fisici. Per far ciò il kernel (o il driver del timer) mantiene una lista ordinata cronologicamente delle interruzioni richieste dagli utenti e dalle proprie procedure, e imposta il timer per la prima scadenza. Quando il timer genera l'interruzione, il kernel manda un segnale al richiedente, e reimposta il timer per la scadenza successiva.

I computer sono dotati di un hardware di temporizzazione che viene utilizzato per una varietà di scopi. I pc moderni includono un timer di eventi ad alta precisione (hpet), che funziona a velocità attorno ai 10 megahertz. Questo timer dispone di diversi comparatori che possono essere impostati per attivarsi una volta o ripetutamente quando il valore che contengono corrisponde a quello di hpet. Al momento dell'attivazione viene generata un'interruzione e le routine del sistema operativo di gestione del clock determinano il tipo di richiesta e intraprendono le opportune azioni. La precisione degli impulsi d'attivazione è limitata dalla bassa frequenza del timer, e dall'overhead aggiuntivo dato dal mantenimento di orologi virtuali. Inoltre, se lo stesso timer si usa per fornire l'ora corrente del sistema, questa potrà allontanarsi dai valori corretti (andare alla deriva). Il problema della deriva può essere corretto tramite protocolli progettati a tale scopo come ntp (Network Time Protocol), che utilizza sofisticati calcoli di latenza per mantenere l'orologio di un computer preciso quasi quanto un orologio atomico. Nella maggior parte dei calcolatori, l'orologio hardware è costruito sulla base di un contatore ad alta frequenza. In alcuni casi è possibile leggere da un registro del dispositivo il valore di questo contatore, cosicché esso possa essere visto come un orologio ad alta precisione. Sebbene non sia in grado di generare interruzioni, offre una misura accurata degli intervalli di tempo.

12.3.4 i/o non bloccante e asincrono

Un altro aspetto dell'interfaccia delle chiamate di sistema è la scelta fra i/o bloccante e non bloccante. Quando un'applicazione impiega una chiamata di sistema bloccante si sospende l'esecuzione dell'applicazione, che passa dalla coda dei processi pronti per l'esecuzione a una coda d'attesa del sistema. Quando la chiamata di sistema termina, l'applicazione è posta nuovamente nella coda dei processi pronti in modo che possa riprendere l'esecuzione; Quando riprenderà l'esecuzione, riceverà i valori riportati dalla chiamata di sistema. Le operazioni fisiche compiute dai dispositivi di i/o sono in genere asincrone – richiedono un tempo variabile o non prevedibile. Cionondimeno, la maggior parte dei sistemi operativi impiega chiamate di sistema bloccanti come interfaccia per le applicazioni, perché in questo caso il codice delle applicazioni è più facilmente comprensibile del corrispondente codice non bloccante.

Alcuni processi a livello utente necessitano di una forma non bloccante di i/o. Un esempio è quello di un'interfaccia utente con cui s'interagisce col mouse e la tastiera mentre elabora dati e li mostra sullo schermo. Un altro esempio è un'applicazione video che legge frame da un file su disco e simultaneamente li decomprime e li mostra sullo schermo.

Uno dei modi in cui chi progetta un'applicazione può sovrapporre elaborazione e i/o è scrivere un'applicazione a più thread. Alcuni di loro eseguono chiamate di sistema bloccanti, mentre altri continuano l'elaborazione. Alcuni sistemi operativi forniscono chiamate di sistema non bloccanti per l'i/o. Una chiamata di questo tipo non arresta l'esecuzione dell'applicazione per un tempo significativo. Al contrario, essa restituisce rapidamente il controllo all'applicazione, fornendo un parametro che indica quanti byte di dati sono stati trasferiti.

Una possibile alternativa alle chiamate di sistema non bloccanti è costituita dalle chiamate di sistema asincrone. Esse restituiscono immediatamente il controllo al chiamante, senza attendere che l'i/o sia stato completato. L'applicazione continua a essere eseguita, e il completamento dell'i/o è successivamente comunicato all'applicazione per mezzo dell'impostazione del valore di una variabile nello spazio d'indirizzi dell'applicazione oppure tramite la generazione di un segnale o interrupt software, o ancora tramite una procedura di richiamo (callback) eseguita fuori del normale flusso lineare d'elaborazione dell'applicazione. La differenza fra chiamate di sistema non bloccanti e asincrone è che una `read()` non bloccante restituisce immediatamente il controllo, fornendo i dati che è stato possibile leggere (l'intero numero di byte richiesti, una parte, o anche nessun dato). Una chiamata `read()` asincrona richiede un trasferimento di cui il sistema garantisce il completamento, ma solo in un momento successivo e non prevedibile. Entrambi i metodi sono illustrati dalla Figura 12.9.

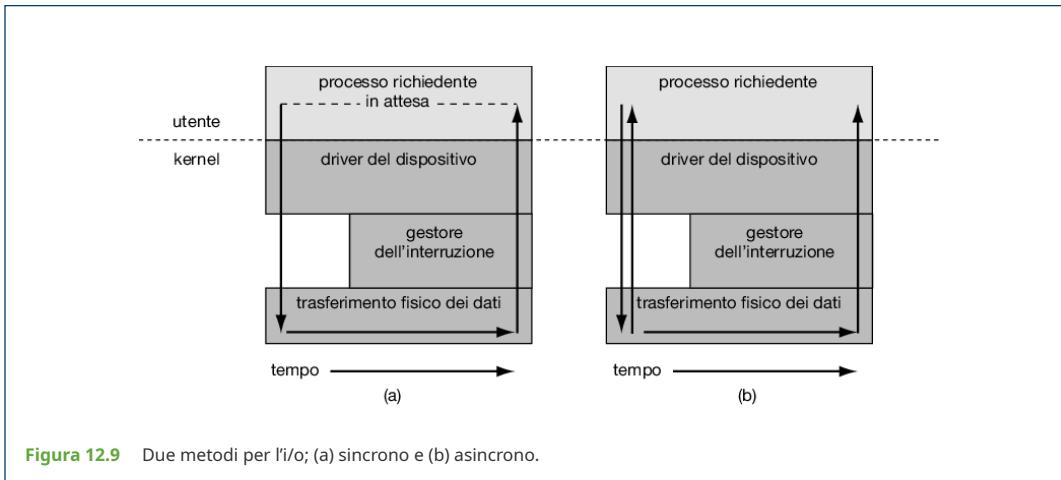


Figura 12.9 Due metodi per l'i/o; (a) sincrono e (b) asincrono.

In tutti i moderni sistemi operativi si verificano attività asincrone. Spesso queste attività non sono gestite da utenti o applicazioni, ma fanno parte del funzionamento del sistema operativo. Due validi esempi sono l'i/o del disco e della rete. Di norma, quando un'applicazione emette una richiesta di invio sulla rete o una richiesta di scrittura su disco, il sistema operativo rileva la richiesta, inserisce l'i/o in un buffer e restituisce il controllo all'applicazione. Appena possibile, per ottimizzare le prestazioni complessive, il sistema operativo completa la richiesta. Se nel frattempo si verifica un errore di sistema, l'applicazione perderà tutte le richieste in atto. I sistemi operativi impongono pertanto, di solito, un limite sul tempo massimo per rispondere a una richiesta. Per esempio, alcune versioni di unix ripuliscono i propri buffer del disco ogni 30 secondi; in altre parole ogni richiesta deve essere evasa entro 30 secondi. La coerenza dei dati delle applicazioni viene mantenuta dal kernel, che legge i dati dai suoi buffer prima di inviare richieste di i/o ai dispositivi, assicurando che i dati non ancora scritti siano comunque restituiti al richiedente. Si noti che più thread che eseguono i/o sullo stesso file potrebbero non ricevere dati coerenti, a seconda di come il kernel implementa il suo i/o. In questa situazione, i thread potrebbero dover utilizzare protocolli di locking. Alcune richieste di i/o devono essere eseguite immediatamente, per cui le chiamate di sistema di i/o forniscono di solito un modo per indicare che una determinata richiesta, o l'i/o verso un particolare dispositivo, dovrà essere eseguita in maniera sincrona.

Un buon esempio di comportamento non bloccante è la chiamata di sistema `select()` per le socket di rete. Essa include un argomento che specifica il tempo d'attesa massimo. Se questo valore è 0, l'applicazione può rilevare attività di rete senza arrestarsi. Tuttavia, l'uso della `select()` introduce un overhead aggiuntivo, perché essa può stabilire soltanto se sia possibile compiere dell'i/o: per un effettivo trasferimento di dati, la `select()` deve essere seguita da istruzioni come `read()` o `write()`. Una variante di questo metodo, adottata per esempio dal sistema Mach, è l'impiego di una chiamata di sistema bloccante per la lettura multipla. Essa specifica con una singola chiamata di sistema le richieste di lettura desiderate per diversi dispositivi, e termina non appena una di loro sia stata soddisfatta.

12.3.5 I/O vettorizzato

Alcuni sistemi operativi forniscono un'altra importante variante di i/o tramite le loro interfacce delle applicazioni. L'i/o vettorizzato permette a una chiamata di sistema di eseguire più operazioni di i/o che coinvolgono più locazioni. Per esempio, la chiamata di sistema unix `readv` accetta un vettore di buffer e permette di inserire nel vettore i dati letti da una sorgente oppure di scrivere da quel vettore verso una destinazione. Lo stesso trasferimento potrebbe essere realizzato per mezzo di diverse singole invocazioni di chiamate di sistema, ma questo metodo, detto scatter-gather, risulta utile per una serie di motivi.

Il trasferimento del contenuto di più buffer distinti tramite un'unica chiamata di sistema permette di evitare cambi di contesto e l'overhead delle chiamate di sistema. In assenza di i/o vettorizzato i dati dovrebbero prima essere trasferiti in un unico buffer più grande, nel giusto ordine, e poi trasmessi. Quest'ultimo metodo risulta piuttosto inefficiente. Inoltre, alcune versioni di scatter-gather forniscono l'atomicità, assicurando così che tutti gli i/o vengano eseguiti senza interruzioni (evitando dunque la corruzione di dati nel caso in cui altri thread stiano effettuando i/o verso gli stessi buffer). Quando possibile, i programmati sfruttano le potenzialità dell'i/o scatter-gather per aumentare la produttività e diminuire l'overhead del sistema.

12.4 Sottosistema di I/O del kernel

Il kernel fornisce molti servizi riguardanti l'i/o; vari servizi – scheduling, gestione del buffer, delle cache, delle code di spooling, riservazione dei dispositivi e gestione degli errori – sono offerti dal sottosistema di i/o del kernel, e sono realizzati a partire dai dispositivi e dai relativi driver. Il sottosistema di i/o è responsabile anche della propria salvaguardia contro processi malfunzionanti e utenti malintenzionati.

12.4.1 Scheduling dell'I/O

Fare lo scheduling di un insieme di richieste di i/o significa stabilirne un ordine d'esecuzione efficace; l'ordine in cui si verificano le chiamate di sistema da parte delle applicazioni è raramente la scelta migliore. Lo scheduling può migliorare le prestazioni complessive del sistema, distribuire egualmente gli accessi dei processi ai dispositivi e ridurre il tempo d'attesa medio per il completamento di un'operazione di i/o. Ecco un semplice esempio che illustra queste potenzialità. Si supponga che la testina di lettura di un'unità a disco sia vicina alla parte iniziale del disco, e che tre applicazioni impartiscano comandi di lettura bloccanti per quest'unità. L'applicazione 1 richiede la lettura di un blocco che si trova vicino alla parte finale del disco, l'applicazione 2 quella di un blocco vicino alla parte iniziale e l'applicazione 3 quella di un blocco situato nella zona centrale. Il sistema operativo può ridurre la distanza percorsa dalla testina del disco servendo le richieste nell'ordine 2, 3, 1. Simili riordinamenti delle sequenze di servizio delle richieste sono l'essenza dello scheduling dell'i/o.

I progettisti di sistemi operativi realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando un'applicazione richiede l'esecuzione di una chiamata di sistema di i/o bloccante, si aggiunge la richiesta alla coda relativa al dispositivo. Lo scheduler dell'i/o riorganizza l'ordine della coda per migliorare l'efficienza globale del sistema e il tempo medio d'attesa cui sono sottoposte le applicazioni. Il sistema operativo può anche tentare di essere equo, in modo che nessuna applicazione riceva un servizio carente, o può dare priorità alle richieste sensibili al ritardo. Per esempio, le richieste del sottosistema per la memoria virtuale potrebbero avere priorità su quelle delle applicazioni. Parecchi algoritmi di scheduling per l'i/o delle unità a disco sono descritti dettagliatamente nel Paragrafo 11.2.

I kernel che mettono a disposizione l'i/o asincrono devono essere in grado di tener traccia di più richieste di i/o contemporaneamente. A questo fine, alcuni sistemi associano una tabella dello stato dei dispositivi alla coda dei processi in attesa. Gli elementi della tabella – uno per ogni dispositivo di i/o – indicano il tipo, l'indirizzo e lo stato del dispositivo: non funzionante, inattivo o occupato. Se il dispositivo è impegnato nel servire una richiesta, il corrispondente elemento della tabella riporterà il tipo della richiesta e altri parametri a essa relativi. Si veda la Figura 12.10.

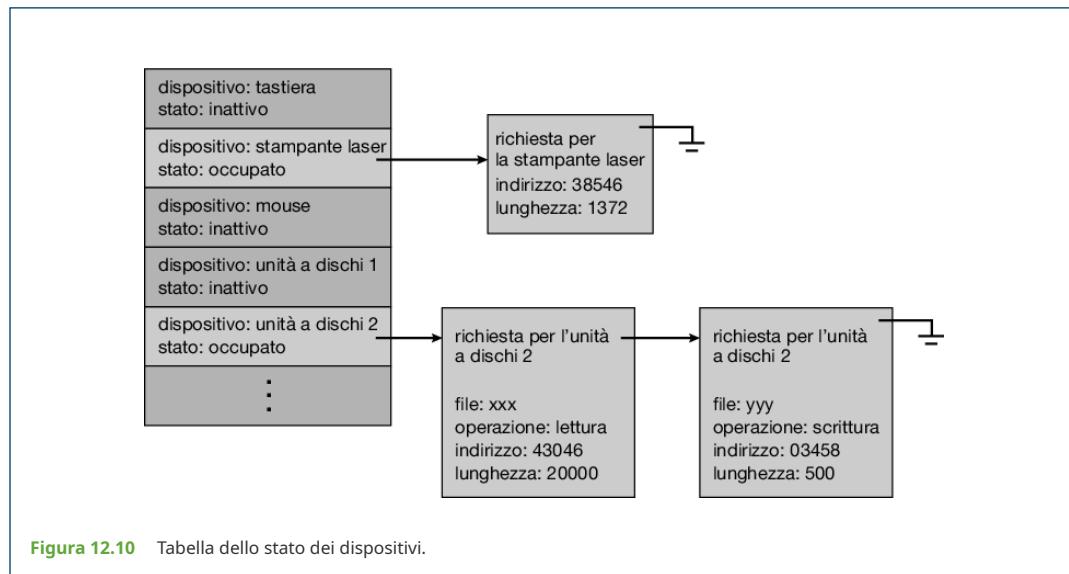


Figura 12.10 Tabella dello stato dei dispositivi.

Lo scheduling dell'i/o è uno dei modi in cui il sottosistema di i/o migliora l'efficienza di un calcolatore; un altro è l'uso di spazio di memorizzazione nella memoria centrale o nei dischi, per tecniche di buffering, caching e spooling.

12.4.2 Gestione dei buffer

Un buffer è un'area di memoria che contiene dati durante il trasferimento fra due dispositivi o tra un'applicazione e un dispositivo. Si ricorre ai buffer per tre ragioni. La prima è la necessità di gestire la differenza di velocità fra il produttore e il consumatore di un flusso di dati. Si supponga, per esempio, di ricevere un file attraverso un modem e di volerlo memorizzare in un'unità a disco: il

modem è circa mille volte più lento del disco, perciò conviene creare un buffer nella memoria principale per accumulare i byte che giungono dal modem. Quando tale buffer è pieno, si trasferisce il suo contenuto nel disco con un'unica operazione. Poiché quest'operazione di scrittura non è istantanea e il modem ha bisogno di ulteriore spazio per memorizzare i dati in arrivo, è necessario impiegare due buffer di questo tipo: quando il primo è pieno, si richiede la scrittura nel disco del suo contenuto e il modem comincia a scrivere nel secondo buffer mentre il primo viene scritto su disco. La scrittura nel disco dovrebbe terminare prima che il modem possa riempirlo, cosicché il modem potrà ricominciare a usare il primo buffer, mentre si trasferisce nel disco il contenuto del secondo. Questa doppia bufferizzazione svincola il produttore dal consumatore, rendendo così meno critico il problema della loro sincronizzazione. La necessità di questo disaccoppiamento è illustrata dalla Figura 12.11, che mostra le enormi differenze di velocità tra i dispositivi tipici di un calcolatore.

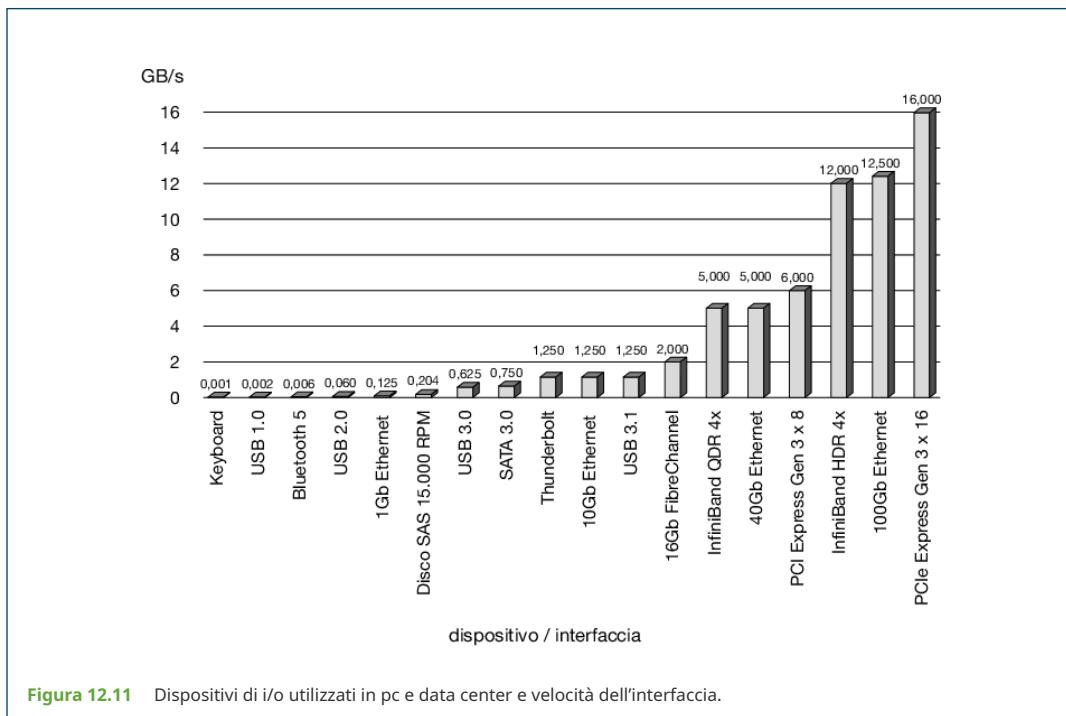


Figura 12.11 Dispositivi di i/o utilizzati in pc e data center e velocità dell'interfaccia.

Un secondo uso della bufferizzazione riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensioni diverse. Queste disparità sono particolarmente comuni nelle reti di calcolatori, dove spesso i buffer sono usati per frammentare e ricomporre messaggi. Quando un mittente spedisce un messaggio molto lungo, esso è spezzato in piccoli pacchetti che si spediscono attraverso la rete; il sistema destinatario provvede a ricostituire in un apposito buffer l'intero messaggio originario.

Il terzo modo in cui si può impiegare un buffer è per la realizzazione della *semantica della copia* nell'ambito dell'i/o delle applicazioni. Un esempio chiarirà il significato di semantica della copia. Si supponga che un'applicazione disponga di un buffer contenente dati da trasferire in un disco: esso richiederà l'esecuzione della chiamata di sistema `write()`, fornendo un puntatore al buffer e un numero intero che specifica il numero di byte da trasferire. Ci si può chiedere che cosa succede se, dopo che la chiamata di sistema restituisce il controllo all'applicazione, quest'ultima modifica il contenuto del buffer. Ebbene, la semantica della copia garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nel buffer al momento della chiamata di sistema, indipendentemente da ogni successiva modifica. Una semplice maniera di realizzare questa semantica consiste nel far sì che la chiamata di sistema `write()` copi i dati forniti dall'applicazione in un buffer del kernel prima di restituire il controllo all'applicazione stessa. La scrittura nel disco si compie dalla memoria del kernel, cosicché ogni successivo cambiamento nel buffer dell'applicazione non avrà effetti. In molti sistemi operativi si usa il metodo appena descritto: nonostante implichi una diminuzione dell'efficienza di certe operazioni di i/o, la sua semantica è chiara. Lo stesso effetto, tuttavia, si può ottenere più efficientemente tramite un uso intelligente della memoria virtuale e della protezione data dal copy-on-write delle pagine.

12.4.3 Cache

Una cache è una regione di memoria veloce che serve per mantenere copie di dati: l'accesso a queste copie è più rapido dell'accesso agli originali. Per esempio, le istruzioni di un processo correntemente in esecuzione sono memorizzate in un disco, copiate nella memoria fisica (che assume il ruolo di cache rispetto al disco) e copiate ulteriormente nelle cache primaria e secondaria della cpu. La differenza fra un buffer e una cache consiste nel fatto che il primo può contenere dati di cui non esiste altra copia, mentre una cache, per definizione, mantiene su un mezzo più efficiente una copia di informazioni memorizzate altrove.

L'uso delle cache e l'uso dei buffer sono due funzioni distinte, anche se a volte una stessa regione di memoria si può usare per entrambi gli scopi. Per esempio, per realizzare la semantica della copia e permettere uno scheduling efficiente dell'i/o su disco, il sistema operativo impiega dei buffer in memoria centrale per i dati dei dischi. Questi buffer sono anche usati come cache per migliorare l'efficienza delle operazioni di i/o che coinvolgono file condivisi da più applicazioni o file per i quali gli accessi per lettura e scrittura si susseguono rapidamente. Quando riceve una richiesta di i/o relativa a un file, il kernel controlla se la parte interessata del file è già presente nella cache: in questo caso è possibile evitare o differire l'accesso fisico al disco. Inoltre, i dati da scrivere nel disco sono depositati nella cache per diversi secondi, cosicché si accumulano grandi quantità di dati da trasferire: ciò permette uno scheduling efficiente. La strategia consistente nel differire le scritture per migliorare l'efficienza dell'i/o è illustrata nel Paragrafo 19.6.4, nel contesto dell'accesso ai file remoti.

12.4.4 Code di spooling e riservazione dei dispositivi

Una coda di spooling è un buffer contenente dati da inviare a un dispositivo che non può accettare flussi di dati intercalati, per esempio una stampante. Sebbene una stampante possa servire una sola richiesta alla volta, diverse applicazioni devono poter richiedere simultaneamente la stampa di dati, senza che le stampe si mischino. Il sistema operativo risolve questo problema filtrando tutti i dati per la stampante: i dati da stampare provenienti da ogni singola applicazione si registrano in uno specifico spool file su disco; quando un'applicazione termina di stampare, il sistema di spooling aggiunge tale file alla coda di stampa; quest'ultima viene copiata sulla stampante, un file per volta. In certi sistemi operativi questa funzione viene gestita da un processo di sistema specializzato (demone), in altri da un thread del kernel. In entrambi i casi il sistema operativo fornisce un'interfaccia di controllo che permette agli utenti e agli amministratori del sistema di esaminare la coda, eliminare elementi della coda prima che siano stampati, sospendere il servizio di stampa per attività di manutenzione, e così via.

Alcuni dispositivi, come le unità a nastro e le stampanti, non possono alternare più richieste concorrenti di i/o da parte di diverse applicazioni. Lo spooling è uno dei modi in cui il sistema operativo può coordinare output concorrenti; un altro è quello di fornire esplicite funzioni di coordinamento. Alcuni sistemi operativi, fra i quali il vms, permettono di accedere a un dispositivo in modo esclusivo: un processo può accedere a un dispositivo che non utilizzato, riservandosene l'uso, e restituirlo al sistema quando non ne ha più bisogno. Altri sistemi operativi impediscono l'apertura di più di un handle di file per un dato dispositivo. Molti sistemi operativi forniscono funzioni che permettono ai processi stessi di coordinare l'uso esclusivo dei dispositivi: il sistema Windows, per esempio, mette a disposizione chiamate di sistema che permettono a un'applicazione di aspettare finché un certo dispositivo si liberi. Inoltre la sua chiamata di sistema `OpenFile()` accetta un parametro che specifica il tipo d'accesso concesso ad altri thread concorrenti. In questi sistemi le applicazioni hanno la responsabilità di evitare le situazioni di stallo.

12.4.5 Gestione degli errori

Un sistema operativo che usa la protezione della memoria può difendersi da molti tipi di errori dovuti all'hardware o alle applicazioni, cosicché il blocco completo del sistema non è la necessaria conseguenza di ogni piccolo malfunzionamento. I dispositivi e i trasferimenti di i/o possono essere soggetti a errori in molti modi, sia per motivi contingenti, come il sovraccarico di una rete di comunicazione, sia per ragioni "permanenti", come nel caso in cui il controllore dell'unità a disco si guasti. I sistemi operativi sono spesso capaci di compensare efficacemente le conseguenze negative dovute a errori temporanei: se per esempio una chiamata di sistema `read()` non ha successo, il sistema ritenterà la lettura; se una chiamata `send()` provoca un errore, il protocollo di rete può richiedere una `resend()`. Purtroppo, però, è difficile che il sistema operativo riesca a compensare gli effetti di errori dovuti a guasti permanenti di qualche componente importante.

Di norma, una chiamata di sistema di i/o riporta un bit d'informazione sullo stato d'esecuzione della chiamata, che indica la riuscita o l'insuccesso dell'operazione richiesta. Il sistema operativo unix usa inoltre una variabile intera detta `errno` per codificare piuttosto genericamente il tipo d'errore avvenuto; i valori possibili sono un centinaio e denotano errori dovuti per esempio a puntatori non validi, file non aperti o argomenti oltre i limiti ammessi. Per contro, alcuni tipi di dispositivi possono fornire informazioni assai dettagliate sugli errori, sebbene molti sistemi operativi attuali non siano progettati per passare questi dati alle applicazioni. Per esempio, il malfunzionamento di un dispositivo scsi è riportato dal protocollo scsi a tre livelli di dettaglio: usando un codice di rilevazione (*sense key*) che identifica la natura generale dell'errore (per esempio: errore hardware, o richiesta illegale); un codice di rilevazione addizionale che descrive la categoria cui appartiene il malfunzionamento (parametro del comando errato, o insuccesso del self-test del dispositivo); e infine un qualificatore del codice di rilevazione addizionale che fornisce informazioni ancora più dettagliate (quale parametro è errato, o quale componente del dispositivo non ha superato il test). Inoltre, molti dispositivi scsi mantengono internamente pagine di log degli errori avvenuti; queste pagine possono essere richieste dalla macchina, ma ciò accade raramente.

12.4.6 Protezione dell'I/O

Gli errori sono strettamente connessi alla tematica della protezione. Un processo utente che, intenzionalmente o accidentalmente, cerchi di impartire istruzioni di i/o illegali può danneggiare il funzionamento normale di un sistema. Per impedire che tali danneggiamenti abbiano luogo, si possono opporre diverse contromisure.

Onde evitare che gli utenti effettuino operazioni di i/o illegali, si definiscono come privilegiate tutte le istruzioni relative all'i/o. Ne consegue che gli utenti non potranno impartire in modo diretto alcuna istruzione, ma dovranno farlo attraverso il sistema operativo. Un programma utente, per eseguire l'i/o, invoca una chiamata di sistema per chiedere al sistema operativo di svolgere una data operazione nel suo interesse (Figura 12.12). Il sistema, passando alla modalità privilegiata, verifica che la richiesta sia valida e, in tal caso, esegue l'operazione; esso trasferisce quindi il controllo all'utente.

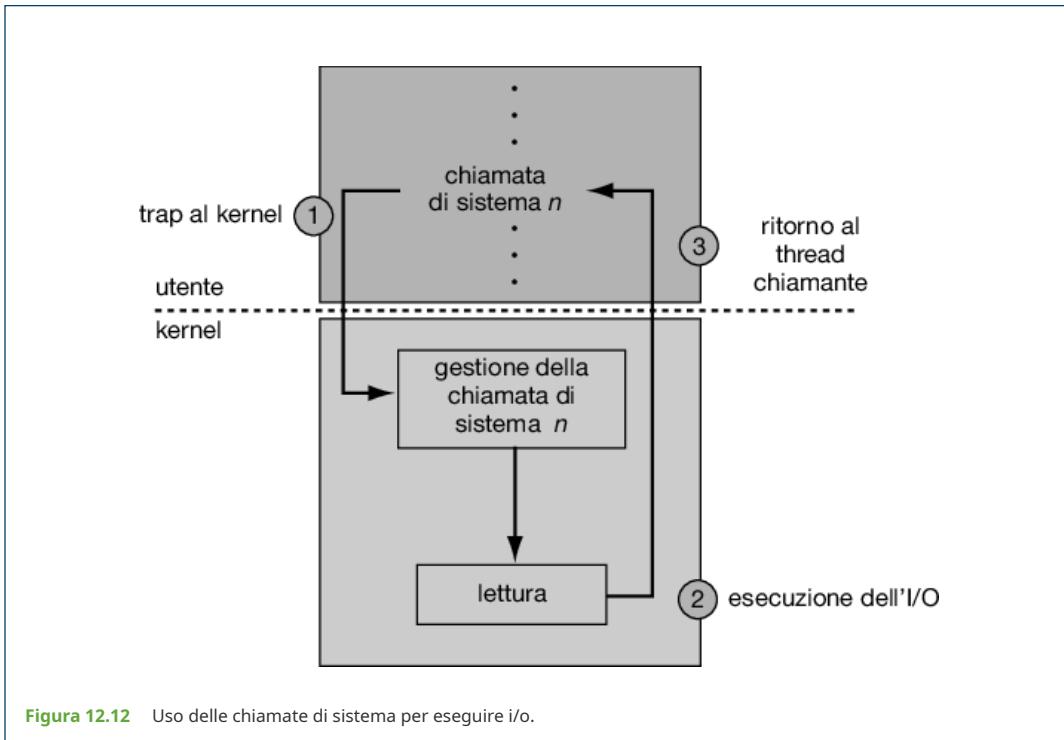


Figura 12.12 Uso delle chiamate di sistema per eseguire i/o.

Inoltre, il sistema di protezione della memoria deve tutelare dall'accesso degli utenti tutti gli indirizzi mappati in memoria e gli indirizzi delle porte di i/o. Il kernel, tuttavia, non può semplicemente negare qualunque tentativo di accesso da parte degli utenti: quasi tutti i videogiochi, nonché i programmi per il montaggio e la riproduzione di video, per esempio, per ottimizzare le prestazioni della grafica necessitano dell'accesso diretto alla memoria del controllore della grafica, che è gestita in modalità memory mapped. In questi casi, il kernel potrebbe applicare dei lock per assegnare a un solo processo per volta la porzione della memoria grafica che rappresenta una finestra sullo schermo.

12.4.7 Strutture dati del kernel

Il kernel ha bisogno di mantenere informazioni sullo stato dei componenti di i/o, e usa a questo fine diverse strutture dati interne, un esempio delle quali è la tabella dei file aperti descritta nel Paragrafo 14.1. Il kernel usa molte strutture di questo tipo per tener traccia dei collegamenti di rete, delle comunicazioni con i dispositivi a caratteri e di altre attività di i/o.

Il sistema operativo unix permette l'accesso, con le modalità tipiche del file system, a diversi oggetti: file degli utenti, dispositivi, spazio d'indirizzi dei processi, e altri ancora. Sebbene ognuno di questi oggetti supporti una chiamata `read()`, le semantiche sono diverse secondo i casi. Quando il kernel, per esempio, deve leggere un file utente, ha bisogno di controllare la *buffer cache* prima di decidere l'effettiva esecuzione di un'operazione di i/o su un disco. Per leggere un disco privo di struttura logica (*raw disk*), il kernel deve accertarsi del fatto che la dimensione dell'insieme dei dati di cui è stato richiesto il trasferimento sia un multiplo della dimensione dei settori del disco e sia allineato con il settore interessato. Per leggere l'immagine di un processo, tutto ciò che occorre è copiare dati dalla memoria. unix incapsula queste differenze in una struttura uniforme usando una tecnica orientata agli oggetti. Il record di un file aperto, mostrato nella Figura 12.13, contiene una tabella di puntatori alle procedure appropriate secondo il tipo di file in questione.

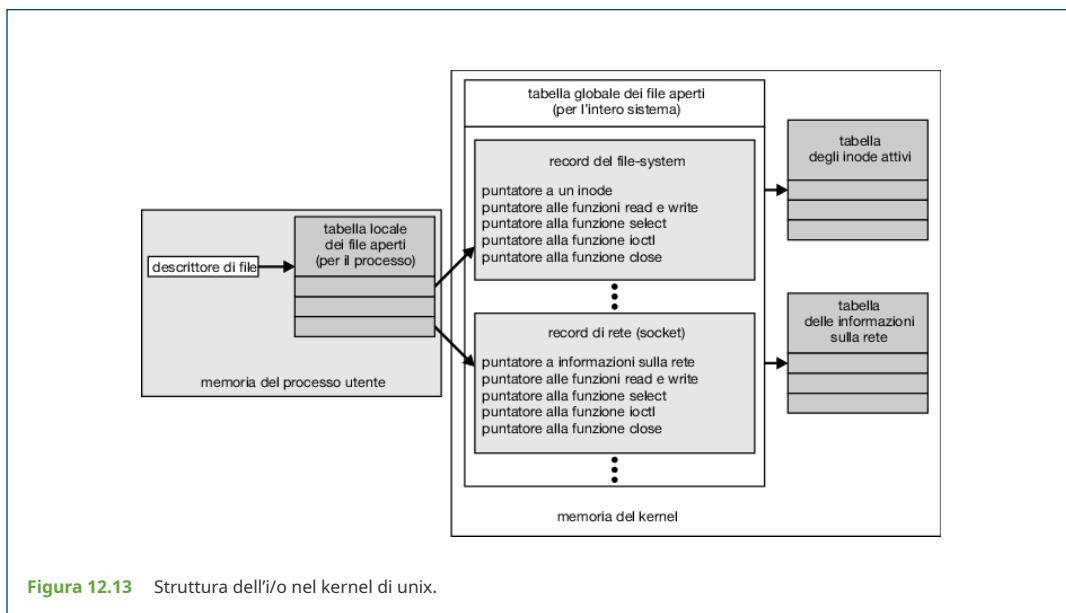


Figura 12.13 Struttura dell'i/o nel kernel di unix.

Alcuni sistemi operativi applicano metodi orientati agli oggetti in misura più rilevante: il sistema Windows, per esempio, usa per l'i/o un sistema basato sullo scambio di messaggi. Una richiesta di i/o si converte in un messaggio che s'invia tramite il kernel al sottosistema per la gestione dell'i/o, quindi al driver del dispositivo; i contenuti del messaggio possono essere modificati a ogni passaggio intermedio. Quando l'operazione richiesta è di output, il messaggio contiene i dati da scrivere; quando invece l'operazione richiesta è di input, il messaggio contiene un buffer che si usa per ricevere i dati. Questo metodo può comportare una minore efficienza rispetto alle tecniche procedurali basate sulla condivisione delle strutture dati, ma semplifica la progettazione e la struttura del sistema di i/o e permette una maggiore flessibilità.

12.4.8 Gestione energetica

I computer che risiedono nei data center possono sembrare esenti dai problemi legati al consumo di energia, ma poiché i costi dell'energia aumentano e il mondo diventa sempre più preoccupato per gli effetti a lungo termine delle emissioni di gas serra, i data center sono diventati motivo di preoccupazione e un importante obiettivo è di aumentarne l'efficienza energetica. Visto che l'elettricità genera calore e i componenti del computer possono guastarsi a causa delle alte temperature, bisogna tener conto anche del raffreddamento: si pensi che il raffreddamento di un moderno data center può consumare il doppio dell'energia elettrica necessaria per alimentare l'apparecchiatura. Sono oggi in uso molti approcci per l'ottimizzazione dell'utilizzo di energia di un data center, che vanno dalle tecniche per il ricircolo dell'aria, al raffreddamento mediante fonti naturali come l'acqua dei laghi, all'utilizzo dei pannelli solari.

Anche i sistemi operativi giocano un ruolo nell'uso dell'energia (e quindi nella generazione di calore e nel raffreddamento). Negli ambienti di cloud computing i carichi di elaborazione possono essere regolati mediante strumenti di monitoraggio e gestione in modo da poter togliere tutti i processi utente da alcuni sistemi, disattivare tali sistemi e spegnerli finché il carico non ne rende indispensabile l'utilizzo. Un sistema operativo potrebbe analizzare il suo carico di lavoro e, se sufficientemente basso e se consentito dall'hardware, spegnere componenti come cpu e dispositivi di i/o esterni.

I core della cpu possono essere messi in sospensione quando il carico del sistema non li richiede e riattivati quando il carico aumenta e sono necessari più core per eseguire i thread in coda. Il loro stato, ovviamente, deve essere salvato durante la sospensione e ripristinato alla riattivazione. Questa funzione è necessaria nei server, perché un data center con molti server può utilizzare grandi quantità di elettricità e disabilitare i core non necessari può ridurre le esigenze di energia elettrica (e di raffreddamento).

Nel mobile computing la gestione dell'alimentazione diventa un aspetto prioritario del sistema operativo. La riduzione del consumo energetico e quindi l'ottimizzazione della durata della batteria aumentano l'usabilità di un dispositivo e lo aiutano a competere con i dispositivi concorrenti. I dispositivi mobili di oggi offrono le funzionalità dei desktop di fascia alta del passato, ma sono alimentati da batterie e sono abbastanza piccoli da poter essere riposti in tasca. Al fine di garantire una durata soddisfacente della batteria, i moderni sistemi operativi mobili sono stati progettati ex-novo, facendo della gestione energetica una caratteristica chiave. Esaminiamo in dettaglio tre funzioni principali che consentono al popolare sistema mobile Android di massimizzare la durata della batteria: collasso di potenza, gestione dei consumi a livello di componenti e wakelock.

Il collasso di potenza è la capacità di mettere un dispositivo in un profondo stato di riposo in cui utilizza solo una quantità marginale di energia in più rispetto a quando è completamente spento, ma rimane in grado di rispondere a stimoli esterni, come la pressione di un pulsante che porta a una sua rapida riaccensione. Il collasso di potenza si ottiene spegnendo molti dei singoli componenti all'interno di un dispositivo, per esempio lo schermo, i diffusori audio e il sottosistema di i/o, in modo che non consumino energia. Il sistema operativo colloca quindi la cpu nello stato di sospensione più basso. Una moderna cpu arm può consumare centinaia di milliwatt per core sotto un tipico carico di lavoro, ma solo una manciata di milliwatt nel suo stato di sospensione più basso. In tale

stato, sebbene la cpu sia inattiva, può ricevere un'interruzione, svegliarsi e riprendere la sua attività precedente molto rapidamente. Pertanto, un telefono Android inattivo nella tua tasca utilizza pochissima energia, ma può prendere vita quando riceve una telefonata.

In che modo Android è in grado di spegnere i singoli componenti di un telefono? Come fa a sapere quando è sicuro spegnere la memoria flash e come fa a farlo prima di spegnere il sottosistema generale di i/o? La risposta è la gestione energetica a livello di componente, un'infrastruttura in grado di capire la relazione tra i componenti e determinare se un componente è in uso. Per capire la relazione tra i componenti, Android crea un albero che rappresenta la topologia dei dispositivi fisici del telefono. Per esempio, in una tale topologia, le memorie flash e usb sono sottorami del sottosistema di i/o, che è sottorami del bus di sistema, che a sua volta si collega alla cpu. Per determinare l'utilizzo dei componenti, ogni componente è associato al suo driver di dispositivo che verifica se è in uso, per esempio se c'è un i/o in attesa della memoria flash o se un'applicazione ha un riferimento aperto al sottosistema audio. Con queste informazioni, Android può gestire la potenza da fornire ai singoli componenti del telefono: se un componente non è utilizzato, viene disattivato. Se, per esempio, tutti i componenti sul bus di sistema non sono utilizzati, il bus di sistema viene disattivato, e se tutti i componenti dell'intero albero dei dispositivi non sono utilizzati, il sistema potrebbe utilizzare il collasso di potenza.

Grazie a queste tecnologie, Android può gestire in modo aggressivo il suo consumo energetico. Manca però un ultimo pezzo al puzzle: la capacità delle applicazioni di impedire temporaneamente al sistema di utilizzare il collasso di potenza. Considerate un utente che gioca, guarda un video o aspetta che una pagina web si apra: in tutti questi casi l'applicazione necessita di un modo per mantenere il dispositivo attivo, almeno temporaneamente. I wakelock abilitano questa funzionalità. Le applicazioni acquisiscono e rilasciano wakelock in base alle esigenze. Quando un'applicazione detiene un wakelock, il kernel impedirà al sistema di utilizzare il collasso di potenza. Per esempio, durante l'aggiornamento di un'applicazione, Android Market manterrà un wakelock per garantire che il sistema non vada in sospensione fino a quando l'aggiornamento non è completo. Una volta completato l'aggiornamento, Android Market rilascerà il wakelock, consentendo al sistema di entrare in modalità collasso di potenza.

Il risparmio energetico si basa in generale sulla gestione dei dispositivi, che è più complicata di quanto abbiano finora descritto. Al momento dell'avvio, il firmware analizza l'hardware del sistema e crea un albero dei dispositivi in ram. Il kernel utilizza quindi la struttura ad albero per caricare i driver di dispositivo e gestire i dispositivi. Tuttavia è necessario gestire molte altre attività relative ai dispositivi, tra cui l'aggiunta e la rimozione di dispositivi da un sistema in esecuzione ("hot-plug"), la comprensione e la modifica degli stati del dispositivo e la gestione energetica. I moderni computer general-purpose utilizzano un altro tipo di codice firmware, l'acpi (advanced configuration and power interface), per gestire questi aspetti dell'hardware. acpi è uno standard (<http://www.acpi.info>) con molte funzionalità e fornisce il codice che viene eseguito come routine richiamabile dal kernel per la rilevazione e la gestione dello stato del dispositivo, la gestione degli errori del dispositivo e la gestione dell'energia. Per esempio, quando il kernel ha bisogno di spegnere un dispositivo, richiama il driver del dispositivo, che richiama le routine acpi, che comunicano poi con il dispositivo.

12.4.9 Concetti principali del sottosistema di I/O del kernel

Riassumendo, il sistema per l'i/o coordina un'ampia raccolta di servizi disponibili per le applicazioni e per altre parti del kernel; in generale sovrintende alle seguenti funzioni:

- gestione dello spazio dei nomi per file e dispositivi;
- controllo dell'accesso ai file e ai dispositivi;
- controllo delle operazioni (per esempio, un modem non può effettuare un `seek()`);
- allocazione dello spazio per il file system;
- allocazione dei dispositivi;
- gestione dei buffer, delle cache e delle code di spooling;
- scheduling dell'i/o;
- controllo dello stato dei dispositivi, gestione degli errori e procedure di ripristino;
- configurazione e inizializzazione dei driver dei dispositivi;
- gestione energetica dei dispositivi dell'i/o.

I livelli superiori del sottosistema per la gestione dell'i/o accedono ai dispositivi per mezzo dell'interfaccia uniforme fornita dai driver.

12.5 Trasformazione delle richieste di I/O in operazioni hardware

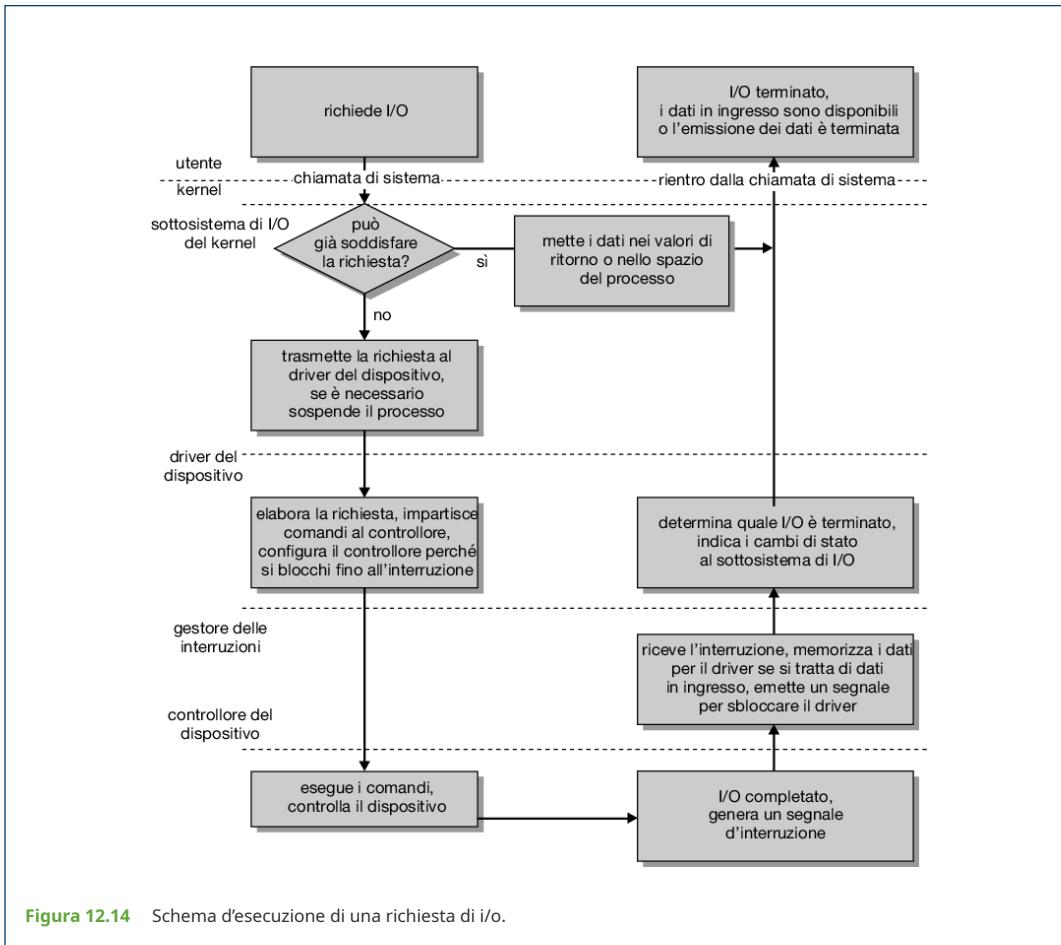
Il meccanismo di handshaking tra un driver e un controllore di dispositivo è già stato illustrato; tuttavia, non si è ancora spiegato come il sistema operativo associa alla richiesta di un'applicazione un insieme di fili di rete o uno specifico settore di disco. Si consideri per esempio la lettura di un file da un'unità a disco. L'applicazione fa riferimento ai dati per mezzo del nome del file: è compito del file system fornire il modo di giungere, attraverso la struttura delle directory, alla regione del disco appropriata, cioè quella dove i dati del file sono fisicamente residenti. Nell'ms-dos, per esempio, il nome del file è associato a un numero che individua un elemento della tabella d'accesso ai file; tale elemento identifica i blocchi del disco assegnati al file. In unix il nome è associato a un numero di *inode*; l'*inode* corrispondente contiene le informazioni necessarie per individuare lo spazio allocato. Ma come viene effettuato il collegamento fra il nome del file e il controller del disco (indirizzo hardware della porta o registri memory mapped del controller)?

Un metodo è quello utilizzato da un sistema relativamente semplice come l'ms-dos fat (un file system relativamente semplice, tuttora utilizzato come comune formato di interscambio). La prima parte di un nome di file dell'ms-dos, precisamente la parte che precede i due punti, identifica uno specifico dispositivo. Per esempio, c: è la parte iniziale di ogni nome di file residente nell'unità a disco principale. Questa convenzione è codificata all'interno del sistema operativo: c: è associato a uno specifico indirizzo di porta per mezzo di una tabella dei dispositivi. Grazie all'uso dei due punti come separatore, lo spazio dei nomi dei dispositivi è distinto dallo spazio dei nomi del file system, ciò semplifica al sistema operativo l'associazione di funzioni aggiuntive ai dispositivi. Per esempio, è facile attivare lo spooling per i file di cui è stata richiesta la stampa.

Se, invece, i nomi dei dispositivi sono inclusi nell'ordinario spazio dei nomi del file system, come in unix, sono automaticamente disponibili i servizi legati ai nomi dei file. Per esempio, se il file system associa dei possessori ai nomi dei file e fornisce il controllo degli accessi a ogni nome di file, si potrà controllare anche l'accesso ai dispositivi, ed essi avranno un possessore. Visto che i file risiedono nei dispositivi, una tale interfaccia fornisce due livelli d'accesso al sistema d'i/o: i nomi si possono usare per accedere ai dispositivi stessi o ai file in essi contenuti.

unix rappresenta i nomi dei dispositivi all'interno dell'ordinario spazio dei nomi del file system. A differenza di un nome di file dell'ms-dos fat, che include i due punti come separatore, in un nome di percorso di unix il nome del dispositivo non è esplicitamente separato. In effetti, nessuna parte del nome di percorso di un file è il nome di un dispositivo; unix impiega una tabella di montaggio (*mount table*), per associare i prefissi dei nomi di percorso ai corrispondenti nomi di dispositivi. Quando deve risolvere un nome di percorso, il sistema esamina la tabella per trovare il più lungo prefisso corrispondente: questo elemento della tabella indica il nome del dispositivo voluto. Anche questo nome è rappresentato come un oggetto del file system: tuttavia, quando unix cerca questo nome nelle strutture delle directory del file system, non trova il numero di un *inode*, ma una coppia di numeri *<principale, secondario>* (*<major, minor>*) che identifica un dispositivo. Il numero principale individua il driver che si deve usare per gestire l'i/o nel dispositivo in questione, mentre il numero secondario deve essere passato a questo driver affinché esso possa determinare, per mezzo di un'altra tabella, l'indirizzo della porta o l'indirizzo memory mapped del controllore del dispositivo interessato. I moderni sistemi operativi riescono a ottenere un notevole grado di flessibilità grazie all'uso di tabelle di lookup a vari livelli durante il processo che porta da una richiesta al controllore del dispositivo; questo processo, inoltre, è del tutto generale, cosicché non è necessario ricompilare il kernel ogni volta che si aggiungono al calcolatore nuovi dispositivi e nuovi driver. In effetti, alcuni sistemi operativi hanno la capacità di caricare driver di dispositivi su richiesta: all'avviamento, il sistema sonda i bus per determinare quali dispositivi siano presenti; quindi carica i necessari driver, operazione che può anche essere rinviata fino alla prima richiesta di i/o.

La seguente descrizione del tipico svolgimento di una richiesta di lettura bloccante (Figura 12.14) indica che l'esecuzione di un'operazione di i/o richiede una gran quantità di passi; ciò implica l'uso di un'enorme numero di cicli di cpu.



1. Un processo esegue una chiamata di sistema `read()` bloccante relativa a un descrittore di file di un file precedentemente aperto.
2. Il codice della chiamata di sistema all'interno del kernel controlla la correttezza dei parametri. Nel caso di input, se i dati sono già presenti nella *buffer cache*, si passano al processo chiamante e l'operazione è conclusa.
3. Altrimenti, è necessario eseguire un'operazione di i/o fisico. Si rimuove il processo dalla coda dei processi pronti per l'esecuzione per inserirlo nella coda d'attesa relativa al dispositivo interessato e si effettua lo scheduling della richiesta di i/o. Infine il sottosistema di i/o invia la richiesta al driver del dispositivo; a seconda del sistema operativo, ciò avviene tramite la chiamata di una procedura, o per mezzo dell'invio di un messaggio interno al kernel.
4. Il driver del dispositivo assegna un buffer nello spazio d'indirizzi del kernel che serve per ricevere i dati immessi, ed esegue lo scheduling dell'i/o. Infine il driver imparisce comandi al controllore del dispositivo scrivendo nei suoi registri.
5. Il controllore aziona il dispositivo hardware per compiere il trasferimento dei dati.
6. Il driver può operare in polling, o può aver predisposto un trasferimento dma nella memoria del kernel. Supponiamo che il trasferimento sia gestito dal controllore dma, il quale genera un'interruzione al termine dell'operazione.
7. Tramite il vettore delle interruzioni, si attiva l'appropriato gestore dell'interruzione che, dopo aver memorizzato i dati necessari, avverte con un segnale il driver del dispositivo ed effettua il ritorno dall'interruzione.
8. Il driver riceve il segnale, individua la richiesta di i/o che è stata completata, si accerta della riuscita o del fallimento dell'operazione e segnala al sottosistema di i/o del kernel il completamento dell'operazione.
9. Il kernel trasferisce dati e/o codici di stato nello spazio degli indirizzi del processo chiamante, e sposta tale processo dalla coda d'attesa alla coda dei processi pronti per l'esecuzione.
10. Nel momento in cui è posto nella coda dei processi pronti per l'esecuzione, il processo non è più bloccato: quando lo scheduler gli assegnerà la cpu, esso riprenderà l'elaborazione. L'esecuzione della chiamata di sistema è completata.

12.6 STREAMS

Il sistema operativo unix System V offre un interessante meccanismo, chiamato streams, che permette a un'applicazione di comporre dinamicamente catene di codice di driver. Uno *stream* è una connessione *full-duplex* fra un driver di dispositivo e un processo utente, e consiste di un elemento di interfaccia per il processo utente (*stream head*), un elemento che controlla il dispositivo (*driver end*), ed eventualmente un certo numero di moduli intermedi fra questi due elementi (*stream modules*). Tutti questi componenti contengono una coppia di code, una di lettura e una di scrittura; per il trasferimento dei dati tra le due code s'impiega uno schema a scambio di messaggi. La Figura 12.15 mostra la struttura del meccanismo di streams.

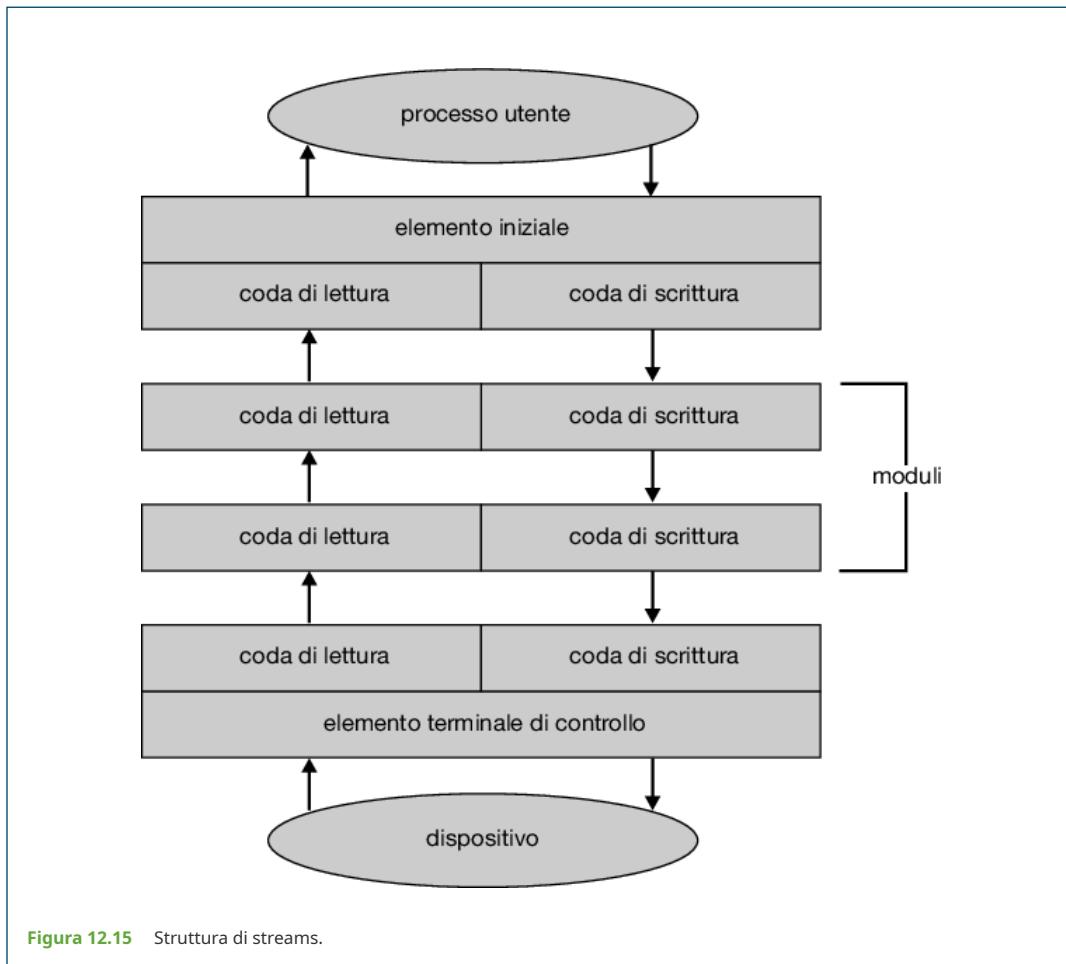


Figura 12.15 Struttura di streams.

Le funzioni d'elaborazione di streams sono fornite da moduli che si inseriscono nello stream attraverso la chiamata di sistema `ioctl()`. Un processo può, per esempio, aprire tramite uno stream una porta seriale, e può inserire un modulo per editare i dati immessi. Poiché i messaggi sono scambiati tra code di moduli adiacenti, la coda di un modulo potrebbe mandare in overflow la coda di un modulo adiacente. Per prevenire questo problema, una coda può disporre di un meccanismo di controllo di flusso. Senza controllo di flusso, una coda accetta tutti i messaggi e li trasferisce immediatamente alla coda del modulo adiacente, senza buffering. Una coda che invece impiega il controllo di flusso memorizza i messaggi in un buffer; se lo spazio disponibile in esso non è sufficiente, non accetta messaggi. Questo meccanismo richiede scambi di messaggi di controllo tra code in moduli adiacenti.

Un processo utente usa le chiamate di sistema `write()` o `putmsg()` per scrivere dati in un dispositivo; la chiamata di sistema `write()` scrive semplicemente dati non strutturati nello *stream*, mentre `putmsg()` permette al processo utente di specificare un messaggio. Indipendentemente dalla chiamata di sistema adoperata dal processo utente, lo stream head copia i dati in un messaggio e li recapita alla coda del modulo successivo. Questa copiatura dei messaggi continua finché il messaggio non giunge al driver end e quindi al dispositivo. Analogamente, il processo utente legge i dati dallo stream head usando la chiamata di sistema `read()` oppure `getmsg()`. Se si usa la `read()` lo stream head preleva un messaggio dalla coda del modulo adiacente e riporta al processo dati ordinari (una sequenza non strutturata di byte). Se si usa la `getmsg()` viene inviato un messaggio al processo utente.

L'i/o per mezzo di streams è asincrono (o non bloccante), con l'eccezione di quando il processo utente comunica con lo stream head. Mentre scrive nello *stream*, il processo utente si blocca, se la coda successiva impiega il controllo di flusso, finché non ci sia spazio sufficiente per copiarvi il messaggio. Analogamente, il processo utente si blocca durante la lettura dallo *stream* finché non ci sono dati disponibili.

Come si è detto, il driver end, come lo stream head e i moduli intermedi, ha una coda di lettura e una di scrittura. Tuttavia deve poter rispondere a interruzioni come quelle generate quando un pacchetto di dati è pronto per essere letto da una rete. A differenza dello stream head che si può bloccare se non è possibile copiare un messaggio nella coda del modulo successivo, il driver end deve gestire tutti i dati in arrivo. I driver devono anche supportare il controllo di flusso. Tuttavia, se il buffer di un dispositivo è pieno, di solito ignora i messaggi in entrata. Si consideri una scheda di rete il cui buffer d'ingresso sia pieno; la scheda di rete deve semplicemente ignorare gli ulteriori messaggi fintantoché non vi sia sufficiente spazio per memorizzare i messaggi in arrivo.

Il vantaggio nell'utilizzo di streams consiste nel disporre di un ambiente che permette uno sviluppo modulare e incrementale di driver di dispositivi e protocolli di rete. I moduli possono essere usati da diversi stream e quindi da diversi dispositivi. Per esempio, un modulo di rete potrebbe essere adoperato sia da una scheda di rete Ethernet sia da una scheda di rete wireless 802.11. Inoltre, invece di trattare l'i/o di dispositivi a caratteri come una sequenza non strutturata di byte, streams permette la gestione dei limiti dei messaggi e delle informazioni di controllo tra i diversi moduli. L'impiego di streams è assai diffuso nella maggior parte dei sistemi unix, ed è il metodo più usato per la scrittura di protocolli e driver di dispositivi. In unix System V e in Solaris, per esempio, il meccanismo delle socket è realizzato tramite streams.

12.7 Prestazioni

L'i/o è uno tra i principali fattori che influiscono sulle prestazioni di un sistema: richiede un notevole impegno della cpu per l'esecuzione del codice dei driver e per uno scheduling equo ed efficiente dei processi quando essi sono bloccati e sbloccati. I risultanti cambi di contesto sfruttano fino in fondo la cpu e le sue memorie cache. L'i/o, inoltre, rivela le eventuali inefficienze dei meccanismi del kernel per la gestione delle interruzioni, e impegna il bus della memoria durante i trasferimenti di dati tra i controllori dei dispositivi e la memoria fisica, e ancora tra i buffer del kernel e lo spazio d'indirizzi delle applicazioni. Soddisfare in modo elegante tutte queste esigenze è una delle principali questioni che riguardano i progettisti di un calcolatore.

Sebbene i calcolatori moderni siano capaci di gestire molte migliaia di interruzioni al secondo, la loro gestione è un processo relativamente oneroso. Ogni interruzione fa sì che il sistema cambi stato, esegua il gestore delle interruzioni, e infine ripristini lo stato originario. Se il numero di cicli impiegato nel busy waiting non è eccessivo, l'i/o programmato può essere più efficiente di quello basato sulle interruzioni. Il completamento di un'operazione di i/o in genere implica lo sblocco di un processo, comportando così l'overhead dovuto a un cambio di contesto.

Anche il traffico di una rete può portare a un alto numero di cambi di contesto; si consideri, per esempio, il login remoto da un calcolatore a un altro. Ciascun carattere inserito in un calcolatore deve essere comunicato all'altro: quando si inserisce un carattere nel primo calcolatore, la tastiera produce un segnale d'interruzione; il carattere arriva tramite il gestore delle interruzioni al driver del dispositivo, da questo al kernel, e quindi al processo utente. Il processo utente esegue una chiamata di sistema di i/o di rete al fine di inviare il carattere al secondo calcolatore. All'interno del kernel del primo calcolatore, il carattere attraversa gli strati di protocollo necessari per la costruzione di un pacchetto di rete e giunge al driver di rete, il quale trasferisce il pacchetto al controllore della interfaccia di rete. Quest'ultimo invia il carattere e genera un'interruzione che segnala al kernel il completamento della chiamata di sistema di i/o di rete.

A questo punto, il dispositivo di rete del secondo calcolatore riceve il pacchetto, e genera un'interruzione. I protocolli di rete estraggono il carattere dal pacchetto e lo consegnano all'appropriato demone di rete. Il demone di rete individua a quale sessione di login remoto appartiene il carattere e lo passa al sottodemone che gestisce la specifica sessione. Durante questo processo avvengono cambi di contesto e di stato (Figura 12.16). Di solito il destinatario rispedisce al mittente, sotto forma di eco, una copia del carattere originario, e ciò raddoppia il lavoro necessario.

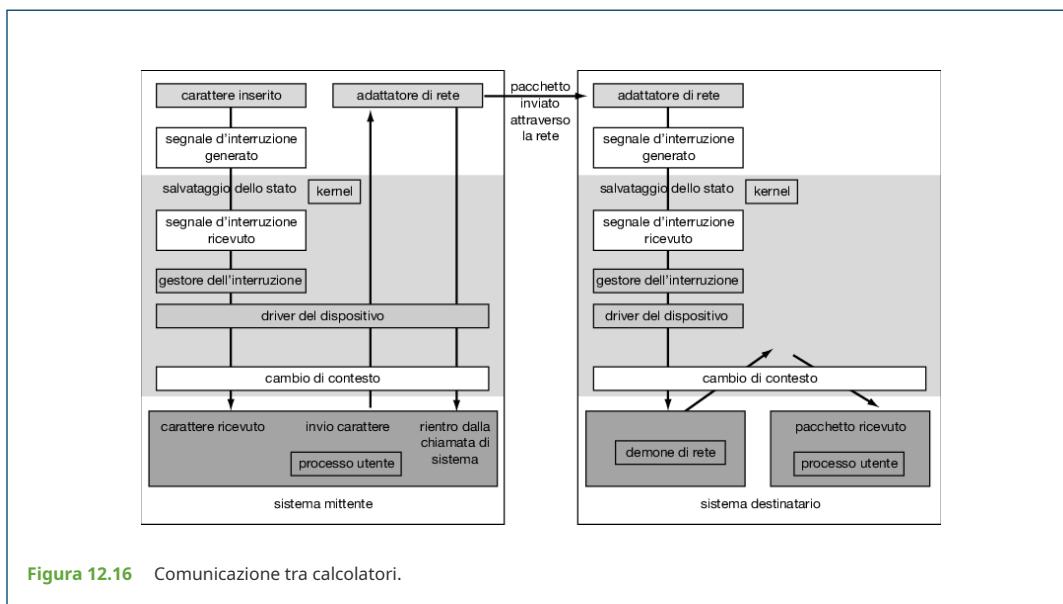


Figura 12.16 Comunicazione tra calcolatori.

Alcuni sistemi usano unità d'elaborazione specifiche per la gestione dell'i/o dei terminali, riducendo così il carico delle gestioni delle interruzioni gravante sulla cpu. Per esempio, i concentratori di terminali convogliano il traffico proveniente da centinaia di terminali su un'unica porta di un grande calcolatore. I canali di i/o sono unità d'elaborazione specializzate presenti nei mainframe e altri sistemi di alto profilo; il loro compito è sollevare la cpu di una parte del peso della gestione dell'i/o. L'idea di base è che i canali di i/o mantengano il flusso dei dati costante e uniforme, mentre la cpu rimane libera di elaborare i dati acquisiti. Come i controller dei dispositivi e i controllori dma che si trovano nei calcolatori di minori dimensioni, un canale può eseguire programmi più raffinati e generali, e può quindi essere tarato per specifici carichi di lavoro.

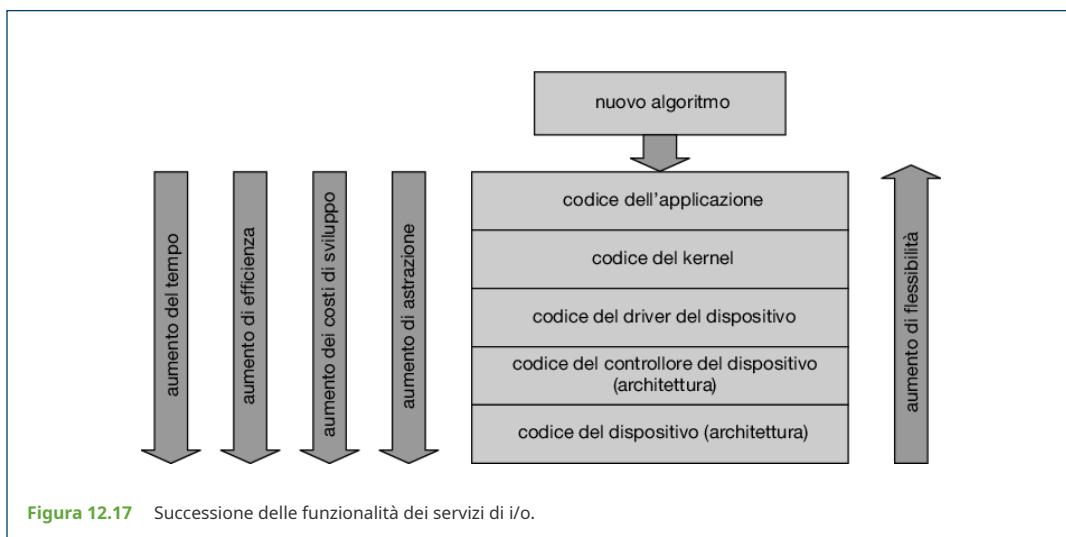
Per migliorare l'efficienza dell'i/o si possono applicare diversi principi.

- Ridurre il numero dei cambi di contesto.
- Ridurre il numero di copiature dei dati in memoria durante i trasferimenti fra dispositivi e applicazioni.

- Ridurre la frequenza delle interruzioni utilizzando il trasferimento in blocco di grandi quantità di dati, controllori intelligenti e il polling (nel caso in cui si possa minimizzare il busy waiting).
- Aumentare il tasso di concorrenza usando controllori dma intelligenti o canali di i/o per sollevare la cpu dalle semplici operazioni di copiatura di dati.
- Realizzare le primitive di elaborazione direttamente nell'hardware, così da permettere che la loro esecuzione sia simultanea alle operazioni di bus e di cpu.
- Equilibrare le prestazioni della cpu, del sottosistema di memoria, del bus e dell'i/o, giacché il sovraccarico di uno qualunque di questi settori provoca l'inutilizzo degli altri.

La complessità dei dispositivi è assai variabile: un mouse per esempio è piuttosto semplice; i suoi movimenti e la pressione sui pulsanti sono convertiti in valori numerici, passati attraverso il driver del mouse, all'applicazione. Per contro, i servizi forniti dal driver delle unità a disco del sistema operativo Windows sono assai complessi: non solo il driver gestisce singole unità a disco, ma costruisce anche array raid (si veda il Paragrafo 11.8) convertendo le richieste di lettura o scrittura di un'applicazione in un insieme coordinato di operazioni di i/o sui dischi. Il driver, inoltre, esegue sofisticati algoritmi di gestione degli errori e di recupero dei dati, e svolge diverse funzioni di ottimizzazione delle prestazioni dei dischi.

Ci si può chiedere se i servizi di i/o si debbano implementare nei dispositivi hardware, nei loro driver, o nelle applicazioni. Talvolta si può osservare (Figura 12.17) la seguente successione:



- Inizialmente, gli algoritmi sperimentali per l'i/o si codificano a livello dell'applicazione, dato che il codice dell'applicazione è flessibile ed è difficile che errori di programmazione causino l'arresto completo del sistema. In questo modo si evita inoltre di dover riavviare o ricaricare i driver dei dispositivi ogni volta che si modifica il codice degli algoritmi. Tuttavia, questi algoritmi sono spesso inefficienti a causa dell'overhead dovuto ai cambi di contesto necessari e dell'impossibilità di sfruttare le strutture dati del kernel e i suoi meccanismi interni (per esempio, la gestione dei messaggi, l'uso dei thread, i lock).
- Quando uno di questi algoritmi è stato messo a punto, è possibile ricodificarlo all'interno del kernel. Ciò può portare a un miglioramento delle prestazioni, ma la stesura del codice è più impegnativa, perché il kernel è un ambiente vasto e complesso. È inoltre necessario verificare accuratamente la correttezza del codice al fine di evitare alterazioni dei dati e l'arresto del sistema.
- Le prestazioni migliori si ottengono con l'integrazione delle funzioni di tali algoritmi in hardware, nei dispositivi o nei controllori. Gli svantaggi di questa tecnica comprendono la difficoltà e il costo di successive migliorie o dell'eliminazione di eventuali errori, il maggior tempo richiesto per portarne a termine la realizzazione (mesi invece di giorni), e la minore flessibilità. Per esempio, un controllore raid può essere privo di funzioni che permettono al kernel di modificare la locazione o l'ordine di lettura o scrittura di singoli blocchi, anche se il kernel potrebbe possedere informazioni particolari sul carico di lavoro che gli permetterebbero di migliorare le prestazioni dell'i/o.

Nel tempo, come avviene in altri ambiti dell'informatica, la velocità dei dispositivi di i/o è aumentata. I dispositivi nvm stanno crescendo in popolarità e nella varietà di dispositivi disponibili. La velocità raggiunta dai dispositivi nvm è molto alta, talvolta straordinaria, e i dispositivi delle future generazioni si avvicineranno alla velocità delle dram. Questi sviluppi stanno aumentando l'attenzione posta ai sottosistemi di i/o e agli algoritmi del sistema operativo in grado di sfruttare le velocità di lettura/scrittura che sono ora disponibili. La Figura 12.18 mostra cpu e dispositivi di memoria in un grafico dove le due dimensioni rappresentano la capacità e la latenza delle operazioni di i/o. Inoltre, la figura mostra una rappresentazione della latenza di rete, utile per rivelare il tributo aggiuntivo imposto dal networking in termini di prestazioni.

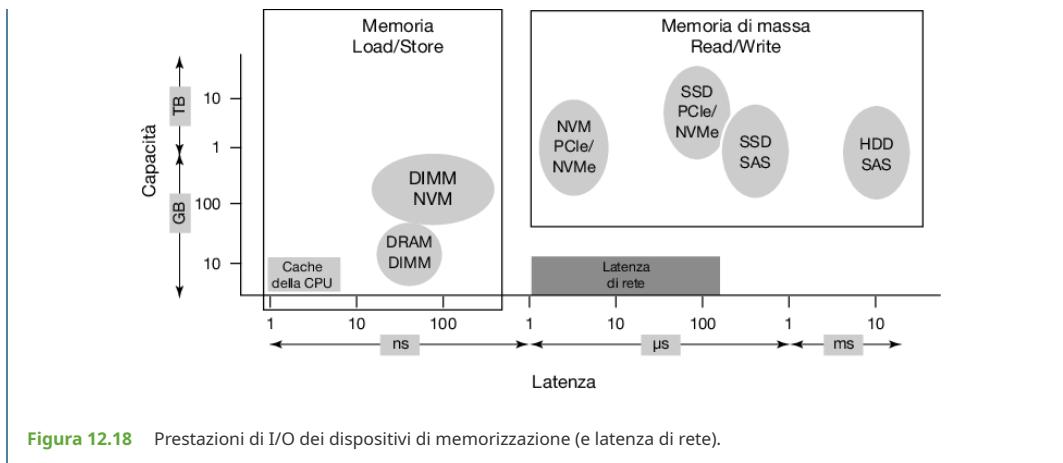


Figura 12.18 Prestazioni di I/O dei dispositivi di memorizzazione (e latenza di rete).

12.8 Sommario

- I principali elementi hardware di un calcolatore coinvolti nell'esecuzione dell'i/o sono i bus, i controllori dei dispositivi e i dispositivi stessi.
- I trasferimenti dei dati tra i dispositivi e la memoria centrale sono controllati dalla cpu nel caso di i/o programmato, altrimenti, sono demandati al controllore dma.
- Un modulo del kernel che controlla un dispositivo è detto driver. L'interfaccia fornita alle applicazioni, costituita dalle chiamate di sistema, è progettata per gestire diverse categorie fondamentali di dispositivi hardware, come i dispositivi a blocchi e a caratteri, i timer programmabili, i file mappati in memoria, le socket di rete. Di solito le chiamate di sistema bloccano il processo che le ha invocate; il kernel e le applicazioni che non devono attendere il completamento di un'operazione di i/o impiegano chiamate di sistema non bloccanti o asincrone.
- Il sottosistema di i/o del kernel fornisce numerosi servizi: fra gli altri, lo scheduling dell'i/o, il buffering, il caching, le code di spooling, la gestione degli errori e la riservazione dei dispositivi. Un altro servizio è la traduzione dei nomi, che permette di associare ai nomi simbolici di file usati dalle applicazioni i dispositivi hardware corrispondenti. Si tratta di un processo che passa attraverso diversi stadi: dalla sequenza di caratteri che rappresenta il nome a un driver specifico e all'indirizzo di un dispositivo, e da qui all'indirizzo fisico delle porte di i/o o dei controllori di bus. Tale interpretazione può avvenire nell'ambito dello spazio dei nomi del file system (come in unix), o in uno specifico spazio di nomi dei dispositivi (come nell'ms-dos).
- streams è una realizzazione e una metodologia che permettono di sviluppare in modo modulare e incrementale i driver e i protocolli di rete. Utilizzando gli stream, i driver possono essere organizzati in una catena, attraverso cui passano i dati in maniera sequenziale e bidirezionale per l'elaborazione.
- A causa dei molti strati di software presenti fra un dispositivo fisico e l'applicazione, le chiamate di sistema per l'i/o sono onerose in termini di utilizzazione della cpu. Questa struttura a strati comporta diversi overhead: per realizzare i cambi di contesto, gestire le interruzioni e i segnali usati per la comunicazione con i dispositivi, e copiare dati fra le aree di memoria per l'i/o del kernel e lo spazio d'indirizzi delle applicazioni.

Esercizi di ripasso

12.1 Individuate tre vantaggi che si ottengono dal collocare funzionalità all'interno del controllore di un dispositivo piuttosto che nel kernel. Individuate poi tre svantaggi.

12.2 L'esempio di handshaking del Paragrafo 12.2 utilizza 2 bit: un bit busy e un bit command-ready. È possibile implementare la stessa negoziazione con un solo bit? Se è possibile, descrivete il protocollo. Se non lo è, spiegate perché un bit non è sufficiente.

12.3 Perché un sistema potrebbe utilizzare i/o guidato dalle interruzioni per gestire una porta seriale singola e il polling per gestire un processore di front-end come un terminal concentrator?

12.4 Il polling per il completamento di i/o può sprecare un gran numero di cicli di cpu se il processore itera un ciclo busy-waiting molte volte prima che l'i/o sia terminato. D'altro canto, se il dispositivo di i/o è pronto per il servizio, l'interrogazione ciclica può essere molto più efficiente rispetto a rilevare e gestire un'interruzione. Descrivete una strategia ibrida per un dispositivo di i/o che combini polling, sleeping e interruzioni. Per ognuna di queste tre strategie (polling puro, interruzioni pure e ibrida) descrivete un contesto in cui quella strategia sia più efficiente delle altre.

12.5 In che modo il dma aumenta la concorrenza? In che senso complica il progetto dell'hardware?

12.6 Perché è importante aumentare la velocità del bus di sistema e delle periferiche all'aumentare della velocità della cpu?

12.7 Fate una distinzione tra un driver streams e un modulo streams.

Esercizi

12.8 Nel caso di interruzioni multiple, inviate da dispositivi diversi approssimativamente nello stesso istante, si può applicare un criterio di priorità per stabilire l'ordine di precedenza da dare alle interruzioni. Valutate gli elementi che è necessario considerare per assegnare priorità alle interruzioni.

12.9 Valutate gli aspetti positivi e negativi causati della mappatura in memoria dei registri di controllo dei dispositivi per l'i/o.

12.10 Considerate le seguenti situazioni riguardanti l'i/o in un pc monoutente:

- a. un mouse usato insieme con un'interfaccia utente grafica;
- b. un lettore di nastri in un sistema operativo multitasking (assumete l'impossibilità di preassegnare i dispositivi);
- c. un'unità a disco contenente i file dell'utente;
- d. una scheda grafica con connessione diretta tramite bus, accessibile per mezzo di i/o memory mapped.

Per ognuna di queste situazioni, dite se è opportuno progettare il sistema operativo in modo che possa impiegare la gestione del buffer, lo spooling, il caching, o una loro combinazione. Dite inoltre se è opportuno usare il polling o le interruzioni. Argomentate le risposte.

12.11 In molti sistemi multiprogrammati, i programmi utenti accedono alla memoria per mezzo degli indirizzi virtuali, mentre il sistema operativo utilizza direttamente gli indirizzi fisici. Come si riflette questo fatto sulle operazioni di i/o, nella fase di avvio da parte del programma utente e, in seguito, sulla loro esecuzione da parte del sistema?

12.12 Quali sono le diverse forme di overhead di gestione causate dal servire un'interruzione?

12.13 Descrivete tre circostanze in cui sarebbe opportuno usare l'i/o bloccante, e altre tre in cui dovrebbe essere usato l'i/o non bloccante. Riflettete sulla possibilità di realizzare semplicemente l'i/o non bloccante e lasciare che i processi interroghino ciclicamente i dispositivi richiesti finché essi sono pronti.

12.14 Di norma, quando un dispositivo completa il proprio i/o, si genera una singola interruzione, gestita dal processore nel modo appropriato. In alcuni frangenti, tuttavia, il codice da eseguire al termine del lavoro di i/o può essere suddiviso in due segmenti: il primo, eseguito subito dopo che l'i/o è completato, pianifica anche una seconda interruzione per innescare l'esecuzione del secondo segmento di codice, in un momento successivo. A quale scopo si adotta questa strategia nel progettare i gestori delle interruzioni?

12.15 Alcuni controllori dma forniscono l'accesso diretto alla memoria virtuale. In questo caso, i destinatari delle operazioni di i/o sono indirizzi virtuali, tradotti poi in indirizzi fisici durante l'accesso diretto alla memoria. Per quali versi tale funzionalità complica la progettazione del controllore dma? Quali sono i suoi vantaggi?

12.16 Il sistema unix coordina le attività dei componenti per l'i/o del kernel manipolando le strutture dati condivise interne al kernel; invece il sistema Windows utilizza lo scambio di messaggi tra i componenti del kernel, con tecniche orientate agli oggetti. Valutate tre elementi a favore e tre a sfavore di ciascuna strategia.

12.17 Scrivete in uno pseudocodice una procedura che realizzi un orologio virtuale che comprenda l'accodamento e la gestione delle richieste del timer per il kernel e le applicazioni. Assumete che l'hardware fornisca tre canali di temporizzazione.

12.18 Considerate vantaggi e svantaggi che derivano dal garantire un trasferimento dei dati affidabile tra i moduli di un'applicazione streams.

CAPITOLO 13

Interfaccia del file system

Per la maggior parte degli utenti il file system è l'aspetto più visibile di un sistema operativo. Esso fornisce il meccanismo per la memorizzazione in linea di dati e programmi appartenenti al sistema operativo e a tutti gli utenti del sistema elaborativo. Il file system consiste di due parti distinte: un insieme di *file*, ciascuno dei quali contiene i dati, e una *struttura della directory*, che organizza tutti i file nel sistema e fornisce le informazioni relative. La maggior parte dei file system risiede su dispositivi di memoria secondaria, che abbiamo descritto nel Capitolo 11 e tratteremo ancora nel prossimo capitolo. In questo capitolo considereremo i vari aspetti dei file e i principali tipi di strutture della directory. Discuteremo anche la semantica della condivisione dei file fra più processi, utenti e calcolatori. Esamineremo inoltre la gestione della *protezione dei file*, necessaria in un ambiente in cui più utenti hanno accesso ai file, e dove si vuole controllare chi e in che modo vi ha accesso.

13.1 Concetto di file

I calcolatori possono memorizzare le informazioni su diversi supporti, come dischi, nastri magnetici e dischi ottici. Per rendere agevole l'uso del calcolatore, il sistema operativo offre una visione logica uniforme delle informazioni memorizzate; fornisce un'astrazione delle caratteristiche fisiche dei propri dispositivi di memoria definendo un'unità di memorizzazione logica, il *file*. Il sistema operativo associa i file a dispositivi fisici, che di solito sono non volatili, in modo che il loro contenuto non vada perduto a causa dei riavvi del sistema.

Un file è un insieme di informazioni correlate, registrate in memoria secondaria, cui è stato assegnato un nome. Dal punto di vista dell'utente, un file è la più piccola porzione di memoria logica secondaria; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file. Di solito i file rappresentano programmi, in forma sorgente o oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari. I file possono non possedere un formato specifico, come i file di testo, oppure essere rigidamente formattati. In genere un file è formato da una sequenza di bit, byte, righe o *record* il cui significato è definito dal creatore e dall'utente del file stesso. Il concetto di file è quindi estremamente generale.

Poiché i file sono il metodo utilizzato dagli utenti e dalle applicazioni per archiviare e recuperare i dati e poiché sono così adatti a un uso generale, il loro utilizzo si è esteso oltre i confini originali. Per esempio, unix, Linux e alcuni altri sistemi operativi forniscono un file system `proc` che utilizza le interfacce del file system per fornire accesso alle informazioni di sistema (come i dettagli dei processi).

Le informazioni contenute in un file sono definite dal suo creatore e possono essere di molti tipi: programmi sorgente, programmi oggetto, dati numerici, testo, foto, musica, video, e così via. Un file ha una struttura definita secondo il tipo: un file di *testo* è formato da una sequenza di caratteri organizzati in righe, ed eventualmente pagine; un file *sorgente* è formato da una sequenza di funzioni, ciascuna delle quali è a sua volta organizzata in dichiarazioni seguite da istruzioni eseguibili; un file *eseguibile* consiste di una serie di sezioni di codice che il caricatore può caricare in memoria ed eseguire.

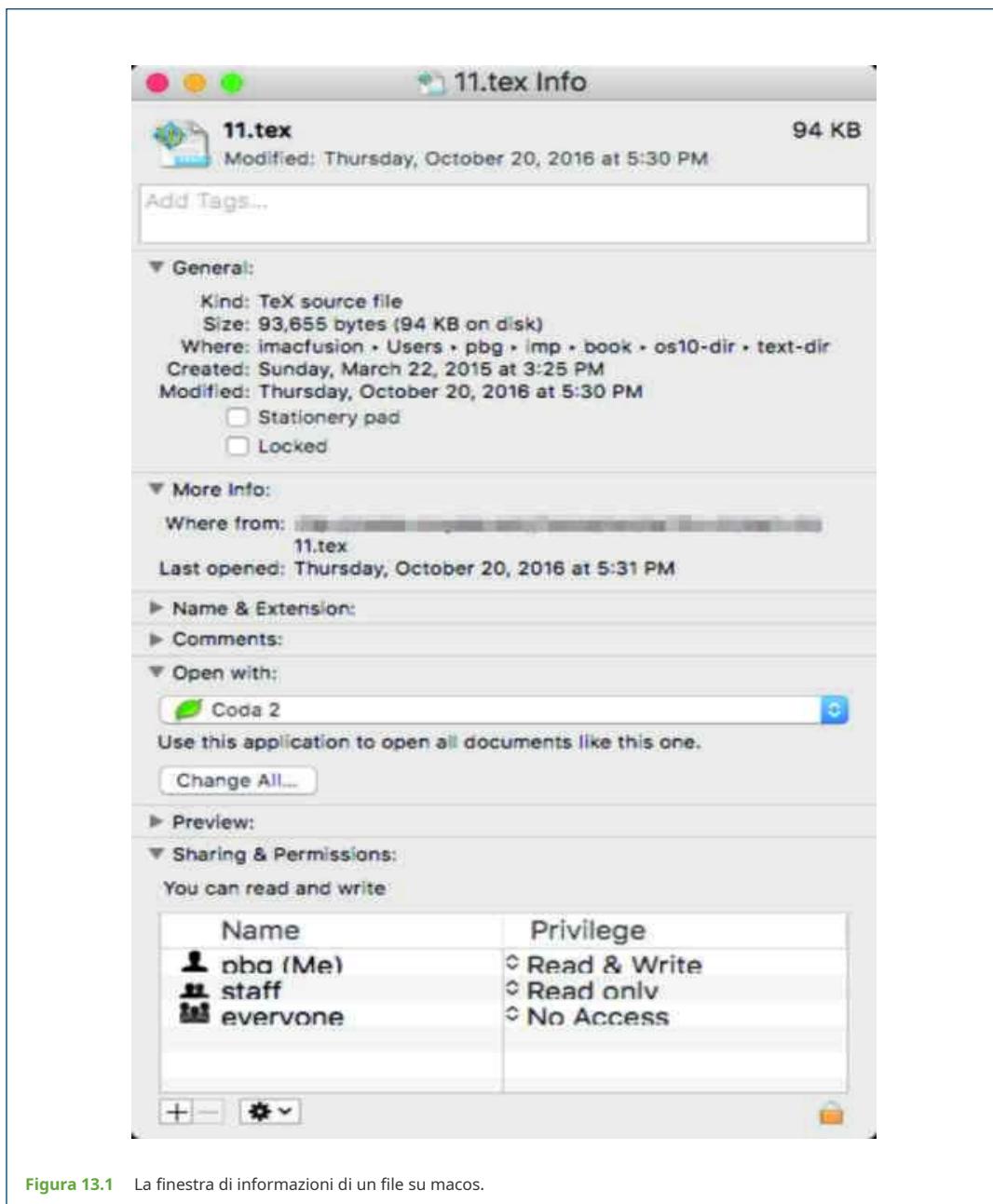
13.1.1 Attributi dei file

Per comodità degli utenti, ogni file ha un nome che si usa come riferimento. Un nome, di solito, è una sequenza di caratteri come `esempio.c`. Alcuni sistemi, per quel che riguarda i nomi, distinguono le lettere maiuscole dalle minuscole, altri le considerano equivalenti. Una volta ricevuto il nome, il file diviene indipendente dal processo, dall'utente, e anche dal sistema da cui è stato creato. Per esempio, un utente potrebbe creare il file `esempio.c` e un altro utente potrebbe modificarlo specificandone il nome. Il proprietario del file potrebbe registrare il file in una chiavetta usb, inviarlo per posta elettronica come allegato o copiarlo attraverso la rete, ed esso potrebbe ancora chiamarsi `esempio.c` nel sistema di destinazione. A meno che non ci sia un metodo di condivisione e sincronizzazione, questa seconda copia è ora indipendente dalla prima e può essere modificata separatamente.

Un file ha attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti.

- Nome. Il nome simbolico del file è l'unica informazione in forma umanamente leggibile.
- Identificatore. Si tratta di un'etichetta unica, di solito un numero, che identifica il file all'interno del file system; è il nome impiegato dal sistema per il file.
- Tipo. Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi.
- Locazione. Si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.
- Dimensione. Si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- Protezione. Le informazioni di controllo degli accessi controllano chi può leggere, scrivere o eseguire il file.
- Ora, data e identificazione dell'utente. Queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione, della sicurezza e del monitoraggio del suo utilizzo.

Alcuni file system più recenti supportano anche gli attributi estesi dei file, tra cui la codifica dei caratteri del file e funzioni di sicurezza come la checksum. La Figura 13.1 illustra una finestra di informazioni di un file su macos nella quale sono visualizzati gli attributi di un file.



Le informazioni sui file sono conservate nella struttura della directory, che risiede sullo stesso dispositivo dove sono memorizzati i file. Di solito un elemento di directory consiste di un nome di file e di un identificatore unico, che a sua volta individua gli altri attributi del file. Un elemento di directory può richiedere più di un kilobyte per contenere queste informazioni per ciascun file. In un sistema con molti file, la dimensione della stessa directory può essere dell'ordine dei megabyte o dei gigabyte. Poiché le directory, come i file, devono essere non volatili, si devono registrare sul dispositivo di memorizzazione di massa e caricare in memoria centrale un po' per volta, secondo le necessità.

13.1.2 Operazioni sui file

Un file è un tipo di dato astratto. Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file. Esaminiamo ciò che deve fare un sistema operativo per ciascuna di queste sei operazioni di base. Ciò dovrebbe aiutare a vedere come si possano realizzare altre operazioni simili, per esempio la ridefinizione di un file.

- Creazione di un file. Per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system; l'allocazione dello spazio per i file è rimandata al Capitolo 14. Inoltre, per il file si deve creare un nuovo elemento nella directory.
- Scrittura di un file. Per scrivere in un file viene effettuata una chiamata di sistema che specifica il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema ricerca la directory per individuare la posizione del file. Il file system deve mantenere un *puntatore di scrittura* alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogniqualvolta si esegue una scrittura.
- Lettura di un file. Per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il blocco del file da leggere. Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un *puntatore di lettura* alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo o legge o scrive in un file, e la posizione corrente è mantenuta come un puntatore alla posizione corrente del file specifico del processo. Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- Riposizionamento in un file. Si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di i/o. Questa operazione è anche nota come *riposizionamento o ricerca* (seek) nel file.
- Cancellazione di un file. Per cancellare un file si cerca l'elemento della directory associato al file designato, si rilascia lo spazio associato al file (in modo che possa essere adoperato per altri) e si elimina l'elemento della directory.
- Troncamento di un file. Si potrebbe voler cancellare il contenuto di un file, ma mantenere i suoi attributi. Invece di forzare gli utenti a cancellare il file e quindi ricrearlo, questa funzione consente di mantenere immutati gli attributi (a esclusione della lunghezza del file) pur azzerando la lunghezza del file e rilasciando lo spazio occupato.

Queste sei operazioni di base comprendono l'insieme minimo delle operazioni richieste per i file. Altre operazioni comuni comprendono l'*aggiunta* (Appending) di nuove informazioni alla fine di un file esistente e la *ridenominazione* di un file esistente. Queste operazioni primitive si possono combinare per compiere altre operazioni. Per esempio, per creare una *copia* di un file, o per copiare il file in un altro dispositivo di i/o, come una stampante o un video, è sufficiente creare un nuovo file, leggere i dati dal file vecchio e scriverli nel nuovo. Sono inoltre necessarie operazioni che consentano a un utente di leggere e impostare i vari attributi di un file. Per esempio, si potrebbe voler un'operazione che consenta all'utente di determinare lo stato di un file, come la lunghezza, e che consenta di definire gli attributi di un file, come il proprietario.

La maggior parte delle operazioni sopra citate richiede una ricerca nella directory dell'elemento associato al file specificato. Per evitare questa continua ricerca, molti sistemi richiedono l'impiego di una chiamata di sistema `open()` prima che un file venga utilizzato. Il sistema operativo mantiene una tabella contenente informazioni riguardanti tutti i file aperti (detta, per l'appunto, tabella dei file aperti). Quando si richiede un'operazione su un file, questo viene individuato tramite un indice in tale tabella, in questo modo si evita qualsiasi ricerca. Quando il file non è più attivamente usato viene *chiuso* dal processo, e il sistema operativo rimuove l'elemento a esso associato dalla tabella dei file aperti. Le chiamate di sistema che lavorano su file chiusi invece che aperti sono `create()` e `delete()`.

Alcuni sistemi aprono implicitamente un file al primo riferimento e lo chiudono automaticamente quando il processo che lo ha aperto termina. La maggior parte dei sistemi invece esige che il programmatore richieda l'apertura del file in modo esplicito per mezzo di una chiamata di sistema `open()` prima che sia possibile adoperarlo. L'operazione `open()` riceve il nome del file, lo cerca nella directory e copia l'elemento della directory a esso associato nella tabella dei file aperti. La chiamata di sistema `open()` può accettare anche informazioni sui modi d'accesso: creazione, sola lettura, lettura e scrittura, sola aggiunta, ecc. Si controllano i permessi relativi al file, e se la modalità d'accesso richiesta è consentita, si apre il file per il processo. La chiamata di sistema `open()` riporta di solito un puntatore all'elemento nella tabella dei file aperti; questo puntatore si adopera al posto dell'effettivo nome del file in tutte le operazioni di i/o, evitando così successive operazioni di ricerca e semplificando l'interfaccia delle chiamate di sistema.

La realizzazione delle operazioni `open()` e `close()` è più complicata in un ambiente multiente dove più processi possono aprire un file contemporaneamente. Di solito il sistema operativo introduce due livelli di tabelle interne: una tabella per ciascun processo e una tabella di sistema. La tabella del processo contiene i riferimenti a tutti i file aperti da quel processo. In questa tabella sono memorizzate le informazioni sull'uso del file da parte del processo; per esempio, si trovano in questa tabella il puntatore alla posizione corrente per ciascun file e le informazioni sui diritti d'accesso ai file.

Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una tabella di sistema dei file aperti, contenente le informazioni indipendenti dai processi come la posizione dei file nei dischi, le date degli accessi e le dimensioni dei file. Quando un file è stato aperto da un processo, la tabella dei file aperti del sistema contiene un elemento relativo al file; una `open()` eseguita da un altro processo comporta solamente l'aggiunta di un nuovo elemento nella tabella dei file aperti associata al processo, che punta al corrispondente elemento della tabella di sistema. In genere, la tabella dei file aperti ha anche un *contatore delle aperture* associato a ciascun file, indicante il numero di processi che hanno aperto quel file. Ogni `close()` decrementa questo *contatore*; quando raggiunge il valore zero il file non è più in uso e si elimina l'elemento corrispondente dalla tabella dei file aperti.

Riassumendo, a ciascun file aperto sono associate le diverse seguenti informazioni.

- Puntatore al file. Nei sistemi che non prevedono un offset come parametro delle chiamate di sistema `read()` e `write()`, il sistema deve tener traccia dell'ultima posizione di lettura e scrittura sotto forma di un puntatore alla posizione corrente nel file. Questo puntatore è unico per ogni processo che opera sul file e quindi deve essere tenuto separato dagli attributi del file residenti nel disco.
- Contatore dei file aperti. Man mano che si chiudono i file, per evitare di esaurire lo spazio associato alla propria tabella dei file aperti, il sistema operativo deve riutilizzarne gli elementi. Poiché più processi possono aprire uno stesso file, prima di rimuovere l'elemento corrispondente, il sistema deve attendere l'ultima chiusura del file. Questo contatore tiene traccia del numero di `open()` e `close()`, e raggiunge il valore zero dopo l'ultima chiusura, momento in cui il sistema può rimuovere l'elemento della tabella.
- Posizione nel disco del file. La maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file (ovunque si trovi, sia esso su una memoria di massa, su un file server in rete, o su un'unità ram) è mantenuta in memoria, per evitare di doverla prelevare dal disco a ogni operazione.
- Diritti d'accesso. Ciascun processo apre un file in una delle modalità d'accesso. Questa informazione è contenuta nella tabella del processo in modo che il sistema operativo possa permettere o negare le successive richieste di i/o.

Alcuni sistemi operativi offrono la possibilità di applicare lock a un file aperto (o a parti di esso). Quando un processo intende proteggere un file dall'accesso concorrente di altri processi, si serve dei lock. L'utilità dei lock dei file emerge nel caso di file condivisi da diversi processi: un file di log, per esempio, può subire modifiche da parte di molti processi nel sistema.

I lock dei file sono basati su una funzionalità simile ai lock di lettura-scrittura (Paragrafo 7.1.2). Un lock condiviso è assimilabile, per funzionamento, ai lock di lettura: entrambi consentono a più processi concorrenti di appropriarsene. Un lock esclusivo mostra invece analogie con i lock di scrittura, perché un solo processo per volta può acquisire questo tipo di lock. Si noti bene che non in tutti i sistemi operativi forniscono entrambi i tipi di lock; alcuni sistemi forniscono solamente lock esclusivi dei file.

Inoltre, il sistema operativo può fornire meccanismi di lock dei file obbligatori (*mandatory*), oppure consultivi (*advisory*). Se un lock è obbligatorio, il sistema operativo impedirà a qualunque altro processo di accedere al file interessato una volta che il suo lock sia stato acquisito. Poniamo, per esempio, che un processo acquisisca un lock esclusivo del file `system.log`. Se un altro processo – per esempio, un editor – tentasse di aprire `system.log`, il sistema operativo negherebbe l'accesso finché il lock esclusivo ritorni disponibile. Ciò accade anche se l'editor non è esplicitamente programmato per acquisire il lock. Qualora invece il lock è consultivo, il sistema operativo non impedirà l'accesso dell'editor a `system.log`. Tuttavia, per poter accedere al file, l'editor deve essere scritto in modo tale da acquisire esplicitamente il lock. In altri termini, se il lock è obbligatorio, il sistema operativo assicura l'integrità dei dati soggetti a lock; se il lock è consultivo, è compito dei programmatore garantire la corretta acquisizione e cessione dei lock. In linea generale, i sistemi operativi Windows adottano i lock obbligatori, mentre i sistemi unix impiegano i lock consultivi.

L'uso dei lock dei file richiede l'osservazione delle stesse accortezze della sincronizzazione dei processi. Per esempio, i programmatore impegnati a sviluppare su sistemi con lock obbligatori devono prestare attenzione a detenere i lock esclusivi solo per l'effettiva durata degli accessi ai file; in caso contrario, bloccheranno anche gli accessi da parte di altri processi. Occorre, inoltre, attuare misure appropriate al fine di evitare che due o più processi entrino in stallo nel tentativo di acquisire i lock per i file.

Figura 13.2 TECNICA DI LOCK NEL LINGUAGGIO JAVA

L'acquisizione del lock di un file tramite la api Java prevede in primo luogo l'acquisizione del `FileChannel` relativo al file; il metodo `lock()` del `FileChannel` permette poi di ottenere il lock. Il prototipo del metodo `lock()` è:

```
FileLock lock(long begin, long end, boolean shared)
```

dove `begin` ed `end` sono il punto iniziale e finale della parte da sottoporre a lock. Se `shared` vale `true`, si ottiene un lock condiviso; altrimenti, un lock esclusivo. Il lock si rilascia tramite il metodo `release()` invocato sull'oggetto `FileLock` restituito da `lock()`.

Il programma della Figura 13.2 illustra questa tecnica. Esso acquisisce due lock del file `file.txt`. La prima metà del file è soggetta a lock esclusivo, la seconda a lock condiviso.

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {

    public static final boolean EXCLUSIVE = false;

    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;
```

```
try {

RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

// acquisisce il canale per il file

FileChannel ch = raf.getChannel();

// acquisisce lock esclusivo per la prima metà del file

exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);

/* Modifica i dati . . . */

// rilascia il lock

exclusiveLock.release();

// acquisisce lock condiviso per la seconda metà del file

sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);

/* Legge i dati . . . */

// rilascia il lock

sharedLock.release();

} catch (java.io.IOException ioe) {

System.err.println(ioe);

}

finally {

if (exclusiveLock != null)

exclusiveLock.release();

if (sharedLock != null)

sharedLock.release();

}

}
```

Esempio di applicazione di lock a un file in Java.

13.1.3 Tipi di file

Nella progettazione di un file system, ma anche dell'intero sistema operativo, si deve sempre considerare la possibilità o meno che quest'ultimo riconosca e gestisca i tipi di file. Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole. Per esempio, un errore abbastanza comune consiste nel tentativo, da parte degli utenti, di stampare un programma oggetto in forma binaria; di solito questo tentativo porta semplicemente a uno spreco di carta, ma si potrebbe impedire se il sistema operativo fosse informato del fatto che il file è un programma oggetto in forma binaria.

Una tecnica comune per realizzare la gestione dei tipi di file consiste nell'includere il tipo nel nome del file. Il nome è suddiviso in due parti, un nome e un'*estensione*, di solito separate da un punto (Figura 13.3); in questo modo l'utente e il sistema operativo

possono risalire al tipo del file semplicemente esaminandone il nome. La maggior parte dei sistemi operativi permette agli utenti di specificare i nomi dei file come sequenze di caratteri seguite da un punto e concluse da un'estensione formata da caratteri aggiuntivi. Esempi di nomi di file sono `resume.docx`, `server.c` e `ReaderThread.cpp`. Il sistema usa l'estensione per stabilire il *tipo* del file e le operazioni che si possono eseguire su tale file. Per esempio, solamente i file con estensione `.com`, `.exe` o `.sh` sono eseguibili; i file con estensione `.com` e `.exe` sono due formati di file eseguibili, mentre i file con estensione `.sh` sono script di shell contenenti una sequenza di comandi, scritti in formato ascii, diretti al sistema operativo. Anche le applicazioni utilizzano estensioni per indicare i tipi di file di proprio interesse. Per esempio, i compilatori Java si aspettano sorgenti con un'estensione `.java` e l'elaboratore di testi Microsoft Word si aspetta file con estensione `.doc` o `.docx`.

Tipo di file	Estensione usuale	Funzione
Esegibile	exe, com, bin, o nessuna	Programma eseguibile, in linguaggio macchina
Oggetto	obj, o	Compilato, in linguaggio di macchina, non linkato
Codice sorgente	c, cc, java, perl, asm	Codice sorgente in vari linguaggi di programmazione
Batch	bat, sh	Comandi per l'interprete dei comandi
Markup	xml, html, tex	Dati testuali, documenti
Word processor	xml, rtf, docx	Vari formati di word processor
Libreria	lib, a, so, dll	Librerie di procedure per la programmazione
Stampa o visualizzazione	gif, pdf, jpg	File ASCII o binari in formato per la stampa o la visualizzazione
Archivio	rar, zip, tar	File contenenti più file tra loro correlati, talvolta compressi, per archiviazione o memorizzazione
Multimediali	mpeg, mov, mp3, mp4, avi	File binari contenenti informazioni audio o A/V

Figura 13.3 Comuni tipi di file.

Queste estensioni non sono sempre necessarie, ma la loro presenza consente a un utente di ridurre il numero dei caratteri specificando il nome del file senza estensione e lasciando all'applicazione il compito di cercare il file con il nome impostato e l'estensione attesa. Poiché queste estensioni non sono gestite dal sistema operativo, si possono considerare un suggerimento rivolto alle applicazioni che operano su di loro.

Consideriamo, inoltre, il sistema operativo macos. In questo sistema ciascun file ha un tipo, per esempio `.app` per le applicazioni. Ciascun file possiede anche un attributo di creazione contenente il nome del programma che lo ha creato. Questo attributo è impostato dal sistema operativo durante la chiamata di sistema `create()`, quindi il suo utilizzo è forzato e supportato dal sistema operativo. Per esempio, un file prodotto da un elaboratore di testi avrà il nome dell'elaboratore di testi come attributo di creazione. Quando un utente apre il file, con un doppio clic del mouse sull'icona che lo rappresenta, si attiva automaticamente l'elaboratore di testi che apre il file, pronto per essere letto e modificato.

Il sistema operativo unix si limita a memorizzare un semplice codice (noto come magic number) all'inizio di alcuni file binari allo scopo di indicare il tipo di dati nel file (per esempio, il formato di un file immagine). Allo stesso modo, utilizza i magic cookie all'inizio dei file di testo per indicare il tipo di file (in quale linguaggio di shell è scritto uno script) e così via (per maggiori dettagli sui magic cookie e su altro gergo informatico si veda <http://www.catb.org/esr/jargon/>). Non tutti i file possiedono tale codice, quindi il sistema non può affidarsi unicamente a questo tipo d'informazione; inoltre, unix non memorizza il nome del programma che ha creato il file. unix consente di sfruttare le estensioni come suggerimento del tipo di file; queste non vengono però imposte né vengono utilizzate dal sistema operativo; il loro compito consiste principalmente nell'aiutare gli utenti a riconoscere il tipo di contenuto del file. Un'applicazione può usare o ignorare le estensioni; dipende dalle scelte dei programmatore.

13.1.4 Struttura dei file

I tipi di file si possono anche adoperare per indicare la struttura interna dei file. I file sorgente e i file oggetto hanno una struttura corrispondente a ciò che il programma che dovrà leggerli si attende. Inoltre alcuni file devono rispettare una determinata struttura comprensibile al sistema operativo. Per esempio, il sistema operativo richiede che un file eseguibile abbia una struttura specifica che consenta di determinare dove caricare il file in memoria e quale sia la locazione della prima istruzione. Alcuni sistemi operativi estendono questa idea a un insieme di strutture di file supportate dal sistema, con un insieme di operazioni specifiche per la manipolazione dei file con queste strutture.

L'analisi precedente ci porta a uno degli svantaggi dei sistemi operativi che gestiscono più strutture di file: la dimensione risultante del sistema operativo è rilevante. Se definisce cinque strutture di file differenti, il sistema operativo deve contenere il codice per gestirle tutte. Inoltre qualsiasi file potrebbe dover essere definito come uno dei tipi gestiti dal sistema operativo: ciò provoca notevoli problemi se nuove applicazioni richiedono una strutturazione dei propri dati in modi non previsti dal sistema operativo.

Per esempio, si supponga che un sistema operativo preveda due tipi di file: file di testo (composti da righe di caratteri ascii separate da caratteri di ritorno del carrello e avanzamento di riga) e file binari eseguibili. Un utente che volesse definire un file cifrato per proteggere i propri dati da lettura non autorizzata potrebbe scoprire che nessuna delle due strutture si adatta al problema: non è un file di righe di testo ascii, ma un insieme di bit (apparentemente casuali), e sebbene possa sembrare un file binario, non è eseguibile. Queste limitazioni impongono all'utente di bypassare o usare in modo scorretto il meccanismo dei tipi di file definito dal sistema operativo, oppure di abbandonare lo schema di codifica.

Alcuni sistemi operativi impongono (e supportano) un numero minimo di strutture di file. Questo orientamento è stato seguito da unix, Windows e altri. unix considera ciascun file come una sequenza di byte, senza alcuna interpretazione da parte del sistema operativo. Questo schema garantisce la massima flessibilità, ma il minimo supporto. Qualsiasi programma applicativo deve contenere il proprio codice per interpretare in modo appropriato la struttura di un file in ingresso. A ogni modo, per poter caricare ed eseguire i programmi, tutti i sistemi operativi devono prevedere almeno un tipo di struttura, quella dei file eseguibili.

13.1.5 Struttura interna dei file

Per il sistema operativo la localizzazione di un offset all'interno di un file può essere complicata. I dischi hanno una dimensione dei blocchi ben definita, determinata dalla dimensione di un settore. Tutti gli i/o su disco si eseguono in unità di un blocco (record fisico), e tutti i blocchi hanno la stessa dimensione. È improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico desiderato, che può anche essere variabile. Una soluzione diffusa per questo tipo di problema consiste nell'impaccamento di un certo numero di record logici in blocchi fisici.

Il sistema operativo unix, per esempio, definisce tutti i file semplicemente come un flusso di byte. A ciascun byte si può accedere in modo individuale tramite il suo offset a partire dall'inizio, o dalla fine, del file. In questo caso il record logico è un byte. Il file system *impacca* automaticamente i byte in blocchi fisici (per esempio 512 byte per blocco) com'è necessario.

La dimensione dei record logici, quella dei blocchi fisici e la tecnica d'impaccamento determinano il numero dei record logici all'interno di ogni blocco fisico. L'impaccamento può essere fatto dal programma applicativo dell'utente oppure dal sistema operativo.

In entrambi i casi il file si può considerare come una sequenza di blocchi. Tutte le funzioni di i/o di base operano in termini di blocchi. La conversione da record logici a blocchi fisici è un problema di programmazione relativamente semplice.

Poiché lo spazio del disco è sempre assegnato in blocchi, una parte dell'ultimo blocco di ogni file in genere è sprecata. Se ogni blocco è composto di 512 byte, a un file di 1949 byte si assegnano quattro blocchi (2048 byte); gli ultimi 99 byte sono sprecati. I byte sprecati, a causa della gestione in multipli di blocchi invece che di byte, costituiscono la frammentazione interna. Tutti i file system ne soffrono; maggiore è la dimensione dei blocchi, maggiore sarà la frammentazione interna.

13.2 Metodi d'accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria. Esistono molti metodi per accedere alle informazioni dei file; alcuni sistemi consentono un solo metodo d'accesso ai file, mentre altri offrono diversi metodi d'accesso: in questo caso la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.

13.2.1 Accesso sequenziale

Il più semplice metodo d'accesso è l'accesso sequenziale: le informazioni del file si elaborano ordinatamente, un record dopo l'altro; questo metodo d'accesso è di gran lunga il più comune, ed è usato, per esempio, dagli editor e dai compilatori.

Le più comuni operazioni che si compiono sui file sono le letture e le scritture: un'operazione di lettura – `read_next()` – legge la prossima porzione di file e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di i/o; analogamente, un'operazione di scrittura – `write_next()` – fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. Un file siffatto si può reimpostare sull'inizio e, in alcuni sistemi, un programma può riuscire ad andare avanti o indietro di n record, con n intero (in alcuni casi solo per $n = 1$). L'accesso sequenziale è illustrato nella Figura 13.4. L'accesso sequenziale utilizza per il file il modello del nastro magnetico, e funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.

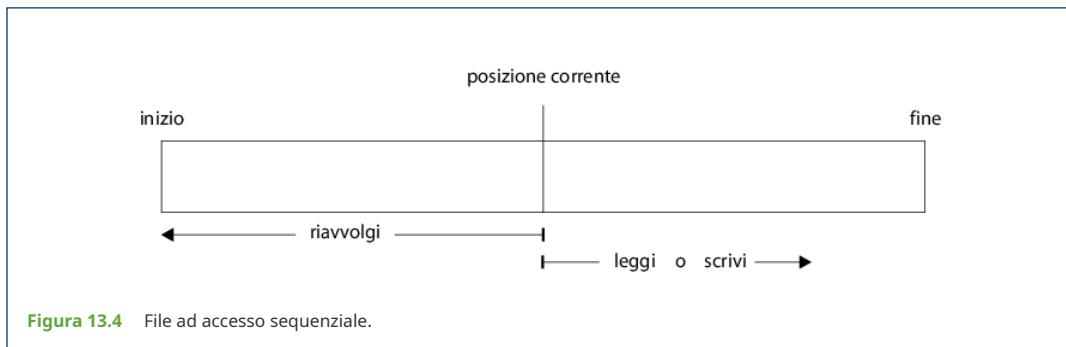


Figura 13.4 File ad accesso sequenziale.

13.2.2 Accesso diretto

Un altro metodo è l'accesso diretto (o accesso relativo). In questo caso, un file è formato da elementi logici (record) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifa al disco: i dischi permettono, infatti, l'accesso diretto a ogni blocco di file. Il file si considera come una sequenza numerata di blocchi o record: si può per esempio leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. Non esistono restrizioni all'ordine di lettura o scrittura di un file ad accesso diretto.

I file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre determinare quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste.

Per esempio, in un sistema di prenotazione dei voli, potremmo registrare tutte le informazioni su un particolare volo, per esempio il volo 713, nel blocco identificato da tale numero di volo. Quindi, il numero di posti disponibili per il volo 713 si memorizza nel blocco 713 del file di prenotazione. Per registrare informazioni riguardanti un gruppo più grande, per esempio una popolazione, si può eseguire la ricerca calcolando una funzione hash sui nomi delle persone, oppure accedere a un piccolo indice residente in memoria per determinare il blocco da leggere e scandire.

Per il metodo ad accesso diretto, si devono modificare le operazioni sui file per inserire il numero del blocco in forma di parametro. Quindi, si hanno `read(n)`, dove n è il numero del blocco, al posto di `read_next()`, e `write(n)`, invece che `write_next()`. Un metodo alternativo prevede di mantenere `read_next()` e `write_next()`, come nell'accesso sequenziale, e di aggiungere un'operazione `position_file(n)`, dove n è il numero del blocco. Quindi un'operazione `read(n)` corrisponde a una `position_file(n)` e una `read_next()`.

Il numero del blocco fornito dall'utente al sistema operativo è normalmente un numero di blocco relativo. Si tratta di un indice relativo all'inizio del file, quindi il primo blocco relativo del file è 0, il successivo è 1 e così via, anche se l'indirizzo assoluto nel disco del blocco può essere 14703 per il primo blocco e 3192 per il secondo. L'uso dei numeri di blocco relativi permette al sistema operativo di decidere dove posizionare il file (si tratta del *problema dell'allocazione* trattato nel Capitolo 14) e aiuta a impedire che l'utente acceda a porzioni del file system che possono non far parte del suo file. Alcuni sistemi iniziano la numerazione dei blocchi relativi da 0, altri da 1.

Come soddisfa il sistema operativo una richiesta di un record N in un file? Assumendo che la lunghezza del record logico sia l , una richiesta per il record N determina una richiesta di i/o per l byte a partire dalla locazione $l * N$ all'interno del file (assumendo che il primo record sia $N = 0$). Lettura, scrittura e cancellazione di un record sono rese semplici dalla sua dimensione fissa.

Non tutti i sistemi operativi gestiscono ambedue i tipi di accesso; alcuni permettono il solo accesso sequenziale, altri solo quello diretto. Alcuni sistemi richiedono che si definisca il tipo d'accesso al file al momento della sua creazione; a tale file si può accedere soltanto nel modo definito. Tuttavia si può facilmente simulare l'accesso sequenziale a un file ad accesso diretto mantenendo una variabile cp che, come illustra la Figura 13.5, definisce la nostra posizione corrente. D'altra parte è estremamente macchinoso e inefficiente simulare l'accesso diretto a un file che di per sé è ad accesso sequenziale.

Accesso sequenziale	Realizzazione nel caso di accesso diretto
reset	<code>cp = 0;</code>
read_next()	<code>read cp; cp = cp + 1;</code>
write_next()	<code>write cp; cp = cp + 1;</code>

Figura 13.5 Simulazione dell'accesso sequenziale a un file ad accesso diretto.

13.2.3 Altri metodi d'accesso

Sulla base di un metodo d'accesso diretto se ne possono costruire altri, che implicano generalmente la costruzione di un indice per il file. L'indice (*index*) contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice, e quindi usare il puntatore per accedere direttamente al file e trovare l'elemento desiderato.

Si consideri, per esempio, un file contenente prezzi al dettaglio, contenente una lista dei codici universali dei prodotti (*universal product codes*, upc), a ciascuno dei quali è associato un prezzo. Dato un record di 16 byte, questo è composto da un codice upc a 10 cifre e un prezzo a 6 cifre. Se il disco usato ha 1024 byte per blocco, in ogni blocco si possono memorizzare 64 record. Un file di 120.000 record occupa circa 2000 blocchi (2 milioni di byte). Ordinando il file secondo il codice upc si può definire un indice composto dal primo codice upc di ogni blocco. Tale indice è costituito di 2000 elementi di 10 cifre ciascuno (20.000 byte) e quindi può essere tenuto in memoria. Per trovare il prezzo di un oggetto specifico si può fare una ricerca binaria nell'indice, che permette di sapere esattamente quale blocco contiene l'elemento desiderato e quindi accedere a quel blocco. Questa struttura permette di compiere ricerche in file molto grandi limitando il numero di operazioni di i/o.

Nel caso di file molto lunghi, lo stesso file indice può diventare troppo lungo perché sia tenuto in memoria. Una soluzione a questo problema è data dalla creazione di un indice per il file indice. Il file indice principale contiene puntatori ai file indice secondari, che puntano agli effettivi elementi di dati.

Il metodo ad accesso sequenziale indicizzato di ibm (*indexed sequential access method*, isam), per esempio, usa un piccolo indice principale che punta ai blocchi del disco di un indice secondario, e i blocchi dell'indice secondario puntano ai blocchi del file effettivo. Il file è ordinato rispetto a una chiave definita. Per trovare un particolare elemento, si fa inizialmente una ricerca binaria nell'indice principale, che fornisce il numero del blocco dell'indice secondario. Questo blocco viene letto e sottoposto a una seconda ricerca binaria che individua il blocco contenente l'elemento richiesto. Infine, si fa una ricerca sequenziale sul blocco. In questo modo si può localizzare ogni elemento tramite la sua chiave con al massimo due letture ad accesso diretto. La Figura 13.6 mostra uno schema simile, com'è realizzato nel vms con indici e relativi file.

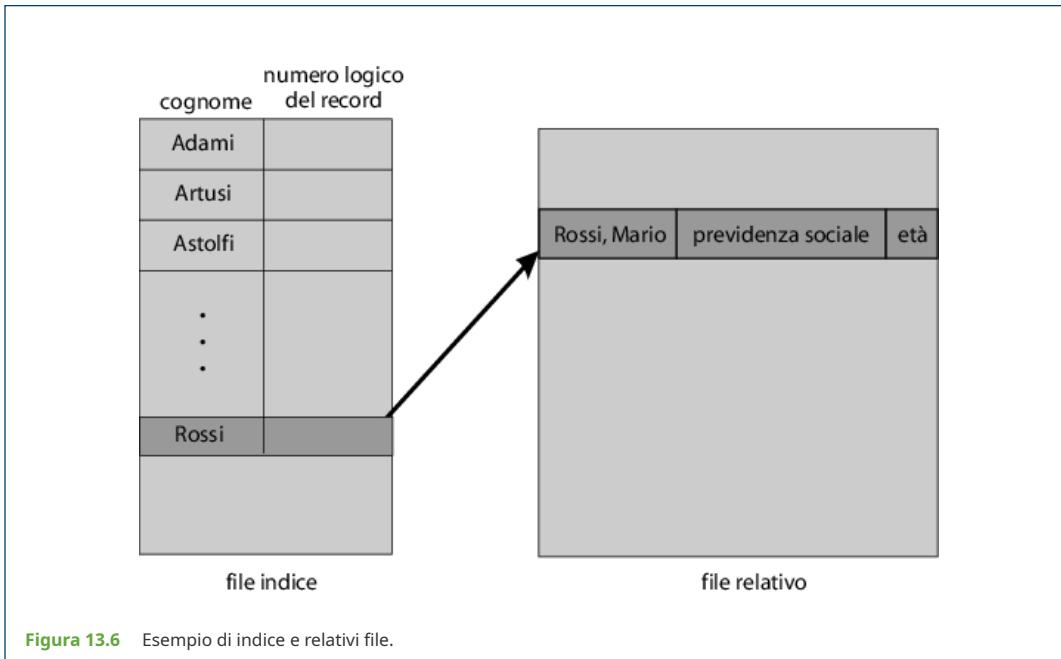


Figura 13.6 Esempio di indice e relativi file.

13.3 Struttura delle directory

La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Considerando questo punto di vista, si capisce che la stessa directory si può organizzare in molti modi diversi; l'organizzazione deve rendere possibile l'inserimento di nuovi elementi, la cancellazione di elementi esistenti, la ricerca di un elemento, e l'elenco di tutti gli elementi della directory. In questo paragrafo esamineremo diversi schemi per definire la struttura logica del sistema di directory. Nel considerare una particolare struttura della directory si deve tenere presente l'insieme delle seguenti operazioni che si possono eseguire su una directory.

- Ricerca di un file. Deve esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- Creazione di un file. Deve essere possibile creare nuovi file e aggiungerli alla directory.
- Cancellazione di un file. Quando non serve più, si deve poter rimuovere un file dalla directory.
- Elencazione di una directory. Deve esistere la possibilità di elencare tutti i file di una directory, e il contenuto degli elementi della directory associati a ciascun file nell'elenco.
- Ridenominazione di un file. Poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l'uso del file subiscono cambiamenti. La ridefinizione di un file potrebbe anche permettere la variazione della posizione del file nella directory.
- Attraversamento del file system. Si potrebbe voler accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell'intero file system a intervalli regolari. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (*backup*) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema. Inoltre, se un file non è più in uso, si può copiarlo su nastro e liberare lo spazio da esso occupato nel disco, rendendolo riutilizzabile per altri file.

Nei paragrafi seguenti sono descritti gli schemi più comuni per la definizione della struttura logica di una directory.

13.3.1 Directory a un livello

La struttura più semplice per una directory è quella a un livello. Tutti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile (Figura 13.7).

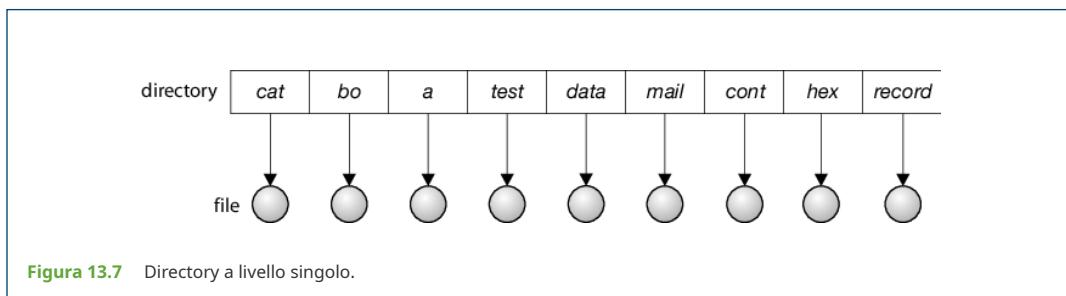


Figura 13.7 Directory a livello singolo.

Una directory a un livello presenta però limiti notevoli quando aumenta il numero dei file in essa contenuti, oppure se il sistema è usato da più utenti. Poiché si trovano tutti nella stessa directory, i file devono avere nomi unici; se due utenti attribuiscono lo stesso nome al loro file di dati, per esempio *test.txt*, si viola la regola del nome unico. Come esempio, si consideri una classe di programmazione in cui 23 studenti chiamano il programma del secondo esercizio *prog2.c* e altri 11 chiamano lo stesso programma *compito2.c*. Fortunatamente, la maggior parte dei file system supporta nomi di file di 255 caratteri, per cui è abbastanza semplice assegnare ai file un nome univoco.

Anche per un solo utente, con una directory a un livello, diventa difficile ricordare i nomi dei file con l'aumentare del loro numero. Non è affatto raro che un utente abbia centinaia di file in un calcolatore e altrettanti file in un altro sistema. In un tale ambiente, sarebbe un compito arduo dover ricordare tanti nomi di file.

13.3.2 Directory a due livelli

Come abbiamo visto, una directory a un livello spesso causa la confusione dei nomi dei file tra diversi utenti. La soluzione più ovvia prevede la creazione di una directory separata per ogni utente.

Nella struttura a due livelli, ogni utente dispone della propria directory utente (*user file directory*, ufd). Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. Quando comincia l'elaborazione di un lavoro dell'utente, oppure un utente inizia una sessione di lavoro, si fa una ricerca nella directory principale (*master file directory*, mfd) del sistema. La directory principale viene indicizzata con il nome dell'utente o il numero di account e ogni suo elemento punta alla relativa directory utente (Figura 13.8).

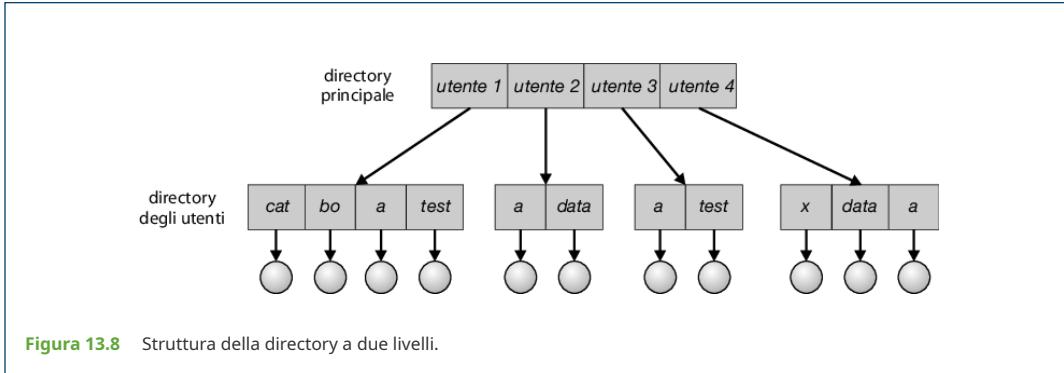


Figura 13.8 Struttura della directory a due livelli.

Quando un utente fa un riferimento a un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory utente siano unici. Per creare un file per un utente, il sistema operativo controlla che non ci sia un altro file con lo stesso nome soltanto nella directory di tale utente. Per cancellare un file il sistema operativo limita la propria ricerca alla directory utente locale, quindi non può cancellare per errore un file con lo stesso nome che appartenga a un altro utente.

Le stesse directory utente devono essere create e cancellate quando è necessario; a tale scopo si esegue uno speciale programma di sistema con nome dell'utente e informazioni relative all'account. Il programma crea una nuova directory utente e aggiunge l'elemento a essa corrispondente nella directory principale. L'esecuzione di questo programma può essere limitata all'amministratore del sistema. L'allocazione dello spazio nei dischi per le directory utente può essere gestita con le tecniche descritte per i file nel Capitolo 14.

Sebbene risolva la questione delle collisioni dei nomi, la struttura della directory a due livelli presenta ancora dei problemi. In effetti, questa struttura isola un utente dagli altri. Questo isolamento può essere un vantaggio quando gli utenti sono completamente indipendenti, ma è uno svantaggio quando più utenti vogliono cooperare e accedere ai rispettivi file. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti.

Se l'accesso è autorizzato, un utente deve avere la possibilità di riferirsi al nome di un file che si trova nella directory di un altro utente. Per attribuire un nome unico a un particolare file di una directory a due livelli, occorre indicare sia il nome dell'utente sia il nome del file. Una directory a due livelli si può pensare come un albero di altezza 2. La radice dell'albero è la directory principale, i suoi diretti discendenti sono le directory utente, da cui discendono i file che sono le foglie dell'albero. Specificando un nome utente e un nome di file si definisce un percorso che parte dalla radice (la directory principale) e arriva a una specifica foglia (il file specificato). Quindi, un nome utente e un nome di file definiscono un *nome di percorso* (*path name*). Ogni file del sistema ha un nome di percorso. Per attribuire un nome unico a un file, un utente deve conoscere il nome di percorso del file desiderato.

Se, per esempio, l'utente A desidera accedere al proprio file chiamato `prova.txt`, è sufficiente che faccia riferimento a `prova.txt`. Invece, per accedere al file denominato `prova.txt` dell'utente B, con nome di elemento della directory `utenteB`, l'utente A deve fare riferimento a `/utenteB/prova.txt`. Ogni sistema ha la propria sintassi per riferirsi ai file delle directory diverse da quella dell'utente.

Per specificare il volume cui appartiene un file occorrono ulteriori regole sintattiche. In Windows, per esempio, il volume è indicato da una lettera seguita dai due punti. Quindi l'indicazione di un file potrebbe essere del tipo `C:\utenteB\prova`. Alcuni sistemi vanno oltre e separano: volume, nome della directory, e nome del file. Nel VMS, per esempio, il file `login.com` potrebbe essere indicato come `u:[sst.jdeck]login.com;1`, dove `u` è il nome del volume, `sst` è il nome della directory, `jdeck` è il nome della sottodirectory e `1` è il numero della versione. Altri sistemi, come Unix e Linux, trattano il nome del volume semplicemente come parte del nome della directory. Il primo elemento è quello del volume, il resto è composto dalla directory e dal file. Per esempio, `/u/pbg/prova` potrebbe indicare il volume `u`, la directory `pbg` e il file `prova`.

Un caso particolare di questa situazione riguarda i file di sistema. I programmi forniti come elementi integranti del sistema, come loader, assemblatori, compilatori, utilità di sistema, librerie e così via, sono infatti definiti come file. Quando si impartiscono al sistema operativo i comandi appropriati, il caricatore legge questi file che poi vengono eseguiti. Molti interpreti di comandi operano semplicemente trattando questo comando come il nome di un file da caricare ed eseguire. Nel sistema di directory che abbiamo definito, questo nome di file viene cercato nella directory utente locale. Una soluzione potrebbe essere la copiatura dei file di sistema in ciascuna directory utente. Tuttavia, con la copiatura di tutti i file di sistema si spreca un'enorme quantità di spazio. Se i file di sistema occupano 5 mb, con 12 utenti si avrebbe un'occupazione di spazio pari a $5 \times 12 = 60$ mb, solo per le copie dei file di sistema.

La soluzione standard prevede una leggera complicazione della procedura di ricerca: si definisce una speciale directory utente contenente i file di sistema, per esempio la directory utente 0. Ogni volta che si indica un file da caricare, il sistema operativo lo cerca innanzitutto nella directory utente locale, e, se lo trova, lo usa; se non lo trova, il sistema cerca automaticamente nella speciale directory utente contenente i file di sistema. La sequenza delle directory in cui è stata fatta la ricerca avviata dal riferimento a un file è detta percorso di ricerca (*search path*). Tale percorso si può estendere in modo da contenere una lista illimitata di directory in cui fare le ricerche quando si dà il nome di un comando. Questo metodo è il più usato in Unix e Windows. Alcuni sistemi prevedono anche che ogni utente disponga del proprio percorso di ricerca personale.

13.3.3 Directory con struttura ad albero

La corrispondenza strutturale tra directory a due livelli e albero a due livelli può essere facilmente generalizzata, estendendo la struttura della directory a un albero di altezza arbitraria (Figura 13.9). Questa generalizzazione permette agli utenti di creare proprie

sottodirectory e di organizzare i file di conseguenza. Un albero è il più comune tipo di struttura delle directory. L'albero ha una directory radice (*root directory*), e ogni file del sistema ha un unico nome di percorso.

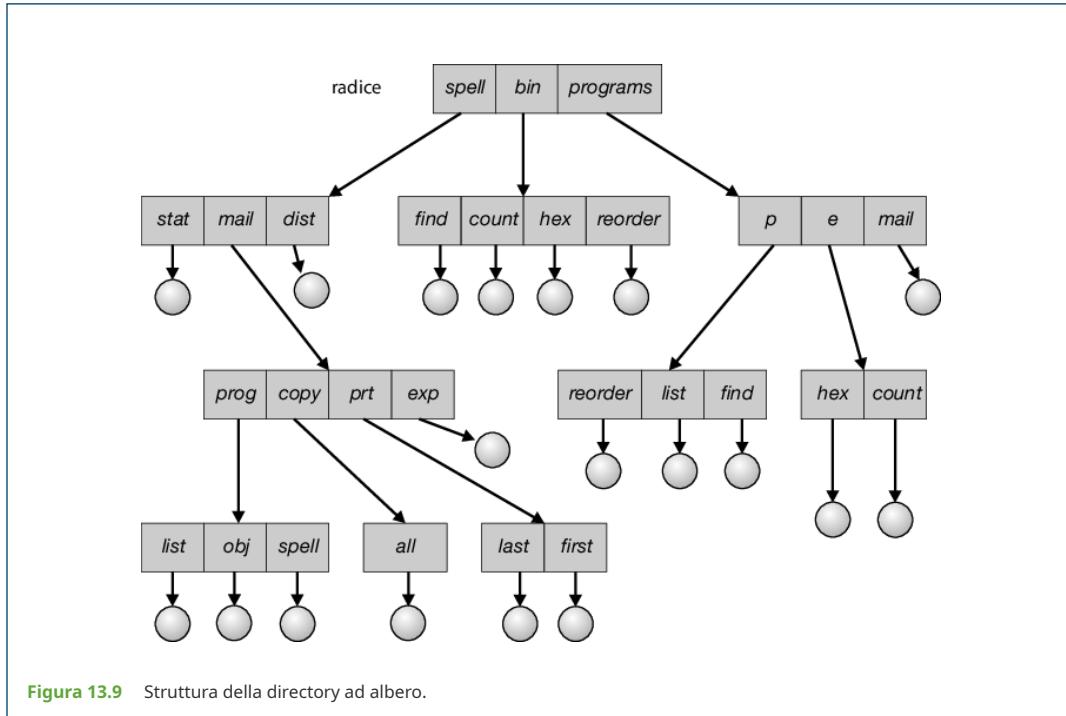


Figura 13.9 Struttura della directory ad albero.

Una directory, o una sottodirectory, contiene un insieme di file o sottodirectory. Le directory sono semplicemente file, trattati però in modo speciale. Tutte le directory hanno lo stesso formato interno. La distinzione tra file e directory è data da un bit di ogni elemento della directory. Per creare e cancellare le directory si adoperano speciali chiamate di sistema.

Normalmente, ogni utente dispone di una directory corrente. La directory corrente dovrebbe contenere la maggior parte dei file di interesse corrente per il processo. Quando si fa un riferimento a un file, si esegue una ricerca nella directory corrente; se il file non si trova in tale directory, l'utente deve specificare un nome di percorso oppure cambiare la directory corrente facendo diventare tale la directory contenente il file desiderato. Per cambiare directory corrente si fa uso di una chiamata di sistema che prende un nome di directory come parametro e lo usa per ridefinire la directory corrente. Quindi, l'utente può cambiare la propria directory corrente ogni volta che lo desidera. Da una chiamata di sistema `change_directory()` alla successiva, tutte le chiamate di sistema `open()` cercano i file specificati nella directory corrente. Si noti che il percorso di ricerca può contenere o meno uno speciale elemento che rappresenta "la directory corrente".

La directory corrente iniziale di un utente è stabilita quando viene lanciato un job dell'utente, oppure quando questi inizia una sessione di lavoro; il sistema operativo cerca nel file di accounting, o in qualche altra locazione predefinita, l'elemento relativo a questo utente. Nel file di accounting è memorizzato un puntatore (oppure il nome) della directory iniziale dell'utente. Tale puntatore viene copiato in una variabile locale per l'utente che specifica la sua directory corrente iniziale. Dalla shell dell'utente si possono poi avviare altri processi: la loro directory corrente è solitamente la directory corrente del processo genitore al momento della creazione del figlio.

I nomi di percorso possono essere di due tipi: nomi di percorso assoluti e nomi di percorso relativi. Un *nome di percorso assoluto* comincia dalla radice dell'albero di directory e segue un percorso che lo porta fino al file specificato indicando i nomi delle directory lungo il percorso. Un *nome di percorso relativo* definisce un percorso che parte dalla directory corrente.

Per esempio, nel file system ad albero della Figura 13.9, se la directory corrente è `root/spell/mail`, il nome di percorso relativo `prt/first` si riferisce allo stesso file indicato dal percorso assoluto `root/spell/mail/prt/first`.

Se si permette all'utente di definire le proprie sottodirectory, gli si consente anche di dare una struttura ai suoi file. Questa struttura può presentare directory distinte per file associati a soggetti diversi; per esempio, si può creare una sottodirectory contenente il testo di questo libro, oppure diversi tipi di informazioni, per esempio la directory `programmi` può contenere programmi sorgente; la directory `bin` può contenere tutti i programmi eseguibili.

Una decisione importante relativa alla strutturazione ad albero delle directory riguarda il modo di gestire la cancellazione di una directory. Se una directory è vuota, è sufficiente cancellare l'elemento che la designa nella directory che la contiene. Tuttavia se la directory da cancellare non è vuota, ma contiene file oppure sottodirectory, è possibile procedere in due modi. Alcuni sistemi non cancellano una directory a meno che non sia vuota; per cancellarla l'utente deve prima cancellare i file in essa contenuti. Se esiste

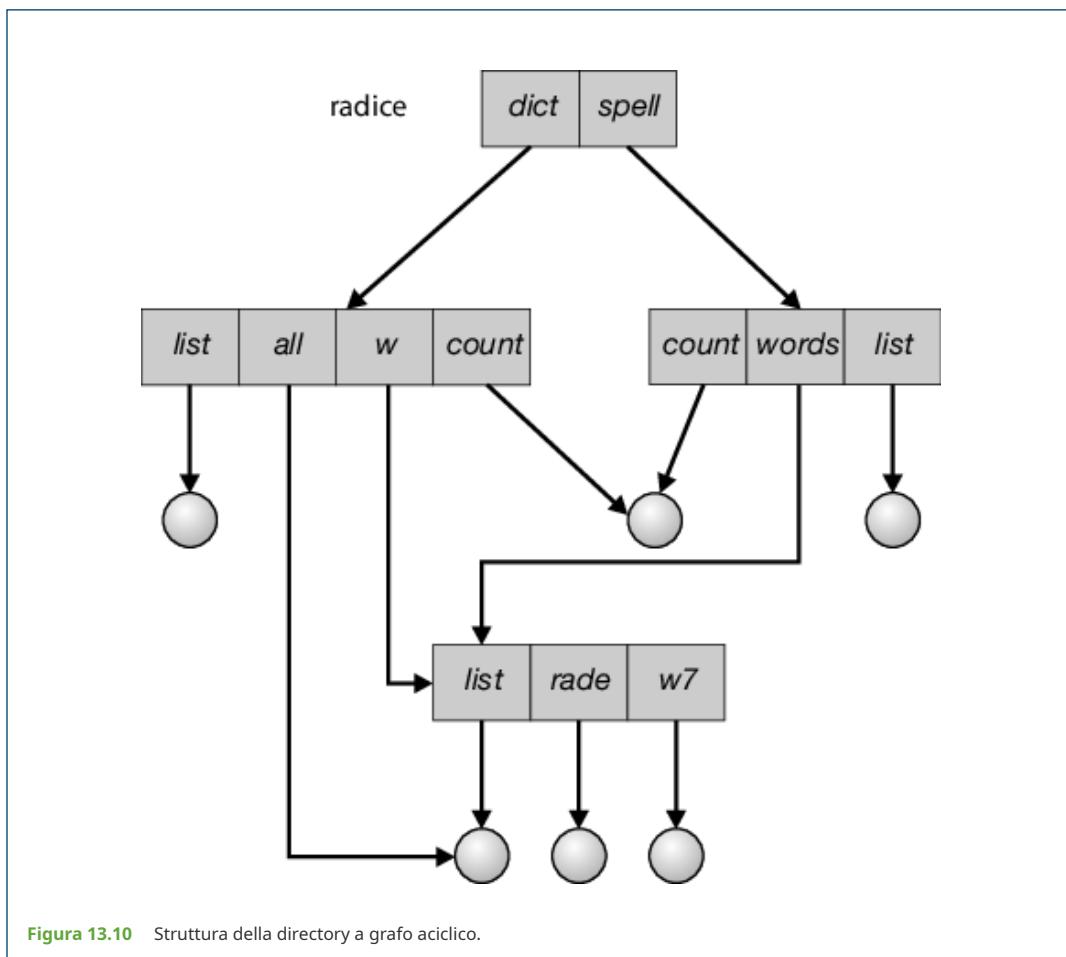
qualche sottodirectory, questa procedura si deve applicare anche alle sottodirectory. Questo metodo può richiedere una discreta quantità di lavoro. In alternativa, come nel comando `rm` di unix, si può avere un'opzione che, alla richiesta di cancellazione di una directory, cancelli anche tutti i file e tutte le sottodirectory in essa contenuti. Entrambi i criteri sono abbastanza facili da realizzare; si tratta soltanto di stabilire la politica da seguire. Il secondo criterio è più comodo, anche se più pericoloso, poiché si può rimuovere un'intera struttura della directory con un solo comando. Se si eseguisse tale comando per sbaglio sarebbe necessario ripristinare un gran numero di file e directory dalle copie di riserva (ipotizzandone l'esistenza).

Con un sistema di directory strutturato ad albero anche l'accesso ai file di altri utenti è di facile realizzazione. Per esempio, l'utente *B* può accedere ai file dell'utente *A* specificando i nomi di percorso assoluti oppure relativi. In alternativa, l'utente *B* può far sì che la propria directory corrente sia quella dell'utente *A* e accedere ai file usando direttamente i loro nomi.

13.3.4 Directory con struttura a grafo aciclico

Si considerino due programmatore che lavorano a un progetto comune. I file associati a quel progetto si possono memorizzare in una sottodirectory, separandoli da altri progetti e file dei due programmatore, ma poiché entrambi i programmatore hanno le stesse responsabilità sul progetto, ciascuno vuole che la sottodirectory si trovi nelle proprie directory. In questa situazione la sottodirectory comune deve essere *condivisa*. Nel sistema esiste quindi una directory o un file condivisi, in due, o più, posizioni contemporaneamente.

La struttura ad albero non ammette la condivisione di file o directory. Un grafo aciclico (cioè senza cicli) permette alle directory di avere sottodirectory e file condivisi (Figura 13.10). Lo stesso file o la stessa sottodirectory possono essere in due directory diverse. Un grafo aciclico rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.



Il fatto che un file sia condiviso, o che lo sia una directory, è diverso dall'avere due copie del file: con due copie ciascun programmatore potrebbe vedere la copia presente nella propria directory e non l'originale; se un programmatore modifica il file, le modifiche non appaiono nell'altra copia. Se invece il file è condiviso esiste *un* solo file effettivo, perciò tutte le modifiche sono

immediatamente visibili. La condivisione è di particolare importanza se applicata alle sottodirectory; un nuovo file creato da un utente appare automaticamente in tutte le sottodirectory condivise.

Quando più persone lavorano insieme, tutti i file da condividere si possono inserire in una directory comune. La directory utente di ogni membro del gruppo contiene questa directory di file condivisi in forma di sottodirectory. Anche nel caso di un singolo utente, l'organizzazione dei file di tale utente può richiedere che alcuni file siano inseriti in più sottodirectory. Per esempio, un programma scritto per un progetto particolare deve trovarsi sia nella directory di tutti i programmi sia nella directory di quel progetto.

I file e le sottodirectory condivisi si possono realizzare in molti modi. Un metodo diffuso, esemplificato da molti tra i sistemi unix, prevede la creazione di un nuovo elemento di directory, chiamato collegamento. Un collegamento (*link*) è un puntatore a un altro file o un'altra directory. Per esempio, un collegamento si può realizzare come un nome di percorso assoluto o relativo. Quando si fa riferimento a un file, si compie una ricerca nella directory. Se l'elemento cercato è contrassegnato come collegamento, riporta il nome di percorso del file reale. Quindi si *risolve* il collegamento usando il nome di percorso per localizzare il file reale. I collegamenti si identificano facilmente tramite il loro formato nell'elemento della directory (o, nei sistemi che gestiscono i tipi, dal tipo speciale), e sono in pratica puntatori indiretti. Durante l'attraversamento degli alberi delle directory il sistema operativo ignora questi collegamenti, preservando così la struttura aciclica.

Un altro comune metodo per la realizzazione dei file condivisi prevede semplicemente la duplicazione di tutte le informazioni relative ai file in entrambe le directory che lo condividono: i due elementi delle directory sono identici. Si consideri la differenza fra i due approcci. Un collegamento è chiaramente diverso dall'elemento originale della directory. Duplicando gli elementi della directory, invece, la copia e l'originale sono resi indistinguibili: sorge allora il problema di mantenere la coerenza se il file viene modificato.

Una struttura di directory a grafo aciclico è più flessibile di una semplice struttura ad albero, ma anche più complessa. Si devono prendere in considerazione parecchi problemi. Un file può avere più nomi di percorso assoluti, quindi nomi diversi possono riferirsi allo stesso file. Questa situazione è simile al problema degli *alias* nei linguaggi di programmazione. Quando si percorre tutto il file system – per trovare un file, per raccogliere dati statistici su tutti i file o per fare le copie di backup dei file – il problema diviene importante poiché le strutture condivise non si devono attraversare più di una volta.

Un altro problema riguarda la cancellazione, poiché è necessario stabilire in quali casi è possibile allocare e riutilizzare lo spazio allocato a un file condiviso. Una possibilità prevede che a ogni operazione di cancellazione seguì l'immediata rimozione del file; quest'azione può però lasciare puntatori sospesi (*dangling*) a un file che ormai non esiste più. Problema ancora più grave, se i puntatori contengono indirizzi effettivi del disco e lo spazio viene poi riutilizzato per altri file, i puntatori potrebbero puntare nel mezzo di questi altri file.

In un sistema dove la condivisione è realizzata da collegamenti simbolici la gestione di questa situazione è relativamente semplice. La cancellazione di un collegamento non influisce sul file originale, poiché si rimuove solo il collegamento. Se si cancella il file, si libera lo spazio corrispondente lasciando in sospeso il collegamento; è possibile ricercare tutti questi collegamenti e rimuoverli, ma se in ogni file non esiste una lista dei collegamenti associati al file stesso questa ricerca può essere abbastanza onerosa. In alternativa, si possono lasciare i collegamenti finché non si tenta di usarli, quindi si scopre che il file con il nome dato dal collegamento non esiste e non si riesce a risolvere il collegamento; l'accesso è trattato proprio come qualsiasi altro nome di file errato. In questo caso, il progettista del sistema deve decidere attentamente cosa si debba fare quando si cancella un file e si crea un altro file con lo stesso nome, prima che sia stato usato un collegamento simbolico al file originario. In unix, quando si cancella un file, i collegamenti simbolici restano, è l'utente che deve rendersi conto che il file originale è scomparso o è stato sostituito. Nella famiglia di sistemi operativi Microsoft Windows si segue lo stesso criterio.

Un altro tipo di approccio alla cancellazione prevede la conservazione del file finché non siano stati cancellati tutti i riferimenti a esso. In questo caso è necessario disporre di un meccanismo che permetta di determinare che l'ultimo riferimento a quel file è stato cancellato; è possibile tenere una lista di tutti i riferimenti a un file (elementi di directory o collegamenti simbolici). Quando si crea un collegamento, oppure una copia dell'elemento della directory, si aggiunge un nuovo elemento alla lista dei riferimenti al file; quando si cancella un collegamento oppure un elemento della directory, si elimina dalla lista l'elemento corrispondente. Quando la sua lista di riferimenti è vuota, il file viene cancellato.

Questo metodo presenta, però, un problema: la dimensione della lista dei riferimenti al file può essere variabile e potenzialmente grande. Tuttavia, non è realmente necessario mantenere l'intera lista, è sufficiente un contatore del numero di riferimenti. Un nuovo collegamento o un nuovo elemento della directory incrementa il numero dei riferimenti; la cancellazione di un collegamento o di un elemento decremente questo numero. Quando il contatore è uguale a 0 si può cancellare il file, poiché non ci sono più riferimenti a tale file. Il sistema operativo unix usa questo metodo per i collegamenti non simbolici, o collegamenti effettivi (*hard link*); il contatore dei riferimenti è tenuto nel blocco di controllo del file o *inode* (si veda il Paragrafo C.7.2 reperibile sulla piattaforma MyLab). Impedendo che si facciano più riferimenti a una directory, si può mantenere una struttura a grafo aciclico.

Per evitare questi problemi alcuni sistemi semplicemente non consentono la condivisione delle directory né i collegamenti.

13.3.5 Directory con struttura a grafo generale

Un serio problema connesso all'uso di una struttura a grafo aciclico consiste nell'assicurare che non vi siano cicli. Iniziando con una directory a due livelli e permettendo agli utenti di creare sottodirectory si crea una directory con struttura ad albero. È facile capire che aggiungendo nuovi file e nuove sottodirectory alla directory con struttura ad albero, la natura di quest'ultima persiste. Tuttavia, quando si aggiungono dei collegamenti a una directory con struttura ad albero, tale struttura si trasforma in una semplice struttura a grafo, come quella illustrata nella Figura 13.11.

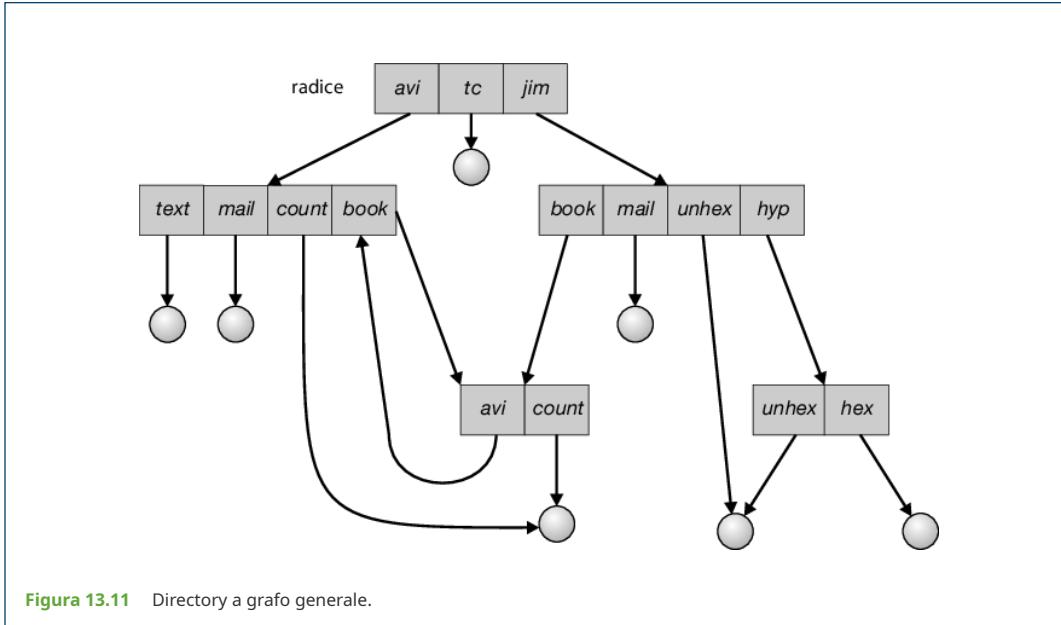


Figura 13.11 Directory a grafo generale.

Il vantaggio principale di un grafo aciclico è dato dalla semplicità degli algoritmi necessari per attraversarlo e per determinare quando non ci siano più riferimenti a un file. È preferibile evitare un duplice attraversamento di sezioni condivise di un grafo aciclico, soprattutto per motivi di prestazioni. Se un file particolare è stato appena cercato in una grande sottodirectory condivisa, ma non è stato trovato, è preferibile evitare una seconda ricerca nella stessa sottodirectory, che costituirebbe solo una perdita di tempo.

Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un ciclo infinito di ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca.

Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con le strutture delle directory a grafo aciclico, la presenza di uno 0 nel contatore dei riferimenti significa che non esistono più riferimenti al file o alla directory, e quindi il file può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore dei riferimenti possa essere non nullo, anche se non è più possibile far riferimento a una directory o a un file. Questa anomalia è dovuta alla possibilità di autoriferimento (ciclo) nella struttura delle directory. In questo caso è generalmente necessario usare un metodo di "ripulitura" (garbage collection) per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. Tale metodo implica l'attraversamento del file system, durante il quale si contrassegna tutto ciò che è accessibile; in un secondo passaggio si raccoglie in un elenco di blocchi liberi tutto ciò che non è contrassegnato. Una procedura di marcatura analoga è utilizzabile per assicurare che un attraversamento o una ricerca coprano tutto quel che si trova nel file system una e una sola volta. La garbage collection di un file system basato su dischi richiede però molto tempo, perciò viene tentata solo di rado.

Inoltre, poiché è necessaria solo a causa della presenza dei cicli, è molto più conveniente lavorare con una struttura a grafo aciclico. La difficoltà consiste nell'evitare i cicli quando si aggiungono nuovi collegamenti alla struttura. Per sapere quando un nuovo collegamento ha completato un ciclo si possono impiegare gli algoritmi che permettono di individuare la presenza di cicli nei grafici. Dal punto di vista del calcolo, però, questi algoritmi sono onerosi, soprattutto quando il grafo si trova in memoria secondaria. Nel caso particolare di directory e collegamenti, un semplice algoritmo prevede di non percorrere i collegamenti durante l'attraversamento delle directory: si evitano così i cicli senza alcun carico ulteriore.

13.4 Protezione

Le informazioni contenute in un sistema elaborativo devono essere protette dai danni fisici (la questione della *affidabilità*) e da accessi impropri (la questione della *protezione*).

Generalmente l'affidabilità è assicurata da più copie dei file. Molti calcolatori hanno programmi di sistema che copiano i file dai dischi ai nastri a intervalli regolari, per esempio una volta al giorno, alla settimana o al mese; quest'operazione di copiatura può essere automatica, o controllata dall'intervento di un operatore. Lo scopo è quello di conservare copie di riserva utili nei casi in cui il file system andasse accidentalmente distrutto. I danni possono essere causati da problemi hardware (di lettura o scrittura), sovraccarichi o cadute della tensione elettrica, rotture delle testine, sporcizia, temperature estreme, atti vandalici, ecc. I file possono inoltre essere cancellati accidentalmente, e anche errori di programmazione possono causare la perdita del contenuto dei file. L'affidabilità è stata trattata con maggiori dettagli nel Capitolo 11.

La protezione si può ottenere in molti modi. Per un computer portatile monoutente, la protezione si può ottenere chiudendo il computer in un cassetto della scrivania oppure in un armadietto. In un sistema multutente più grande sono necessari altri metodi.

13.4.1 Tipi d'accesso

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. I sistemi che non permettono l'accesso ai file di altri utenti non richiedono protezione; quindi si può ottenere una completa protezione proibendo l'accesso. In alternativa si può permettere un accesso totalmente libero senza alcuna protezione. Questi orientamenti sono entrambi eccessivi, quindi non si possono applicare in generale; ciò che serve in realtà è un accesso controllato.

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi d'accesso. Gli accessi si permettono o si negano secondo diversi fattori, innanzitutto i tipi d'accesso richiesti. Si possono controllare diversi tipi di operazione.

- Lettura. Lettura da file.
- Scrittura. Scrittura o riscrittura di file.
- Esecuzione. Caricamento di file in memoria ed esecuzione.
- Aggiunta. Scrittura di nuove informazioni in coda ai file.
- Cancellazione. Cancellazione di file e liberazione del relativo spazio per un possibile riutilizzo.
- Elencazione. Elencazione del nome e degli attributi dei file.

Si possono controllare anche altre operazioni, come ridenominazione, copiatura o modifica dei file. Tuttavia, in molti sistemi queste funzioni di livello superiore si possono realizzare tramite un programma di sistema che compie alcune chiamate di sistema di livello inferiore, quindi la protezione viene garantita a livello inferiore. Per esempio, la copiatura di un file si può realizzare semplicemente con una sequenza di richieste di lettura; in questo caso un utente con accesso per la lettura di un file può richiederne la copiatura, la stampa o altro.

Sono stati proposti molti meccanismi di protezione. Come sempre, ogni meccanismo presenta vantaggi e svantaggi, e deve essere appropriato alla sua particolare applicazione. Un piccolo calcolatore usato soltanto da pochi membri di un gruppo di ricerca non richiede la stessa protezione del sistema elaborativo di una grande società, usato per operazioni di ricerca, finanza e per la gestione del personale. Discutiamo alcuni approcci alla protezione nel seguito e trattiamo il problema più esaustivamente nel Capitolo 17.

13.4.2 Controllo degli accessi

L'approccio più comune al problema della protezione è rendere l'accesso dipendente dall'identità dell'utente. Utenti differenti possono richiedere diversi tipi d'accesso a un file o a una directory. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una lista di controllo degli accessi (*access-control list*, acl) a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti. Quando un utente richiede un accesso a un particolare file il sistema operativo esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista per quel tipo di accesso, viene autorizzato, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

Questo sistema ha il vantaggio di permettere complessi metodi d'accesso. Il problema maggiore delle liste di controllo degli accessi è la loro lunghezza: per permettere a tutti di leggere un file, la lista deve contenere tutti gli utenti con accesso per la lettura. Questa tecnica comporta due inconvenienti:

- la costruzione di una lista di questo tipo può essere un compito noioso e non gratificante, soprattutto se la lista degli utenti del sistema non è nota a priori;
- l'elemento della directory, precedentemente di dimensione fissa, deve essere di dimensione variabile, quindi anche la gestione dello spazio è più complicata.

Questi problemi si possono risolvere introducendo una versione condensata della lista di controllo degli accessi.

Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte.

- Proprietario. È l'utente che ha creato il file.
- Gruppo. Si tratta di un insieme di utenti che condividono il file e hanno bisogno di tipi di accesso simili.
- Universo. Tutti gli altri utenti del sistema.

Il più comune orientamento recente prevede la combinazione delle liste di controllo degli accessi con lo schema di controllo degli accessi per proprietario, gruppo e universo (più facile da realizzare). Il sistema operativo Solaris, per esempio, impiega le tre categorie d'accesso per default ma, se si vuole una maggiore selettività del controllo degli accessi, permette l'attribuzione di liste di controllo degli accessi a specifici file e directory.

Si consideri, per esempio, una persona, Donatella, che sta scrivendo un nuovo libro. Donatella ha assunto tre studenti, Giulia, Paolo e Carlo, per aiutarla a lavorare al progetto. Il testo del libro è mantenuto in un file chiamato `libro.tex`. La protezione associata a tale

file è la seguente:

- Donatella può compiere tutte le operazioni sul file;
- Giulia, Paolo e Carlo possono solo leggere e scrivere il file, ma non possono cancellarlo;
- tutti gli altri utenti possono leggere, ma non scrivere, il file (Donatella ha interesse che il libro sia letto dal maggior numero possibile di persone, in modo da ottenere dei pareri).

Per ottenere tale protezione si deve creare un nuovo gruppo composto da Giulia, Paolo e Carlo, e si deve associare il nome del gruppo, per esempio `testo`, al file `libro.tex` con i diritti d'accesso conformi alla politica ora descritta.

Si consideri un ospite cui Donatella vorrebbe concedere un accesso temporaneo al Capitolo 1. L'ospite non si può aggregare al gruppo `testo` poiché ciò gli darebbe accesso a tutti i capitoli, e considerato che i file possono essere in un solo gruppo, non si può associare un altro gruppo al Capitolo 1. L'aggiunta della funzione delle liste di controllo degli accessi permette di inserire l'ospite nella lista di controllo degli accessi del Capitolo 1.

Affinché questo schema funzioni correttamente, è necessario uno stretto controllo dei permessi e delle liste di controllo degli accessi, fattibile in diversi modi. Nel sistema unix per esempio solo un utente con compiti di gestione (o un *superuser*) può creare e modificare i gruppi, quindi questo controllo si ottiene con la partecipazione umana. Le liste di controllo degli accessi sono trattate anche nel Paragrafo 17.6.2.

PERMESSI IN UN SISTEMA UNIX

Nel sistema unix la protezione delle directory è gestita in modo simile alla protezione dei file. A ciascuna sottodirectory sono associati tre campi (proprietario, gruppo e universo), ciascuno composto dei tre bit rwx. Quindi, un utente può elencare il contenuto di una directory solamente se il bit r è inserito nel campo appropriato. Analogamente, un utente può cambiare la propria directory corrente in un'altra directory (per esempio `foo`) solo se il bit x associato alla directory `foo`, è settato nel campo appropriato.

Un esempio di elenco del contenuto di una directory nell'ambiente unix è illustrato qui di seguito:

```
-rwx-rw-r--    1 pbg      staff     31200    set 3 08:30      intro.ps

drwx---    5 pbg      staff      512    lug 8 09:33      privato/

drwxrwxr-x    2 pbg      staff      512    lug 8 09:35      doc/

drwxrwx--    2 pbg      studente    512    ago 3 14:13      studente-prog/

-rw-r--r--    1 pbg      staff     9423    feb 24 2018      program.c

-rwxr-xr-x    1 pbg      staff     20471   feb 24 2018      program

drwxr-x-x    4 pbg      facoltà    512    lug 31 10:31      lib/

drwx---    3 pbg      staff     1024    ago 29 06:52      mail/
```

```
drwxrwxrwx    3 pbg    staff   512 lug 8 09:35 test/
```

Il primo campo descrive le protezioni di file e directory, il carattere d presente all'inizio del campo contraddistingue le sottodirectory; inoltre, l'elenco contiene il numero di collegamenti relativi al file, il nome del proprietario e del gruppo, la dimensione del file in byte, la data dell'ultima modifica e infine il nome del file (con l'eventuale estensione).

Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi. Normalmente ogni campo è formato di un insieme di bit, ciascuno dei quali permette o impedisce l'accesso che gli è associato. Nel sistema unix, per esempio, sono definiti tre campi di tre bit ciascuno: `rwx`, dove `r` controlla l'accesso per la lettura, `w` quello per la scrittura e `x` per l'esecuzione. Tre campi separati sono riservati al proprietario del file, al gruppo proprietario e a tutti gli altri utenti. In questo schema, per registrare le informazioni di protezione sono necessari nove bit per file. Così, nell'esempio, i campi di protezione per il file `libro.tex` si impostano come segue: per Donatella, il proprietario, tutti e tre i bit sono settati; per il gruppo `testo` i bit `r` e `w`; per l'universo il solo bit `r`.

Nel combinare i due metodi si presenta una difficoltà nell'interfaccia utente; gli utenti devono poter indicare l'impostazione opzionale dei permessi mediante acl per un file. Nell'esempio di Solaris, si aggiunge un segno "+" ai permessi d'accesso standard:

```
19 -rw-r-r-+ 1 alessandra gruppo 130 May 25 22:13 file1
```

Una specifica serie di comandi, `setfacl` e `getfacl`, si usa per gestire le liste di controllo degli accessi.

Gli utenti di Windows, in genere, gestiscono le liste di controllo degli accessi tramite un'interfaccia grafica. La Figura 13.12 mostra la finestra dei permessi di accesso a un file del file system ntfs di Windows 10. In questo esempio si vieta esplicitamente all'utente Guest l'accesso al file `ListPanel.java`.

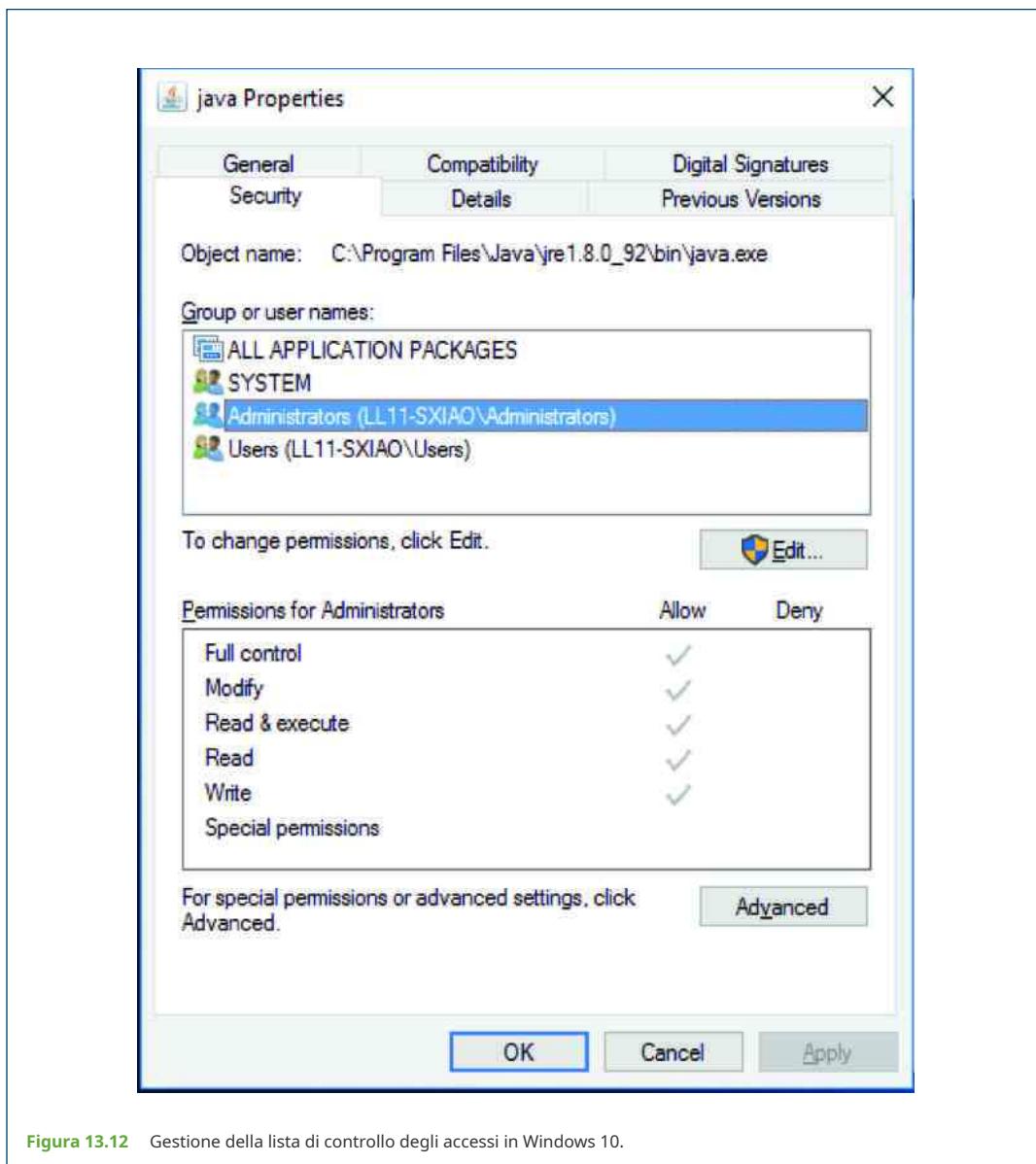


Figura 13.12 Gestione della lista di controllo degli accessi in Windows 10.

Un'altra difficoltà s'incontra nell'assegnazione delle precedenze quando ci sono conflitti tra i permessi e le acl. Per esempio, se Andrea è in un gruppo di file, che ha il permesso di lettura, ma il file ha un lista di controllo degli accessi contenente i permessi di lettura e scrittura per Andrea, si pone il problema della concessione del permesso di scrittura. Nel sistema operativo Solaris hanno precedenza i permessi contenuti nelle liste di controllo degli accessi; sono più selettivi e non sono predefiniti. Si segue il principio generale che la maggiore specificità deve essere prioritaria.

13.4.3 Altri metodi di protezione

Un altro metodo di protezione consiste nell'associazione di una password a ciascun file. Proprio come l'accesso al sistema elaborativo è spesso controllato da una password, anche l'accesso a ogni file può avere lo stesso tipo di protezione. Se le password sono scelte a caso e si cambiano spesso, questo schema può essere efficace nel limitare l'accesso ai file. Questo metodo presenta tuttavia diversi svantaggi: innanzitutto, il numero di parole d'ordine da ricordare può diventare molto alto, rendendo tale metodo impraticabile; secondariamente, se si impiega la stessa parola d'ordine per tutti i file, la sua scoperta li rende tutti accessibili. La protezione è basata sul principio del "o tutto o niente". Per risolvere questo problema alcuni sistemi permettono a un utente di associare una parola d'ordine a una directory anziché a un singolo file.

In una struttura della directory a più livelli è necessario proteggere non solo i singoli file, ma anche gruppi di file contenuti in directory; quindi è necessario disporre di un meccanismo per la protezione delle directory. Le operazioni riguardanti le directory da proteggere sono piuttosto diverse dalle operazioni sui file. Esse sono la creazione e la cancellazione dei file in una directory;

probabilmente va anche controllata la possibilità, per un utente, di determinare l'esistenza di un file in una directory. Talvolta la conoscenza dell'esistenza di un file e del suo nome può essere di per sé significativa, perciò l'elencazione del contenuto di una directory dev'essere un'operazione protetta. Se un nome di percorso fa riferimento a un file in una certa directory, all'utente deve essere consentito l'accesso sia al file sia alla directory. Nei sistemi dove i file possono avere numerosi nomi di percorso (come quelli con struttura a grafo aciclico o a grafo generale) un certo utente può avere diversi diritti d'accesso a un file a seconda del nome di percorso di cui fa uso.

13.5 File mappati in memoria

Esiste un altro metodo, molto utilizzato, per accedere ai file. Si consideri la lettura sequenziale di un file sul disco per mezzo delle consuete chiamate di sistema: `open()`, `read()` e `write()`. Ciascun accesso al file richiede una chiamata di sistema e un accesso al disco. In alternativa, possiamo avvalerci delle tecniche di memoria virtuale analizzate nel Capitolo 10 per trattare l'i/o dei file come l'accesso ordinario alla memoria. Grazie a questa soluzione, nota come mappatura dei file in memoria, una parte dello spazio degli indirizzi virtuali può essere associata logicamente al file. Come vedremo, ciò può portare a un incremento significativo delle prestazioni.

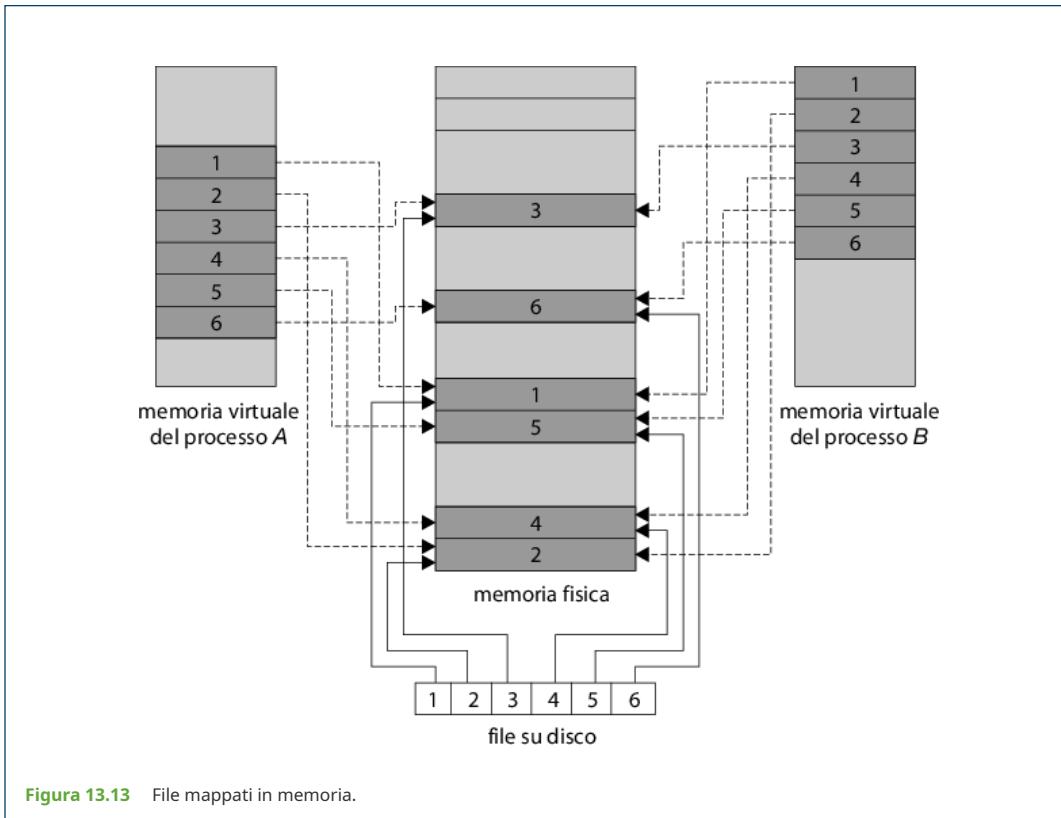
13.5.1 Meccanismo di base

La mappatura di un file in memoria si realizza associando un blocco del disco a una o più pagine residenti in memoria. L'accesso iniziale al file avviene tramite una normale richiesta di paginazione, che causa un errore di page fault. Tuttavia, una porzione del file, che è pari a una pagina, è caricata dal file system in una pagina fisica (alcuni sistemi possono decidere di caricare porzioni più grandi di memoria). Ogni successiva lettura e scrittura del file è gestita come accesso ordinario alla memoria, semplificando e velocizzando così l'accesso al file e il suo utilizzo, in quanto si permette al sistema di manipolare i file attraverso la memoria anziché incorrere nell'overhead delle chiamate di sistema `read()` e `write()`.

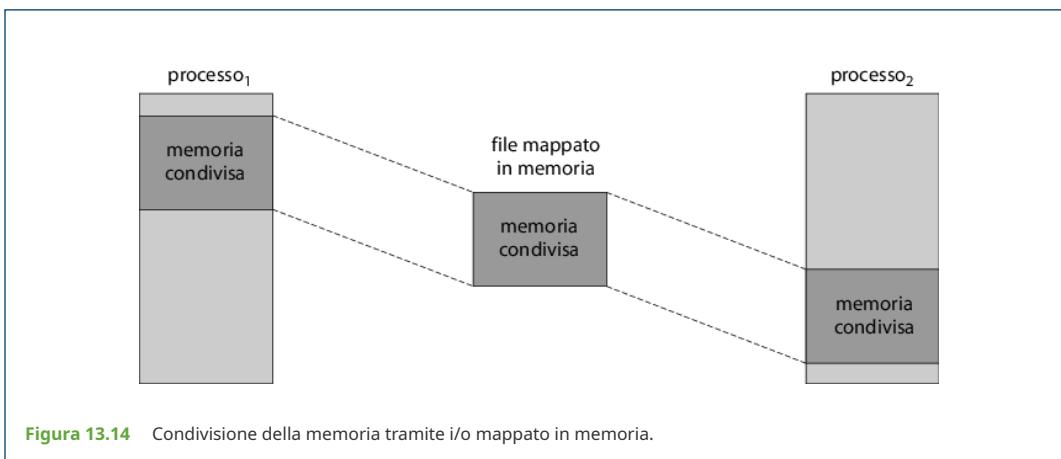
Si osservi come le scritture sul file mappato in memoria non si traducano necessariamente in scritture immediate (sincrone) sul file del disco. Alcuni sistemi scelgono di aggiornare il file fisico quando il sistema operativo esegue un controllo periodico per l'accertamento di eventuali modifiche alle pagine. Quando il file viene chiuso, tutti i dati mappati in memoria sono scritti nuovamente su disco e rimossi dalla memoria virtuale del processo.

Alcuni sistemi operativi prevedono un'apposita chiamata di sistema per la mappatura dei file in memoria; le chiamate ordinarie sono riservate a tutte le altre operazioni di i/o su file. Altri sistemi possono mappare un file in memoria, anche in assenza di un'esplicita indicazione in tal senso. Prendiamo per esempio Solaris. Se si dichiara che un file deve essere mappato in memoria, tramite la chiamata di sistema `mmap()`, Solaris opera la mappatura del file nello spazio degli indirizzi del processo. Se si apre un file, e vi si accede con le chiamate di sistema ordinarie, quali `open()`, `read()` e `write()`, Solaris mappa ancora in memoria il file; tuttavia, il file è mappato nello spazio degli indirizzi del kernel. Quindi, a prescindere da come si apre il file, Solaris considera tutto l'i/o relativo ai file come mappato in memoria, sfruttando per l'accesso ai file l'efficiente sottosistema della memoria.

Per consentire la condivisione dei dati, più processi possono essere autorizzati a mappare contemporaneamente un file in memoria. Le scritture di uno di questi processi modificano i dati nella memoria virtuale e risultano visibili a tutti gli altri processi che mappano la stessa sezione del file. L'analisi sulla memoria virtuale svolta fin qui dovrebbe aver chiarito come la condivisione delle sezioni di memoria interessate dalla mappatura abbia luogo: la memoria virtuale di ciascun processo che partecipa alla condivisione punta alla stessa pagina della memoria fisica – la pagina che ospita una copia del blocco del disco. Tale situazione è illustrata nella Figura 13.13. Le chiamate di sistema per la mappatura in memoria possono inoltre offrire la funzionalità di copiatura su scrittura, che consente ai processi di condividere un file in modalità di sola lettura ma di avere una copia propria dei dati che modificano. Affinché l'accesso ai dati condivisi sia coordinato, i processi interessati potrebbero usare uno dei meccanismi per la mutua esclusione, descritti nel Capitolo 6.



Spesso la memoria condivisa viene implementata utilizzando i file mappati in memoria. La comunicazione fra processi si ottiene in questi casi mappando in memoria uno stesso file negli spazi degli indirizzi virtuali dei processi coinvolti. Il file mappato in memoria funge da area di memoria condivisa tra i processi comunicanti (Figura 13.14). Abbiamo già analizzato questa situazione nel Paragrafo 3.5, in cui si crea un oggetto posix di memoria condivisa e ogni processo comunicante mappa l'oggetto in memoria nel proprio spazio degli indirizzi. Nel paragrafo successivo vedremo come la api Windows fornisca gli strumenti per condividere memoria tramite mappatura dei file in memoria.



13.5.2 Memoria condivisa nella api Windows

La procedura generale per la configurazione di una regione di memoria condivisa utilizzando i file mappati in memoria con la api Windows consiste, dapprima, nel creare un file mapping per il file interessato dall'operazione, per poi stabilire una vista (*view*) del file mappato nello spazio degli indirizzi virtuali di un processo. Un secondo processo, a questo punto, può aprire e creare una sua vista del file mappato nel proprio spazio degli indirizzi virtuali. Il file mappato è l'oggetto di memoria condiviso tramite il quale può svolgersi la comunicazione tra i processi.

Illustriamo ora queste fasi più da vicino. In questo esempio, il processo produttore crea dapprima un oggetto di memoria condiviso, sfruttando le funzionalità per la mappatura di memoria disponibili nella api Windows. Il produttore scrive quindi un messaggio nella memoria condivisa. Il processo consumatore in seguito crea a sua volta una mappatura del file, e legge il messaggio scritto dal produttore.

Per costruire un file mappato in memoria, il processo apre in primo luogo il file da mappare con la funzione `CreateFile()`, che restituisce un riferimento `HANDLE` al file. Il processo allora crea una mappatura di questo `HANDLE`, usando la funzione `CreateFileMapping()`. Una volta stabilita la mappatura del file, il processo genera nel proprio spazio degli indirizzi virtuali una vista del file mappato, con la funzione `MapViewOfFile()`. La vista costituisce la porzione di file che risiederà nello spazio degli indirizzi virtuali del processo (il file può essere mappato solo in parte o per intero). Il programma mostrato nella Figura 13.15 illustra questa procedura (molti controlli sugli errori sono stati omessi per esigenze di brevità).

```
#include <windows.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", /* nome del file */
                      GENERIC_READ | GENERIC_WRITE, /* accesso R/W */
                      0, /* nessuna condivisione del file*/
                      NULL, /* sicurezza di default */
                      OPEN_ALWAYS, /* apre il file (nuovo o esistente) */
                      FILE_ATTRIBUTE_NORMAL, /* attributi del file ordinari */
                      NULL); /* niente template del file*/

    hMapFile = CreateFileMapping(hFile, /* riferimento al file */
                                NULL, /* sicurezza di default */
                                PAGE_READWRITE, /* accesso R/W alle pagine mappate */
                                0, /* mappa l'intero file */
                                0,
                                TEXT("OggettoCondiviso")); /* oggetto condiviso con nome */

    lpMapAddress = MapViewOfFile(hMapFile, /* riferimento al file */
                                FILE_MAP_ALL_ACCESS, /* accesso R/W */
                                0, /* vista dell'intero file */
                                0,
                                0);

    /* scrive nella memoria condivisa */
```

```

sprintf(lpMapAddress," Messaggio nella memoria condivisa ");

UnmapViewOfFile(lpMapAddress);

CloseHandle(hFile);

CloseHandle(hMapFile);

}

```

Figura 13.15 Produttore che scrive nella memoria condivisa tramite la api Windows.

L'invocazione a `CreateFileMapping()` genera un oggetto di memoria condiviso con nome, chiamato `SharedObject`. Il processo consumatore utilizzerà questo segmento di memoria condivisa per comunicare, creando una mappatura del medesimo oggetto con nome. Il produttore, quindi, crea nel proprio spazio degli indirizzi virtuali una vista del file mappato in memoria. Passando agli ultimi tre parametri il valore 0, si richiede che la porzione mappata sia l'intero file; si sarebbero anche potuti passare valori che specifichino l'inizio e la dimensione della porzione da mappare. (Si osservi che non è detto che l'intero file sia caricato in memoria al momento in cui se ne richiede la mappatura: è possibile che il caricamento avvenga tramite paginazione su richiesta, trasferendo perciò di volta in volta le pagine a cui si accede.) La funzione `MapViewOfFile()` restituisce un puntatore all'oggetto di memoria condiviso; ogni accesso a questa locazione di memoria è dunque un accesso al file mappato in memoria. In questo caso, nella memoria condivisa il processo produttore scrive il messaggio "Messaggio nella memoria condivisa".

La Figura 13.16 contiene un programma che dimostra come il processo consumatore stabilisca una vista dell'oggetto di memoria condiviso con nome. Il codice è un po' più semplice di quello della Figura 13.15, poiché il processo deve solo mappare l'oggetto di memoria condiviso con nome già esistente. Come il processo produttore, anche il processo consumatore deve creare una vista del file mappato. Il consumatore, quindi, legge dalla memoria condivisa il messaggio scritto dal processo produttore.

```

#include <windows.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hMapFile;

    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, /* R/W */

        FALSE, /* nessuna ereditarietà */

        TEXT("OggettoCondiviso")); /* nome del file mappato */

    lpMapAddress = MapViewOfFile(hMapFile, /* riferimento al file */

        FILE_MAP_ALL_ACCESS, /* accesso in lettura/scrittura */

        0, /* vista dell'intero file */

        0, /* lettura dalla memoria condivisa */

        printf("Messaggio letto: %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
}

```

```
    CloseHandle(hMapFile);  
}
```

Figura 13.16 Consumatore che legge dalla memoria condivisa tramite la api Windows.

Infine, entrambi i processi eliminano la vista del file mappato chiamando `UnmapViewOfFile()`. Alla fine del capitolo il lettore troverà un esercizio di programmazione per la api Windows incentrato sulla condivisione della memoria tramite mappatura dei file.

13.6 Sommario

- Un file è un tipo di dati astratto definito e realizzato dal sistema operativo. È una sequenza di elementi logici (o *record*), ciascuno dei quali può essere un byte, una riga di lunghezza fissa o variabile, oppure un elemento di dati più complesso. Il sistema operativo può gestire in modo specifico diversi tipi di elementi logici o può lasciare tale gestione al programma applicativo.
- Il compito più importante del sistema operativo consiste nell'associare il concetto logico di file ai dispositivi fisici di memorizzazione, per esempio dischi o nastri magnetici. Poiché le dimensioni dei blocchi fisici dei dispositivi possono non coincidere con quelle dei record logici, può essere necessario riunire un certo numero di record logici per mapparli in un blocco fisico. Anche questo compito può essere gestito dal sistema operativo, oppure lasciato al programma applicativo.
- È utile creare directory per permettere di organizzare i file all'interno di un file system. Una directory a un livello in un sistema multietente causa problemi di naming, poiché ogni file deve avere un nome unico. Una directory a due livelli limita questo problema creando una directory distinta per i file di ciascun utente. Le directory contengono la lista dei file che le costituiscono, insieme con informazioni a essi associate: locazione nei dischi, lunghezza, tipo, proprietario, ora di creazione, ora dell'ultimo uso, e così via.
- La naturale generalizzazione del concetto di directory a due livelli è la directory con struttura ad albero. Tale tipo di struttura permette a un utente di creare sottodirectory in cui organizzare i file. Le strutture delle directory a grafo aciclico permettono la condivisione di sottodirectory e file, ma complicano le funzioni di ricerca e cancellazione. Una struttura a grafo generale permette la massima flessibilità nella condivisione dei file e delle directory, ma talvolta richiede operazioni di "ripulitura" (*garbage collection*) per recuperare lo spazio inutilizzato nei dischi.
- I file system remoti presentano diverse sfide in termini di affidabilità, prestazioni e sicurezza. I sistemi informativi distribuiti mantengono informazioni su utenti, host e accessi in modo che client e server possano condividere le informazioni di stato utili per l'accesso e per la gestione dell'utilizzo del sistema distribuito.
- Poiché i file rappresentano il principale meccanismo di memorizzazione delle informazioni, è necessario che siano dotati di un sistema di protezione. Ogni tipo d'accesso ai file si può controllare separatamente: lettura, scrittura, esecuzione, aggiunta, cancellazione, elencazione del contenuto di directory, e così via. La protezione dei file si può ottenere con password, liste d'accesso, o altre tecniche.

Esercizi di ripasso

13.1 Alcuni sistemi cancellano automaticamente tutti i file utente quando un utente si disconnette oppure quando un job termina, a meno che l'utente non richieda esplicitamente che questi vengano conservati; altri sistemi conservano tutti i file a meno che l'utente non li cancelli esplicitamente. Discutete i meriti di ciascun approccio.

13.2 Perché alcuni sistemi tengono traccia del tipo di un file, mentre altri lasciano questo compito all'utente e altri semplicemente non implementano tipi di file multipli? Quale sistema è "migliore"?

13.3 In modo simile, alcuni sistemi mettono a disposizione molte tipologie diverse di strutture per i dati di un file, mentre altri trattano solo un semplice flusso di byte. Quali sono i vantaggi e gli svantaggi di tali approcci?

13.4 È possibile simulare una struttura di directory multilivello con una struttura di directory a un livello nella quale possono essere usati nomi arbitrariamente lunghi? Se la risposta è affermativa, spiegate come ciò sia fattibile e confrontate questo schema con lo schema a directory multilivello. In caso di risposta negativa, spiegate che cosa impedisce il successo della simulazione. Come cambierebbe la vostra risposta se la lunghezza del nome dei file fosse limitata a sette caratteri?

13.5 Spiegate lo scopo delle operazioni `open()` e `close()`.

13.6 In alcuni sistemi, una sottodirectory può essere letta e scritta da un utente autorizzato esattamente come un file ordinario.

Descrivete i possibili problemi di protezione che ne derivano

Suggerite un metodo per risolvere ciascuno di questi problemi

13.7 Considerate un sistema che supporti 5.000 utenti. Supponete di voler permettere a 4.990 utenti di accedere a un dato file.

a. Come specifichereste questo schema di protezione in unix?

b. Potete suggerire un altro schema di protezione utilizzabile a questo scopo più efficacemente dello schema fornito da unix?

13.8 Alcuni ricercatori hanno suggerito che, invece di associare una lista di accesso a ogni file (dove la lista specifica quali utenti possono accedere al file e come), dovremmo avere *una lista di controllo degli utenti* associata a ogni utente (dove la lista specifica a quali file un utente può accedere e come). Discutete i meriti relativi a questi due schemi.

Esercizi

13.9 Considerate un file system in cui si può cancellare un file e reclamare il suo spazio di memoria secondaria mentre esistono ancora collegamenti (*link*) a esso. Dite quale problema si può presentare se si crea un nuovo file nella stessa area di memoria o con lo stesso nome di percorso assoluto. Spiegate come tali problemi siano evitabili.

13.10 La tabella dei file aperti registra le informazioni riguardanti i file aperti in quel momento. Il sistema operativo dovrebbe mantenere una tabella separata per ciascun utente oppure mantenere una tabella unica per tutti gli utenti con le indicazioni sui file a cui accedono in un certo momento? Se due diversi programmi o utenti eseguono accessi al medesimo file, questi dovrebbero comparire come accessi separati nella tabella dei file aperti? Giustificate le vostre risposte.

13.11 Quali sono vantaggi e svantaggi di un sistema che fornisce lock obbligatori anziché lock consultivi, il cui utilizzo è rimesso al giudizio degli utenti?

13.12 Fornite esempi di applicazioni che tipicamente accedono ai file sequenzialmente e di applicazioni che accedono ai file in maniera casuale.

13.13 Alcuni sistemi aprono un file automaticamente quando ci si riferisce a esso per la prima volta, e lo chiudono al termine del job. Illustrate vantaggi e svantaggi di questo schema, confrontandolo con quello tradizionale, in cui l'utente deve aprire e chiudere il file esplicitamente.

13.14 Qualora il sistema operativo sapesse che una certa applicazione accederà ai dati di un file in modo sequenziale, come potrebbe sfruttare questa informazione per migliorare le prestazioni?

13.15 Illustrate un'applicazione che potrebbe trarre vantaggio da un sistema operativo che offre l'accesso casuale ai file indicizzati.

13.16 Alcuni sistemi consentono la condivisione dei file usando una singola copia di ogni file; altri sistemi impiegano più copie, una per ciascun utente che condivide il file. Discutete i vantaggi di ciascun metodo.

CAPITOLO 14

Realizzazione del file system

Come illustrato nel Capitolo 13, il file system fornisce il meccanismo per la memorizzazione e l'accesso al contenuto dei file, compresi dati e programmi. Il file system risiede permanentemente nella memoria secondaria, progettata per contenere in modo permanente grandi quantità di dati. Questo capitolo riguarda principalmente i problemi connessi alla memorizzazione e all'accesso ai file nel più comune mezzo di memoria secondaria, il disco. Si esaminano varie modalità d'uso dei file, l'allocazione dello spazio dei dischi, il recupero dello spazio liberato, la registrazione delle locazioni dei dati, e l'interfaccia di altri componenti del sistema operativo alla memoria secondaria. Nel corso della trattazione si considerano anche i problemi riguardanti le prestazioni.

Un sistema operativo a uso generale fornisce diversi file system. Inoltre, molti sistemi operativi consentono agli amministratori e agli utenti di aggiungere ulteriori file system. Perché ne servono così tanti? I file system variano in molti aspetti, tra cui funzionalità, prestazioni, affidabilità e obiettivi di progettazione; file system diversi possono servire a scopi diversi. Per esempio, per la memorizzazione e il recupero rapido di file non persistenti viene utilizzato un file system temporaneo, mentre il file system di archiviazione secondaria predefinito (come Linux ext4) sacrifica le prestazioni per garantire affidabilità e funzionalità. Come abbiamo visto nel corso del nostro studio dei sistemi operativi esistono moltissime scelte e varianti, che rendono la copertura completa dei file system una sfida complicata. In questo capitolo ci concentreremo sugli aspetti comuni.

14.1 Struttura del file system

I dischi costituiscono la maggior parte della memoria secondaria in cui si conservano i file system. Hanno due caratteristiche importanti che ne fanno un mezzo adatto a questo scopo:

1. si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione;
2. è possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale sia in modo diretto, e passare da un file all'altro spostando le testine di lettura e scrittura e attendendo la rotazione del disco.

I dispositivi nvm sono sempre più utilizzati per l'archiviazione dei file e su di essi vengono quindi creati dei file system. Questi dispositivi differiscono dagli hard disk, in quanto non possono essere riscritti direttamente e presentano differenti performance La struttura dei dischi e dei dispositivi nvm è analizzata in modo particolareggiato nel Capitolo 11.

Per migliorare l'efficienza dell'i/o, i trasferimenti tra memoria centrale e dischi si eseguono per blocchi. Ciascun blocco è composto da uno o più settori. A seconda dell'unità a disco, la dimensione dei settori è compresa tra 32 byte e 4096 byte; di solito è pari a 512 byte o 4096 byte. I dispositivi nvm hanno normalmente blocchi di 4096 byte e utilizzano metodi di trasferimento simili a quelli delle unità disco.

Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più file system che consentono di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione molto diversi. Il primo riguarda la definizione dell'aspetto del file system agli occhi dell'utente. Questo compito implica la definizione di un file e dei suoi attributi, delle operazionimesse su un file e della struttura delle directory per l'organizzazione dei file. Il secondo riguarda la creazione di algoritmi e strutture dati che permettano di far corrispondere il file system logico ai dispositivi fisici di memoria secondaria.

Lo stesso file system è generalmente composto da molti livelli distinti. La struttura illustrata nella Figura 14.1 è un esempio di struttura stratificata. Ogni livello si serve delle funzioni dei livelli inferiori per crearne di nuove impiegate dai livelli superiori.



Figura 14.1 File system stratificato.

Il livello più basso, il controllo dell'i/o, costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria. Un driver di dispositivo si può concepire come un traduttore che riceva comandi ad alto livello, come "recupera il blocco 123", e che emette istruzioni di basso livello dipendenti dall'hardware, usate dal controllore che fa da interfaccia tra i dispositivi di i/o e il resto del sistema. Un driver di dispositivo di solito scrive specifiche configurazioni di bit in specifiche locazioni della memoria del controllore di i/o per indicare quali azioni il dispositivo di i/o debba compiere, e su quali locazioni. I dettagli dei driver dei dispositivi e le strutture per l'i/o sono trattati nel Capitolo 12.

Il file system di base deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco, per esempio unità 1, cilindro 73, traccia 2, settore 10. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi del file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un i/o richiesto. Le cache servono a conservare i metadati del file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

Il modulo di organizzazione dei file è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici negli indirizzi dei blocchi fisici che il file system di base deve trasferire. I blocchi logici di ciascun file sono numerati da 0 (o 1) a n ; i numeri dei blocchi fisici contenenti i dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione. Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Infine, il file system logico gestisce i metadati; si tratta di tutte le strutture del file system, eccetto gli effettivi dati (il contenuto dei file). Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i blocchi di controllo dei file (*file control block, fcb*), detti inode nei file system unix, contenenti informazioni sui file, come la proprietà, i permessi, e la posizione del contenuto del file. Come si discute nel Capitolo 13 e nel Capitolo 17, il file system logico è responsabile anche della protezione e della sicurezza.

Nei file system stratificati la duplicazione di codice è ridotta al minimo. Il controllo dell'i/o e, talvolta, il codice di base del file system, possono essere utilizzati da più di un file system. Ogni file system ha poi i propri moduli che gestiscono il file system logico e l'organizzazione dei file. Sfortunatamente, la stratificazione può comportare un maggior overhead del sistema operativo, che può generare un conseguente decadimento delle prestazioni. L'utilizzo della stratificazione e le scelte sul numero di strati da impiegare e sulle loro funzionalità rappresentano una grande sfida per la progettazione di nuovi sistemi.

Esistono svariati tipi di file system al giorno d'oggi, e non è raro che i sistemi operativi ne prevedano più d'uno. Molti cd-rom, a esempio, sono scritti nel formato iso 9660, uno standard concordato dai produttori di cd-rom. Oltre ai file system dei supporti rimovibili, ciascun sistema operativo possiede un file system, o più di uno, basato sui dischi. Unix adotta il file system unix (ufs), che si fonda a sua volta sul Berkeley Fast File System (ffs). Windows adotta i formati fat, fat32 e ntfs (o File System di Windows nt), così come i formati per cd-rom e dvd. Sebbene Linux possa funzionare con più di quaranta file system diversi, quello standard è noto come file system esteso, le cui versioni maggiormente diffuse sono ext2 ed ext3. Esistono anche file system distribuiti, in cui un file system su server è montato da uno o più client in una rete.

La ricerca relativa ai file system continua a essere un'area attiva della progettazione e dell'implementazione dei sistemi operativi. Per soddisfare esigenze di memorizzazione e recupero dati specifiche dell'azienda, Google ha progettato un proprio file system che permette un accesso ad alte prestazioni a un gran numero di dischi da parte di numerosi client. Un altro progetto interessante è fuse, che garantisce flessibilità nello sviluppo e nell'utilizzo del file system grazie all'implementazione e all'esecuzione di file system a livello utente invece che a livello del codice del kernel. Gli utenti di fuse possono aggiungere nuovi file system a numerosi sistemi operativi e utilizzarli per gestire i propri file.

14.2 Operazioni del file system

Come detto nel Paragrafo 13.1.2, per permettere ai processi di richiedere l'accesso al contenuto dei file, i sistemi operativi offrono le chiamate di sistema `open()` e `close()`. In questo paragrafo si approfondiscono le strutture dati e le operazioni usate per realizzare le operazioni del file system.

14.2.1 Panoramica

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano a seconda del sistema operativo e del file system, ma esistono dei principi generali. Nei dischi, il file system tiene informazioni su come eseguire l'avviamento di un sistema operativo memorizzato nei dischi stessi, il numero totale di blocchi, il numero e la locazione dei blocchi liberi, la struttura delle directory e i singoli file. Molte di loro sono analizzate in modo particolareggiaato nel seguito di questo capitolo.

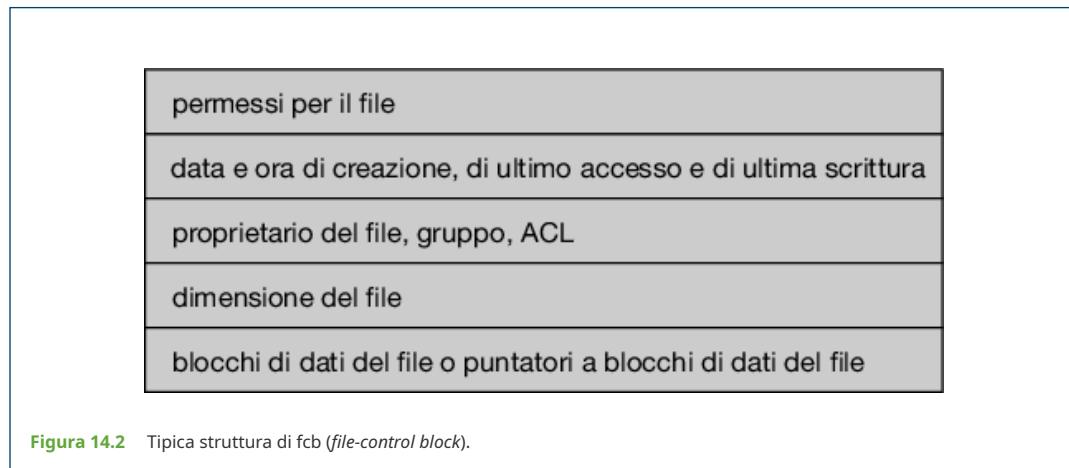
Fra le strutture presenti nei dischi ci sono le seguenti.

- Il blocco di controllo dell'avviamento (*boot control block*), per ogni volume, contenente le informazioni necessarie al sistema per l'avviamento di un sistema operativo da quel volume; se il disco non contiene un sistema operativo, tale blocco può essere vuoto. Di solito è il primo blocco di un volume. Nell'ufs, si chiama blocco d'avviamento (*boot block*); nell'ntfs, settore d'avviamento della partizione (*partition boot sector*).
- Il blocco di controllo del volume (*volume control block*); ciascuno di essi contiene i dettagli riguardanti il relativo volume (o partizione), come il numero e la dimensione dei blocchi nella partizione, il contatore dei blocchi liberi e i relativi puntatori, il contatore degli fcb liberi e i relativi puntatori. Nell'ufs si chiama superblocco; nell'ntfs si chiama tabella principale dei file (*master file table*, mft).
- La struttura della directory (una per file system) usata per organizzare i file. Nel caso dell'ufs comprende i nomi dei file e i numeri di inode associati. Nel caso dell'ntfs è memorizzata nella tabella principale dei file (*master file table*).
- Il blocco di controllo del file (fcb), contenente molti dettagli del relativo file. Ha un identificatore unico per poterlo associare a una voce della directory. Nell'ntfs, queste informazioni sono memorizzate all'interno della tabella principale dei file, che si serve di una struttura di base di dati relazionale, con una riga per ciascun file.

Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l'uso di cache. I dati si caricano al momento del montaggio, si aggiornano mentre si opera sul file system e si eliminano allo smontaggio. Le strutture che vi possono essere incluse sono di diverso tipo:

- la tabella di montaggio, in memoria, che contiene informazioni relative a ciascun volume montato;
- una cache della struttura della directory, tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (per le directory che costituiscono dei punti di montaggio, può essere presente un puntatore alla tabella dei volumi);
- la tabella di sistema dei file aperti, contenente una copia dell'fcb per ciascun file aperto, insieme con altre informazioni;
- la tabella dei file aperti per ciascun processo, contenente un puntatore al corrispondente elemento della tabella generale dei file aperti, insieme con altre informazioni;
- i buffer che conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Per creare un nuovo file, esso crea un nuovo fcb. (In alternativa, nel caso dei file system che creano tutti gli fcb al momento della loro installazione, esso alloca semplicemente un fcb libero.) Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con l'fcb associato, e la scrive nuovamente sul disco. Una tipica struttura di fcb è illustrata nella Figura 14.2.



Alcuni sistemi operativi, compreso unix, trattano le directory esattamente come i file, distinguendole con un campo per il tipo che indica che si tratta di una directory. Altri, tra cui il sistema operativo Windows, dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file. Indipendentemente da tali questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'i/o su directory ai numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'i/o.

14.2.2 Utilizzo

Una volta creato un file, per essere usato per operazioni di i/o deve essere *aperto*. La chiamata di sistema `open()` passa un nome di file al file system logico. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata `open()` dapprima esamina la tabella di sistema dei file aperti. In caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo che punta alla tabella dei file aperti in tutto il sistema. Questo algoritmo può eliminare significativi overhead. Se il file non è già aperto, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'fcb nella tabella di sistema dei file aperti, tenuta in memoria. Questa tabella non solo contiene l'fcb, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.

Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella di sistema e con alcuni altri campi. Questi altri campi possono comprendere un puntatore alla posizione corrente nel file (per successive operazioni `read()` o `write()`) e il tipo d'accesso specificato all'apertura del file. La `open()` riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore. Il nome del file potrebbe non essere contenuto nella tabella dei file aperti, visto che, una volta che il corrispondente fcb è stato individuato nei dischi, il sistema non ne ha bisogno. Tuttavia, potrebbe venir memorizzato in una cache per risparmiare tempo sulle aperture successive dello stesso file. Il nome dato all'elemento della tabella è detto descrittore di file (*file descriptor*) in unix, e handle del file in Windows.

Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella di sistema. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrivono i metadati aggiornati nella struttura della directory nei dischi e si cancella il relativo elemento nella tabella di sistema dei file aperti.

Alcuni sistemi complicano ulteriormente lo schema descritto, usando il file system come interfaccia per altri aspetti del sistema, come la comunicazione in rete. Per esempio, nell'ufs, la tabella generale dei file aperti contiene gli *inode* e altre informazioni su file e directory, ma contiene anche informazioni simili per le connessioni di rete e i dispositivi. In questo modo si può usare un unico meccanismo per molteplici fini.

Le questioni concernenti l'uso delle cache per queste strutture non vanno però trascurate. La maggior parte dei sistemi mantiene in memoria tutta l'informazione su un file aperto, eccetto i suoi effettivi blocchi di dati. Il sistema unix bsd è noto per il suo uso di cache ovunque sia possibile risparmiare su operazioni di i/o nei dischi. La sua frequenza media di successi nella cache, pari all'85 per cento, dimostra l'utilità di queste tecniche. Il sistema unix bsd è descritto esaustivamente nell'Appendice C (reperibile sulla piattaforma MyLab).

La Figura 14.3 riassume le strutture dati che si usano nella realizzazione di un file system.

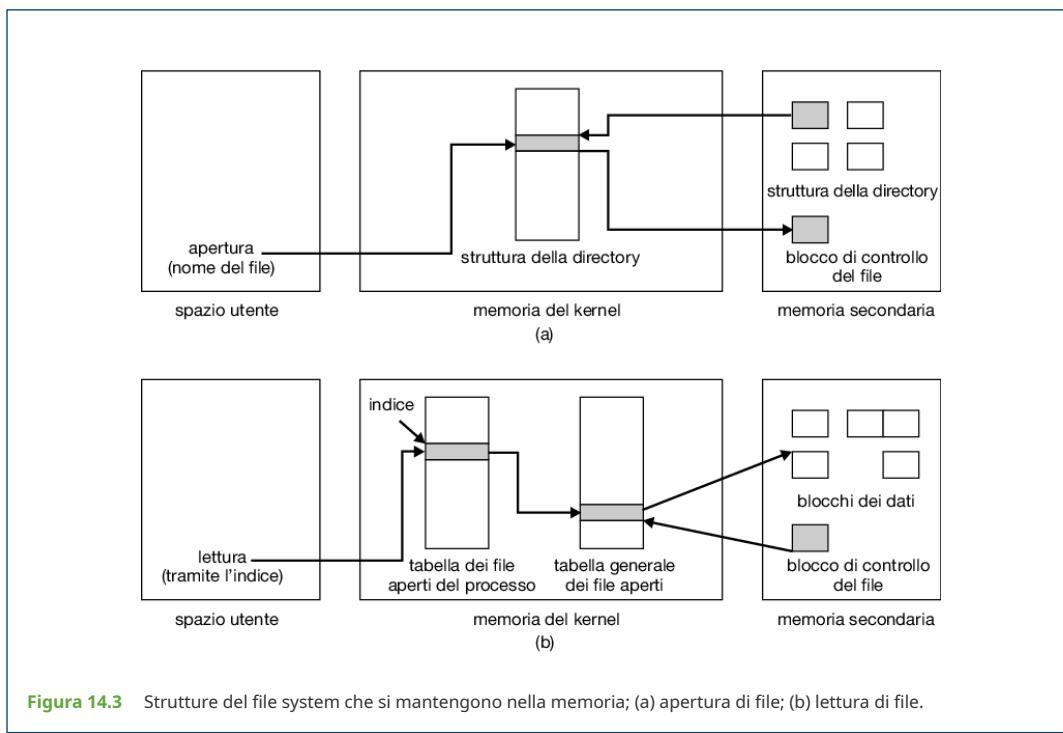


Figura 14.3 Strutture del file system che si mantengono nella memoria; (a) apertura di file; (b) lettura di file.

14.3 Realizzazione delle directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del file system. Nel seguito discuteremo i tradeoff legati alla scelta di questi algoritmi.

14.3.1 Lista lineare

Il più semplice metodo di realizzazione di una directory è basato sull'uso di una lista lineare contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo. Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era assegnato. Esistono vari metodi per riutilizzare un elemento della directory: si può contrassegnare l'elemento come non usato (attribuendogli un nome speciale, come un nome blank, oppure includendo un bit d'uso in ogni elemento), oppure può essere aggiunto a una lista di elementi di directory liberi; una terza possibilità prevede la copiatura dell'ultimo elemento della directory in una locazione liberata e la diminuzione della lunghezza della directory. Per ridurre il tempo di cancellazione di un file si può usare anche una lista concatenata.

Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file. Le informazioni sulla directory vengono usate frequentemente, e gli utenti si accorgono se l'accesso a tali informazioni è lento. In effetti, molti sistemi operativi impiegano una cache software per memorizzare le informazioni di directory usate più recentemente. La presenza nella cache delle informazioni richieste ne evita la continua rilettura dai dischi. Una lista ordinata permette una ricerca binaria e riduce il tempo medio di ricerca; tuttavia il requisito dell'ordinamento può complicare la creazione e la cancellazione di file, poiché, per tenere ordinata la lista, può essere necessario spostare quantità notevoli di informazioni di directory. In questo caso, può essere d'aiuto una struttura dati più raffinata, come un albero bilanciato. Un vantaggio della lista ordinata è che consente di produrre l'elenco ordinato del contenuto della directory senza una fase d'ordinamento separata.

14.3.2 Tabella hash

Un'altra struttura dati che si usa per realizzare le directory è la tabella hash. In questo caso una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato a partire dal nome del file e riporta un puntatore al nome del file nella lista lineare. Questa struttura dati può diminuire notevolmente il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per gestire le collisioni, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

Le maggiori difficoltà legate a una tabella hash sono la sua dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione. Si supponga, per esempio, di realizzare una tabella hash di 64 elementi con gestione lineare delle collisioni; la funzione hash converte i nomi di file in interi da 0 a 63, per esempio, usando il resto di una divisione per 64. Per creare in un secondo tempo un sessantacinquesimo file occorre allungare la tabella hash della directory, per esempio fino a 128 elementi. Occorre quindi una nuova funzione hash per associare i nomi di file all'intervallo compreso tra 0 e 127, e gli elementi esistenti nella directory si devono riorganizzare in modo da riflettere i loro nuovi valori della funzione hash.

Alternativamente, ciascun elemento della tabella hash, anziché un singolo valore, può essere una lista concatenata; ciò consente di risolvere le collisioni aggiungendo il nuovo elemento alla lista concatenata. Le ricerche vengono alquanto rallentate, poiché la ricerca per nome può richiedere l'attraversamento di una lista concatenata degli elementi in collisione della tabella hash; tuttavia tale metodo è verosimilmente più veloce di una ricerca lineare nell'intera directory.

14.4 Metodi di allocazione

La natura ad accesso diretto dei dischi dà flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere contigua, concatenata o indicizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi. Anche se alcuni sistemi dispongono di tutti e tre i metodi, più spesso un sistema usa un unico metodo per tutti i file all'interno di un certo tipo di file system.

14.4.1 Allocazione contigua

Per usare il metodo di allocazione contigua, ogni file deve occupare un insieme di blocchi contigui del disco. Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso. Con questo ordinamento, supponendo che un unico job stia accedendo al disco, l'accesso al blocco $b + 1$ dopo il blocco b non richiede normalmente alcuno spostamento della testina. Se la testina deve essere spostata (dall'ultimo settore di un cilindro al primo settore del cilindro successivo) lo spostamento è di una sola traccia. Quindi, il numero dei posizionamenti (*seek*) richiesti per accedere a file il cui spazio è allocato in modo contiguo è minimo, così com'è trascurabile il tempo di ricerca (*seek time*), quando quest'ultimo è necessario.

L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco e dalla lunghezza (espressa in numero di blocchi). Se il file è lungo n blocchi e comincia dalla locazione b , allora occupa i blocchi $b, b + 1, b + 2, \dots, b + n - 1$. L'elemento di directory per ciascun file indica l'indirizzo del blocco d'inizio e la lunghezza dell'area assegnata per questo file (Figura 14.4).

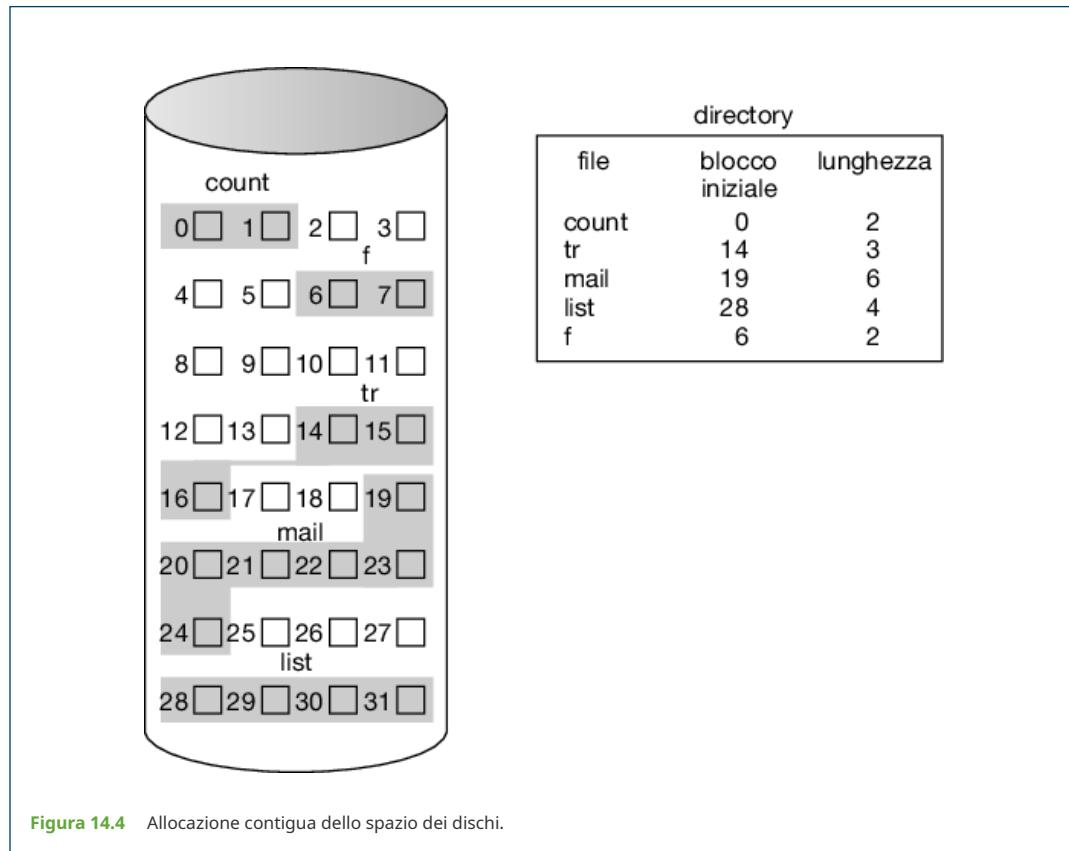


Figura 14.4 Allocazione contigua dello spazio dei dischi.

Accedere a un file il cui spazio è assegnato in modo contiguo è facile. Quando si usa un accesso sequenziale, il file system memorizza l'indirizzo dell'ultimo blocco cui è stato fatto riferimento e, se è necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco i di un file che comincia al blocco b si può accedere immediatamente al blocco $b + i$. Quindi, sia l'accesso sequenziale sia quello diretto si possono gestire con l'allocazione contigua.

L'allocazione contigua presenta però alcuni problemi. Una difficoltà riguarda l'individuazione dello spazio per un nuovo file. La realizzazione del sistema di gestione dello spazio libero, illustrata nel Paragrafo 14.5, determina il modo in cui tale compito viene eseguito. Si può usare ogni sistema di gestione, anche se alcuni sono più lenti di altri.

Il problema dell'allocazione contigua dello spazio dei dischi si può considerare un'applicazione particolare del problema generale dell'allocazione dinamica della memoria, trattato nel Paragrafo 9.2; il problema generale è, infatti, quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono *first-fit* e *best-fit*. Simulazioni hanno dimostrato che questi due criteri sono più efficienti del *worst-fit* sia in termini di tempo sia d'uso della memoria. Nessuno dei due è chiaramente migliore dell'altro rispetto all'uso della memoria, ma *first-fit* è generalmente più rapido.

Questi algoritmi soffrono della frammentazione esterna: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. La frammentazione esterna si ha ognqualvolta lo spazio libero è suddiviso in pezzi, e diviene un problema quando il più grande di tali pezzi contigui non è sufficiente a soddisfare una richiesta; la memoria viene frammentata in tanti buchi, nessuno dei quali è abbastanza grande da contenere i dati. A seconda della capacità dei dischi e della dimensione media dei file, la frammentazione esterna può essere un problema più o meno grave.

Una strategia per prevenire la perdita di una quantità significativa di spazio sul disco a causa della frammentazione esterna consiste nel copiare un intero file system su un altro disco. Così si libera completamente il primo disco creando un ampio spazio libero contiguo; poi si copiano nuovamente i file nel primo disco, assegnando spazio contiguo da questa ampia zona libera. Questo schema compatta efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Il costo di questa compattazione è rappresentato dal tempo necessario, ed è particolarmente alto per i dischi di grande capacità; per essi, compattare lo spazio può richiedere ore e può essere necessario eseguire tale operazione settimanalmente. Alcuni sistemi richiedono l'esecuzione non in linea (*off-line*) di questa funzionalità, ossia con il file system non montato. Durante questo periodo di indisponibilità (*down time*) il normale funzionamento del sistema non è in genere possibile, quindi tale compattazione viene evitata a tutti i costi per i calcolatori operativi. La maggior parte dei sistemi moderni è invece in grado di eseguire la deframmentazione in linea (*on-line*), ossia durante il loro normale funzionamento, a prezzo, però, di una notevole diminuzione delle prestazioni.

Un altro problema che riguarda l'allocazione contigua è la determinazione della quantità di spazio necessaria per un file. Quando si crea un file, occorre trovare e allocare lo spazio di cui necessita. Come può il programma o la persona che lo crea conoscere la dimensione del file da creare? In alcuni casi questa dimensione si può stabilire in modo abbastanza semplice, per esempio quando si copia un file esistente; in generale, tuttavia, non è facile stimare la dimensione di un file che deve contenere dati prodotti da un programma.

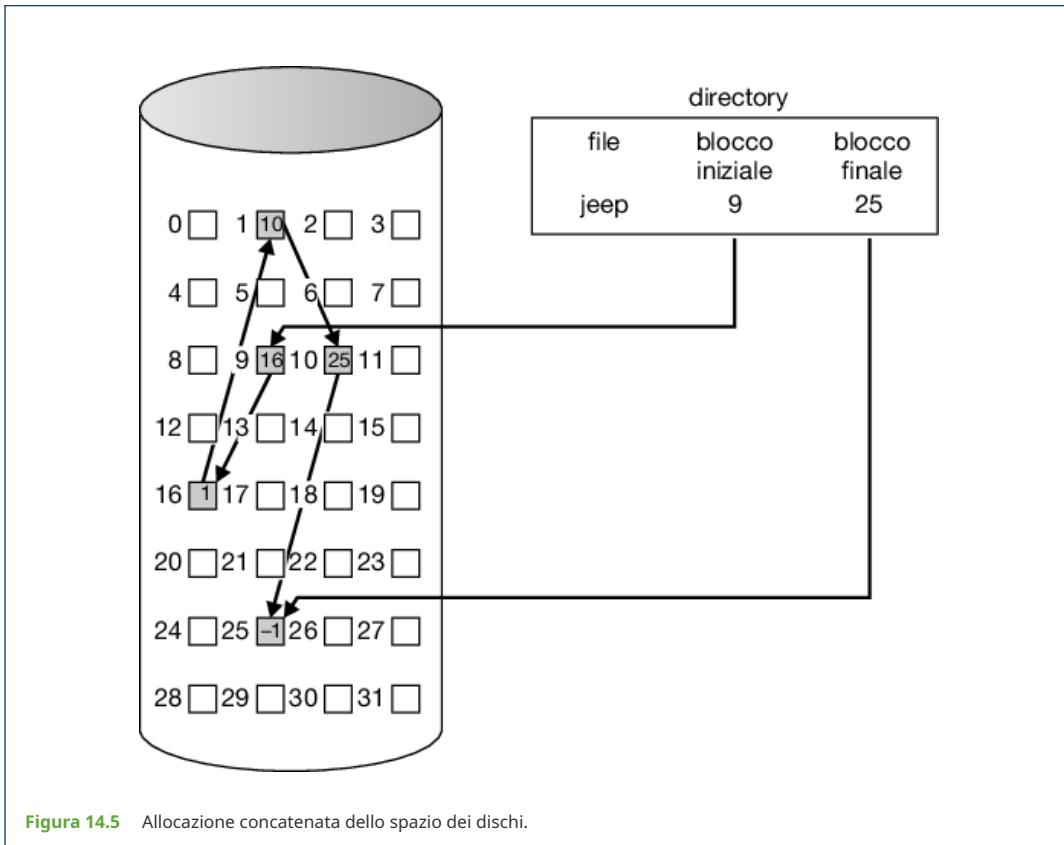
Se un file riceve poco spazio, può essere impossibile estenderlo: soprattutto nel caso in cui si adoperi il criterio di allocazione *best-fit*, lo spazio alle due estremità del file può essere già in uso, quindi non è possibile ampliare il file in modo contiguo. Esistono allora due possibilità. La prima è terminare il programma utente con un idoneo messaggio d'errore. L'utente deve allora allocare più spazio ed eseguire di nuovo il programma. Queste esecuzioni ripetute possono essere onerose; per prevenire tale circostanza, normalmente l'utente sovrastima la quantità di spazio necessaria, sprecandone parecchio. L'altra possibilità consiste nel trovare un buco più grande, copiare il contenuto del file nel nuovo spazio e rilasciare lo spazio precedente. Queste operazioni si possono ripetere finché esiste spazio, anche se ciò può far perdere tempo. In questo caso tuttavia non è necessario informare esplicitamente l'utente su che cosa stia succedendo; anche se sempre più lentamente, il sistema prosegue le attività nonostante il problema.

Anche se si conosce in anticipo la quantità di spazio necessaria per un file, l'allocazione preventiva può in ogni modo essere inefficiente. A un file che cresce lentamente in un periodo di tempo lungo (mesi o anni) si deve allocare spazio sufficiente per la sua dimensione finale, anche se molto di quello spazio può rimanere inutilizzato per parecchio tempo. Il file ha perciò una grande frammentazione interna.

Per ridurre al minimo questi inconvenienti, alcuni sistemi operativi fanno uso di uno schema di allocazione contigua modificato: inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio contiguo, detta estensione. Allora la locazione dei blocchi dei file si registra come una locazione e un numero dei blocchi, insieme con l'indirizzo del primo blocco della prossima estensione. In alcuni sistemi il proprietario del file può impostare la dimensione dell'estensione, ma tale possibilità, se l'indicazione è imprecisa, può causare inefficienze. La frammentazione interna può ancora essere un problema se le estensioni sono troppo grandi; si possono presentare problemi dovuti alla frammentazione esterna quando si assegnano e si rilasciano estensioni di dimensione variabile. Il file system commerciale Symantec Veritas impiega le estensioni per ottimizzare le prestazioni; Veritas è un sostituto ad alte prestazioni dell'ordinario ufs di unix.

14.4.2 Allocazione concatenata

L'allocazione concatenata risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Per esempio, un file di cinque blocchi può cominciare dal blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25 (Figura 14.5). Ogni blocco contiene un puntatore al blocco successivo. Questi puntatori non sono disponibili all'utente, quindi se ogni blocco è formato di 512 byte e un indirizzo del disco (il puntatore) richiede 4 byte, l'utente vede blocchi di 508 byte.



Per creare un nuovo file si crea semplicemente un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore s'inizializza a `null` (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo della dimensione s'imposta a 0. Un'operazione di scrittura nel file determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura in tale blocco, e la concatenazione di tale blocco alla fine del file. Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori da un blocco all'altro. Con l'allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione. Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco.

L'allocazione concatenata presenta comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale. Per trovare l' i -esimo blocco di un file occorre partire dall'inizio del file e seguire i puntatori finché non si raggiunge l' i -esimo blocco. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina. Di conseguenza, per file il cui spazio è assegnato in modo concatenato, la funzione d'accesso diretto è inefficiente.

Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Se un puntatore richiede 4 byte di un blocco di 512 byte, allora lo 0,78 per cento del disco è usato per i puntatori anziché per le informazioni: ogni file richiede un po' più spazio di quanto ne richiederebbe altrimenti.

La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in cluster (gruppi di blocchi), e nell'allocare i cluster anziché i blocchi. Per esempio, il file system può definire cluster di 4 blocchi e operare nel disco soltanto per unità di cluster. Così i puntatori usano una percentuale molto più piccola dello spazio del disco. Questo metodo permette che la corrispondenza tra blocchi logici e blocchi fisici rimanga semplice, ma migliora il throughput del disco poiché si hanno meno posizionamenti della testina, inoltre diminuisce lo spazio necessario per l'allocazione dei blocchi e la gestione della lista dei blocchi liberi. Il costo di questo metodo è dato da un incremento della frammentazione interna, poiché se un cluster è parzialmente pieno si spreca più spazio di quanto se ne sprecerebbe con un solo blocco parzialmente pieno. I cluster si possono usare per ottimizzare l'accesso ai dischi in molti altri algoritmi, quindi s'impiegano nella maggior parte dei file system.

Un altro problema dell'allocazione concatenata riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, s'immagini che cosa accadrebbe se un puntatore andasse perduto o danneggiato. Un errore di programmazione del sistema operativo oppure un errore hardware di un'unità a disco potrebbero causare la lettura di un puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. Una soluzione parziale a tale problema consiste nell'usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco; questi schemi però richiedono un overhead ancora maggiore per ogni file.

Una variante importante del metodo di allocazione concatenata consiste nell'uso della tabella di allocazione dei file (*file allocation table*, fat). Tale metodo di allocazione dello spazio dei dischi, semplice ma efficiente, era usato nei sistemi operativi ms-dos. Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la fat ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file richiede semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file. Un esempio esplicativo di tale metodo è dato dalla struttura della fat della Figura 14.6, per un file formato dai blocchi 217, 618 e 339.

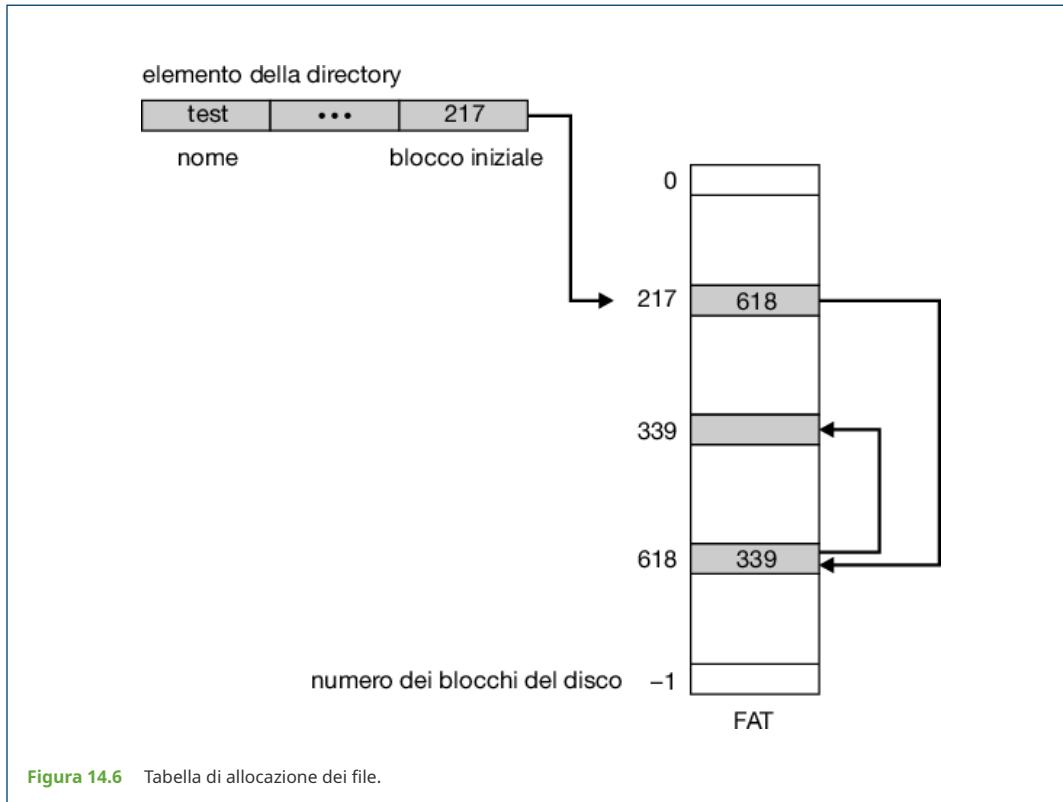


Figura 14.6 Tabella di allocazione dei file.

Lo schema di allocazione basato sulla fat, se non si usa una cache, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio del volume per leggere la fat e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco. Un vantaggio è dato dall'ottimizzazione del tempo d'accesso diretto, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella fat.

14.4.3 Allocazione indicizzata

L'allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni dei file, presenti nell'allocazione contigua. Tuttavia, in mancanza di una fat, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine. L'allocazione indicizzata risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il blocco indice.

Ogni file ha il proprio blocco indice: si tratta di un array d'indirizzi di blocchi del disco. L' i -esimo elemento del blocco indice punta all' i -esimo blocco del file. La directory contiene l'indirizzo del blocco indice, com'è illustrato nella Figura 14.7. Per individuare e leggere l' i -esimo blocco occorre usare il puntatore che si trova nell' i -esimo elemento del blocco indice. Questo schema è simile a quello della paginazione descritto nel Paragrafo 9.3.

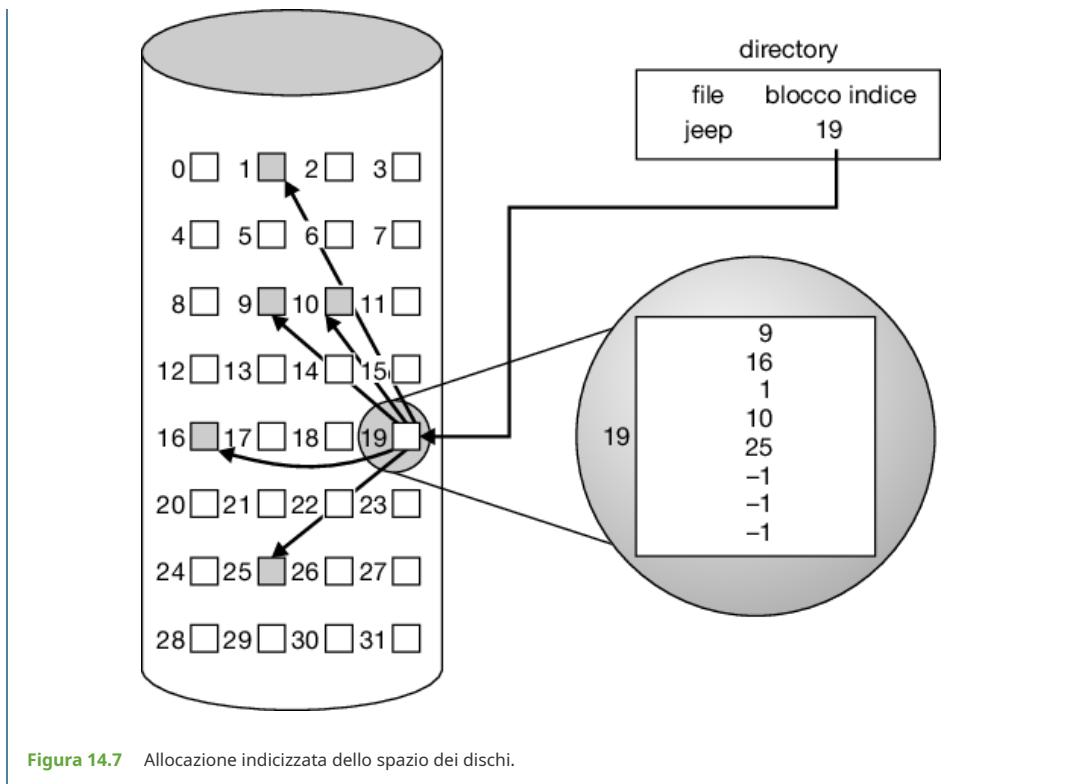


Figura 14.7 Allocazione indicizzata dello spazio dei dischi.

Una volta creato il file, tutti i puntatori del blocco indice sono impostati a `null`. Quando si scrive l' i -esimo blocco per la prima volta, il gestore dei blocchi liberi fornisce un blocco; l'indirizzo di questo blocco viene inserito nell' i -esimo elemento del blocco indice. Poiché ogni blocco libero del disco può soddisfare una richiesta di maggiore spazio, l'allocazione indicizzata consente l'accesso diretto senza soffrire di frammentazione esterna.

L'allocazione indicizzata soffre tuttavia di un overhead maggiore: lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spazio aggiuntivo necessario per l'allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l'allocazione concatenata si perde il solo spazio di un puntatore per blocco, complessivamente uno o due puntatori; con l'allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori sono diversi da `null`.

Questo punto solleva la questione della dimensione del blocco indice. Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo è possibile; ma se il blocco indice è troppo piccolo non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione. Fra i possibili meccanismi vi sono i seguenti.

- Schema concatenato. Un blocco indice è formato normalmente di un solo blocco di disco; perciò, ciascun blocco indice può essere letto e scritto esattamente con un'operazione. Per permettere la presenza di lunghi file è possibile collegare tra loro parecchi blocchi indice. Per esempio, un blocco indice può contenere una piccola intestazione in cui sono riportati il nome del file e l'insieme dei primi 100 indirizzi di blocchi del disco. L'indirizzo successivo, vale a dire l'ultima parola del blocco indice, è `null` (per un file piccolo) oppure è un puntatore a un altro blocco indice (per un file lungo).
- Indice a più livelli. Una variante della rappresentazione concatenata consiste nell'impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file. Per accedere a un blocco, il sistema operativo usa l'indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco di dati richiesto. Questo metodo potrebbe continuare fino a un terzo o quarto livello, a seconda della massima dimensione desiderata del file. Con blocchi di 4096 byte si possono memorizzare 1024 puntatori di 4 byte in un blocco indice. Due livelli di indici consentono 1.048.576 blocchi di dati, che permettono di avere file sino a 4 gb.
- Schema combinato. Un'altra possibilità, utilizzata nei sistemi basati su unix, consistente nel tenere i primi 15 puntatori del blocco indice nell'*inode* del file. I primi 12 di questi 15 puntatori puntano a blocchi diretti, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Quindi, i dati per piccoli file (non più di 12 blocchi) non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4 kb, è possibile accedere direttamente fino a 48 kb di dati. Gli altri tre puntatori puntano a blocchi indiretti. Il primo è un puntatore a un blocco indiretto singolo; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Il secondo è un puntatore a un blocco indiretto doppio contenente l'indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. Il terzo è un puntatore a un blocco indiretto triplo. Un *inode* unix è mostrato nella Figura 14.8.

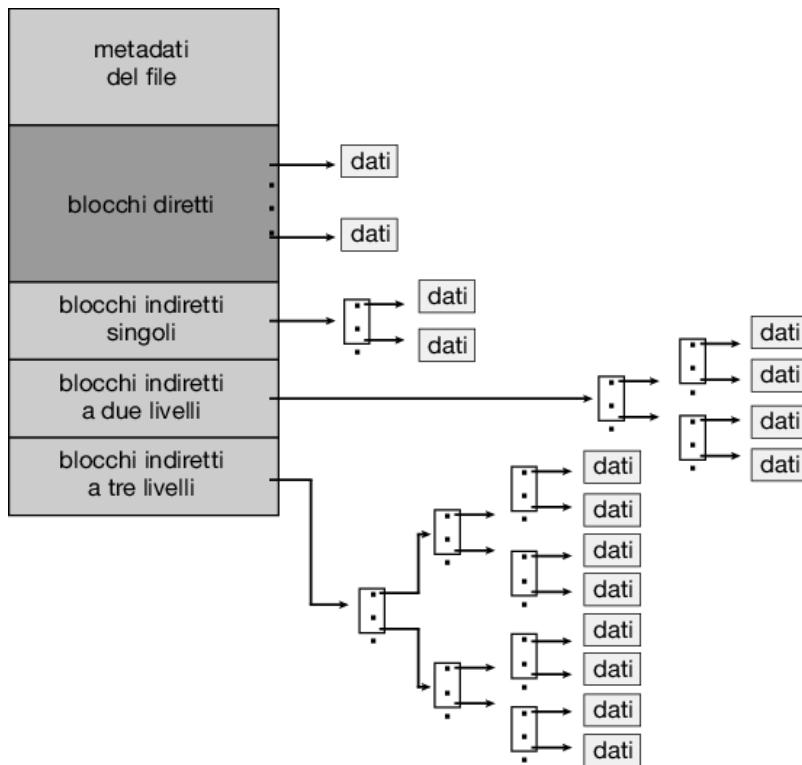


Figura 14.8 Inode di unix.

Con questo metodo il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti sistemi operativi. Un puntatore a file di 32 bit consente di arrivare a soli 2^{32} byte, 4 gb. Molte versioni di unix e Linux ora gestiscono puntatori a file sino a 64 bit. Puntatori di questa dimensione permettono di avere file e file system di dimensioni dell'ordine degli exabyte (2^{60} byte). Il file system zfs supporta puntatori a 128 bit.

Gli schemi d'allocazione indicizzata soffrono di alcuni dei problemi di prestazioni dell'allocazione concatenata. In particolare, i blocchi indice si possono caricare in memoria, ma i blocchi dei dati possono essere sparsi per un intero volume.

14.4.4 Prestazioni

I metodi d'allocazione presentati hanno diversi livelli di efficienza di memorizzazione e differenti tempi d'accesso ai blocchi di dati; entrambi i fattori sono importanti nella scelta del metodo o dei metodi d'allocazione più adatti da impiegare in un sistema operativo.

Prima di scegliere un metodo di allocazione, è necessario determinare il modo in cui si usano i sistemi: un sistema con una prevalenza di accessi sequenziali farà uso di un metodo differente da quello di un sistema con una prevalenza di accessi diretti.

Per qualsiasi tipo d'accesso, l'allocazione contigua richiede un solo accesso per ottenere un blocco. Poiché è facile tenere l'indirizzo iniziale del file in memoria, si può calcolare immediatamente l'indirizzo del disco dell'*i*-esimo blocco, oppure del blocco successivo, e leggerlo direttamente.

Con l'allocazione concatenata si può tenere in memoria anche l'indirizzo del blocco successivo e leggerlo direttamente. Questo metodo è valido per l'accesso sequenziale mentre, per quel che riguarda l'accesso diretto, un accesso all'*i*-esimo blocco può richiedere *i* letture del disco. Questo spiega perché l'allocazione concatenata non si dovrebbe usare per un'applicazione che richiede accessi diretti.

Da tutto ciò segue che alcuni sistemi gestiscono i file ad accesso diretto usando l'allocazione contigua, e i file ad accesso sequenziale tramite l'allocazione concatenata. Per questi sistemi, il tipo d'accesso si deve dichiarare al momento della creazione del file. Un file creato per l'accesso sequenziale è un file concatenato e non si può usare per l'accesso diretto. Un file creato per l'accesso diretto è contiguo e consente entrambi i tipi d'accesso, ma bisogna dichiararne la lunghezza massima al momento della sua creazione. In questo caso, il sistema operativo deve avere strutture dati idonee e algoritmi capaci di gestire *entrambi* i metodi di allocazione. I file si possono convertire da un tipo all'altro creando un nuovo file del tipo desiderato, nel quale si copia il contenuto del vecchio file; si può quindi cancellare quest'ultimo e rinominare il nuovo file.

L'allocazione indicizzata è più complessa. Se il blocco indice è già in memoria, l'accesso può essere diretto. Tuttavia, per tenere il blocco indice in memoria occorre una quantità di spazio considerevole. Se questo spazio di memoria non è disponibile, occorre leggere prima il blocco indice e quindi il blocco di dati desiderato. Per un indice a due livelli possono essere necessarie due letture di blocco indice. Se un file è estremamente grande, per compiere l'accesso a un blocco che si trovi vicino alla fine del file, prima di leggere il blocco dei dati occorre leggere tutti i blocchi indice per seguire la catena dei puntatori. Quindi le prestazioni dell'allocazione indicizzata dipendono dalla struttura dell'indice, dalla dimensione del file e dalla posizione del blocco desiderato.

Alcuni sistemi combinano l'allocazione contigua con l'allocazione indicizzata, usando quella contigua per i file piccoli (fino a tre o quattro blocchi) e passando automaticamente a quella indicizzata per i file grandi. Poiché la maggior parte dei file sono piccoli, e in questo caso l'allocazione contigua è efficiente, le prestazioni medie possono risultare decisamente buone.

Sono possibili e si usano effettivamente anche molte altre ottimizzazioni. Data la disparità tra velocità della cpu e velocità dei dischi, non è irragionevole aggiungere al sistema operativo migliaia di istruzioni solo per risparmiare alcuni movimenti della testina. Con il passare del tempo tale disparità continua ad aumentare tanto che, per ottimizzare i movimenti della testina, si potranno ragionevolmente usare centinaia di migliaia di istruzioni.

I dispositivi nvm non sono dotati di testine: sono quindi necessari algoritmi e ottimizzazioni differenti, perché l'utilizzo di un vecchio algoritmo che spende molti cicli della cpu cercando di ridurre il movimento di una testina che non esiste più sarebbe molto inefficiente. Per ottenere le massime prestazioni dai dispositivi nvm vengono modificati gli attuali file system e ne vengono creati di nuovi. Questi sviluppi mirano a ridurre il numero di istruzioni e il percorso complessivo tra il dispositivo di archiviazione e l'accesso dell'applicazione ai dati.

14.5 Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file, se possibile (i dischi ottici a una sola scrittura permettono una sola scrittura in qualsiasi settore e quindi il riutilizzo è fisicamente impossibile). Per tener traccia dello spazio libero in un disco, il sistema conserva una lista dello spazio libero; vi sono registrati tutti gli spazi *liberi*, cioè non allocati ad alcun file o directory. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista. Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati. A dispetto del suo nome, la lista dello spazio libero potrebbe non essere realizzata come una lista, come vedremo più avanti.

14.5.1 Vettore di bit

Spesso la lista dello spazio libero si realizza come una mappa di bit, o vettore di bit. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0.

Si consideri, per esempio, un disco dove i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 sono liberi e gli altri sono allocati. La mappa di bit dello spazio libero è la seguente:

001111001111100011000001110000...

I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o n blocchi liberi consecutivi nel disco; in effetti, molti calcolatori forniscono istruzioni di manipolazione dei bit utilizzabili con efficacia a tale scopo. Una tecnica per individuare il primo blocco libero su un sistema che usa un vettore di bit per allocare spazio su disco è controllare in modo sequenziale ogni parola nella mappa di bit per verificare che il valore non sia 0, poiché una parola con valore 0 ha tutti i bit a 0 e rappresenta un insieme di blocchi assegnati. La prima parola non 0 viene scandita alla ricerca del primo bit 1, che indica la locazione del primo blocco libero. Il numero del blocco è dato dalla seguente espressione:

$$(\text{numero di bit per parola}) \times (\text{numero di parole di valore } 0) + \text{offset del primo bit } 1.$$

Anche in questo caso le caratteristiche dell'hardware guidano le funzioni del sistema operativo. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, e viene scritto in memoria secondaria solo saltuariamente allo scopo di consentire eventuali operazioni di ripristino; è possibile tenere il vettore in memoria centrale solo se i dischi sono piccoli; tale soluzione non è applicabile ai dischi più grandi. Un disco di 1,3 gb con blocchi di 512 byte richiederebbe una mappa di bit di oltre 332 kb per tenere traccia dei suoi blocchi liberi, anche se il clustering a quattro blocchi riduce questo numero a 83 kb per disco. Un disco di 1 tb con blocchi di 4 kb richiede 32 mb ($2^{40} / 2^{12} = 2^{28}$ bit = 2^5 byte = 2^5 mb) per memorizzare la propria mappa di bit. Dato che la dimensione del disco è in costante crescita, i problemi legati ai vettori di bit continueranno ad aggravarsi.

14.5.2 Lista concatenata

Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Il primo blocco libero contiene un puntatore al successivo, e così via. Facciamo riferimento al nostro esempio precedente (Paragrafo 14.5.1) nel quale i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 erano liberi, mentre i restanti blocchi erano allocati. In questa situazione dovremmo mantenere un puntatore al blocco 2, trattandosi del primo blocco libero. Il blocco 2 conterebbe un puntatore al blocco 3, che punterebbe al blocco 4, che punterebbe al blocco 5, il quale punterebbe a sua volta al blocco 8, e così via (Figura 14.9). Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di i/o. Fortunatamente la necessità di attraversare la lista dello spazio libero non è frequente. Di solito il sistema operativo ha semplicemente bisogno di un blocco libero perché possa assegnarlo a un file, quindi si usa il primo blocco della lista. Il metodo che fa uso della fa include la lista dei blocchi liberi nella struttura dati per l'allocazione; non è necessario un metodo separato.

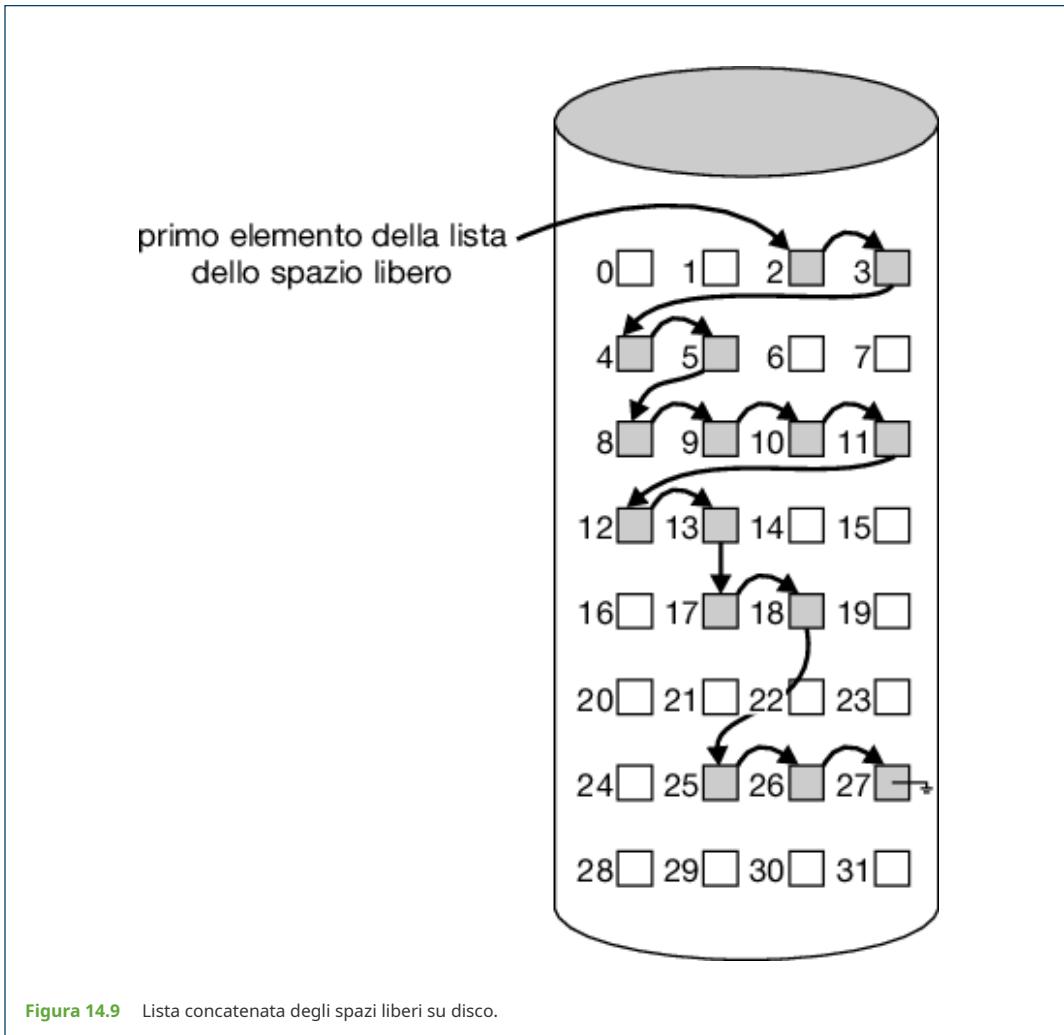


Figura 14.9 Lista concatenata degli spazi liberi su disco.

14.5.3 Raggruppamento

Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di n blocchi liberi nel primo di questi. I primi $n - 1$ di questi blocchi sono effettivamente liberi; l'ultimo blocco contiene gli indirizzi di altri n blocchi liberi, e così via. Con questo metodo, diversamente dall'ordinaria lista concatenata, è possibile trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

14.5.4 Conteggio

Un altro approccio sfrutta il fatto che, generalmente, più blocchi contigui si possono allocare o liberare contemporaneamente, soprattutto quando lo spazio viene allocato usando l'algoritmo di allocazione contigua o attraverso l'uso di cluster. Quindi, anziché tenere una lista di n indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero n di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dello spazio libero è formato da un indirizzo del disco e un contatore. Anche se ogni elemento richiede più spazio di quanto ne richieda un semplice indirizzo del disco, se il contatore è generalmente maggiore di 1 la lista complessiva è più corta. Si noti che questo metodo, il tracciamento dello spazio libero, è simile al metodo delle estensioni per allocare i blocchi. Questi elementi possono essere memorizzati in un albero bilanciato invece che in una lista concatenata, in modo da permettere ricerche, inserzioni e cancellazioni efficienti.

14.5.5 Mappe di spazio

Il file system zfs di Oracle (presente in Solaris e in altri sistemi operativi) è stato progettato per contenere un enorme numero di file, directory e persino di file system (in zfs è possibile creare gerarchie di file system). Su queste scale, l'i/o dei metadati può avere un impatto notevole sulle prestazioni. Notate per esempio che, se la lista dello spazio libero è implementata come una mappa di bit, le mappe di bit devono essere modificate sia quando i blocchi vengono allocati sia quando vengono liberati. Liberare 1 gb di dati su un

disco di 1 tb potrebbe comportare l'aggiornamento di migliaia di blocchi di mappe di bit, perché quei blocchi potrebbero essere sparpagliati sull'intero disco. Chiaramente, le strutture dati per un simile sistema potrebbero essere grandi e inefficienti.

Per la gestione dello spazio libero zfs utilizza una combinazione di tecniche per controllare la dimensione delle strutture di dati e minimizzare l'i/o necessario a gestirle. Per prima cosa, zfs crea metalastre (*metaslab*) per dividere lo spazio sul dispositivo in parti che abbiano una dimensione gestibile. Un dato volume potrebbe contenere centinaia di metalastre. A ogni metalastra è associata una mappa dello spazio. zfs utilizza l'algoritmo di conteggio per memorizzare informazioni riguardanti i blocchi liberi. Piuttosto che scrivere sempre su disco i contatori, li registra utilizzando le tecniche dei file system con logging delle modifiche. La mappa dello spazio è un registro di tutte le attività relative ai blocchi (allocazione e liberazione), in ordine cronologico, in formato di conteggio. Quando zfs decide di allocare o liberare spazio su una metalastra, carica la mappa dello spazio associata nella memoria, in una struttura ad albero bilanciato (per operazioni molto efficienti), indicizzato sulla base degli offset, e replica il log in tale struttura.

A quel punto la mappa dello spazio all'interno della memoria è un'accurata rappresentazione dello spazio allocato e libero nella metalastra. zfs condensa la mappa il più possibile combinando blocchi liberi contigui in una singola voce. Infine la lista dello spazio libero viene aggiornata sul disco come parte delle operazioni orientate alle transazioni di zfs. Durante la fase di raccolta e ordinamento possono verificarsi ancora richieste di blocchi, che vengono soddisfatte dal registro. In sostanza, il registro, insieme all'albero bilanciato, costituiscono la lista delle locazioni libere.

14.5.6 TRIM dei blocchi non utilizzati

Gli hdd e altri supporti di memorizzazione che consentono la sovrascrittura dei blocchi per effettuare aggiornamenti necessitano solo di una lista di blocchi liberi per poter gestire lo spazio libero. I blocchi non devono essere trattati in modo speciale una volta liberati e un blocco liberato conserva in genere i suoi dati (ma senza alcun puntatore al blocco) finché i dati non vengono sovrascritti una volta che il blocco viene successivamente riassegnato.

I dispositivi di archiviazione che non consentono la sovrascrittura, come i dispositivi nvm, basati su flash, risentono negativamente se vengono applicati gli stessi algoritmi. Ricordiamo dal Paragrafo 11.1.2 che tali dispositivi devono essere cancellati prima di poter essere scritti nuovamente, e che le cancellazioni devono essere fatte su grandi raggruppamenti di dati (blocchi, formati da pagine) e richiedono un tempo relativamente lungo rispetto a letture o scritture.

È necessario quindi un nuovo meccanismo per consentire al file system di segnalare al dispositivo che una pagina è libera e può essere considerata per la cancellazione (una volta che il blocco contenente la pagina è completamente libero). La tecnica utilizzata varia in base al controllore del dispositivo di archiviazione: sulle unità con collegamento ata viene utilizzato il comando trim, mentre per l'archiviazione basata su nvme si utilizza il comando `unallocate`. Qualunque sia il comando specifico del controllore, questo meccanismo mantiene lo spazio di archiviazione disponibile per la scrittura. Senza una tale funzionalità, il dispositivo di archiviazione si riempirebbe e richiederebbe la garbage collection e la cancellazione dei blocchi, con conseguente riduzione delle prestazioni dell'i/o di scrittura (questo problema è noto come "write cliff"). Con trim o funzionalità simili, la garbage collection e la cancellazione possono essere eseguite prima che il dispositivo arrivi a essere quasi del tutto pieno, consentendo al dispositivo di fornire prestazioni più uniformi.

14.6 Efficienza e prestazioni

Dopo avere descritto le opzioni di allocazione dei blocchi e di gestione delle directory, è possibile considerare i loro effetti sulle prestazioni e l'efficienza d'uso dei dischi. I dischi tendono di solito a essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti principali di un calcolatore. In questo paragrafo si considerano diverse tecniche utili per migliorare l'efficienza e le prestazioni della memoria secondaria.

14.6.1 Efficienza

L'uso efficiente di un disco dipende fortemente dagli algoritmi usati per l'allocazione del disco e la gestione delle directory. Per esempio, gli *inode* di unix sono preallocati su un volume. Quindi anche un disco "vuoto" impiega una certa percentuale del suo spazio per gli *inode*. D'altra parte, l'allocazione preventiva degli *inode* e la loro distribuzione nel volume migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocazione e di gestione dei blocchi liberi adottati da unix, i quali cercano di mantenere i blocchi di dati di un file vicini al blocco che ne contiene l'*inode* allo scopo di ridurre il tempo di ricerca.

Come ulteriore esempio, si consideri lo schema che fa uso dei cluster presentato nel Paragrafo 14.4, che migliora le prestazioni di ricerca e trasferimento per un file al costo di una maggiore frammentazione interna. Per ridurre questa frammentazione, il bsd unix varia la dimensione del cluster al crescere della dimensione del file. Cluster più grandi si usano dove possono essere riempiti, mentre cluster più piccoli si usano per file di piccole dimensioni e per l'ultimo cluster di un file. Questo sistema è descritto nell'Appendice C (reperibile sulla piattaforma MyLab).

Si devono tenere in considerazione anche il tipo di dati normalmente contenuti in un elemento della directory (o di un *inode*). Di solito si memorizza la *data dell'ultima scrittura* per fornire informazioni all'utente e per determinare se è necessario effettuare il backup del file. Alcuni sistemi mantengono anche la *data dell'ultimo accesso* per consentire all'utente di risalire all'ultima volta che un file è stato letto. Per mantenere questa informazione, ogni volta si legge un file, si deve aggiornare un campo della directory. Questa modifica richiede la lettura nella memoria del blocco, la modifica di una sua parte e la riscrittura del blocco nel disco, poiché sui dischi si può operare solamente per blocchi (o cluster). Quindi, ogni volta che si apre un file per la lettura, si deve leggere e scrivere anche l'elemento della directory a esso associato. Ciò può essere inefficiente per file cui si accede frequentemente, quindi nella fase della progettazione del file system è necessario confrontare i benefici con i costi in termini di prestazioni. In generale, è necessario considerare l'influenza sull'efficienza e sulle prestazioni di *ogni* informazione che si vuole associare a un file.

A titolo d'esempio, si consideri come l'efficienza sia influenzata dalle dimensioni dei puntatori usati per l'accesso ai dati. La maggior parte dei sistemi usa puntatori di 32 o 64 bit ovunque all'interno del sistema operativo. L'uso di puntatori di 32 bit limita la lunghezza di un file a 2^{32} byte (4 gb). I puntatori di 64 bit portano il limite a valori molto più grandi, ma i puntatori di 64 bit richiedono più spazio per la loro memorizzazione e di conseguenza fanno sì che i metodi di allocazione e di gestione dello spazio libero (liste concatenate, indici, e così via) impieghino più spazio.

Una delle difficoltà nella scelta della dimensione dei puntatori, o di qualsiasi altra dimensione di allocazione fissa all'interno di un sistema operativo, è la pianificazione degli effetti provocati dal cambiamento della tecnologia. Basti considerare che il primo ibm pc xt aveva un disco di 10 mb e che il file system dell'ms-dos poteva gestire solamente 32 mb. (Ciascun elemento della fat era di 12 bit e puntava a un cluster di 8 kb.) Con la crescita della capacità dei dischi, i dischi più grandi si dovevano suddividere in partizioni di 32 mb, poiché il file system non poteva tener traccia di blocchi disposti oltre i 32 mb. Quando divennero comuni dischi di capacità superiore ai 100 mb, si dovettero modificare le strutture dati e gli algoritmi usati dall'ms-dos per gestire i dischi in modo da consentire file system più grandi. (La dimensione di ciascun elemento della fat fu portata a 16 bit e più tardi a 32 bit.) La decisione iniziale fu presa per motivi di efficienza; tuttavia, con l'avvento della Versione 4 dell'ms-dos milioni di utenti si trovarono a disagio quando dovettero passare ai nuovi, più grandi file system. Il file system zfs di Solaris adotta puntatori di 128 bit, che in teoria non dovrebbero mai necessitare di un'estensione. (La minima massa di un dispositivo in grado di archiviare 2^{128} byte tramite memorizzazione al livello atomico è di circa 272 miliardi di tonnellate.)

Come altro esempio, si consideri l'evoluzione del sistema operativo Solaris. Originariamente molte strutture dati avevano lunghezza fissa ed erano assegnate all'avviamento del sistema. Queste strutture comprendevano la tabella dei processi e quella dei file aperti. Una volta riempie queste tabelle, non si potevano più creare nuovi processi o aprire nuovi file, sicché il sistema non riusciva nel proprio compito di fornire un servizio agli utenti. L'unico modo di aumentare le dimensioni di queste tabelle era la ricompilazione del kernel e il riavvio del sistema. Nelle versioni più recenti di Solaris quasi tutte le strutture dati del kernel si assegnano in modo dinamico, eliminando questi limiti artificiali alle prestazioni del sistema. Naturalmente, gli algoritmi che manipolano queste tabelle sono ora più complessi e il sistema operativo è un po' più lento dovendo allocare e rilasciare dinamicamente gli elementi delle tabelle, ma questo è il prezzo da pagare per la generalizzazione delle funzioni.

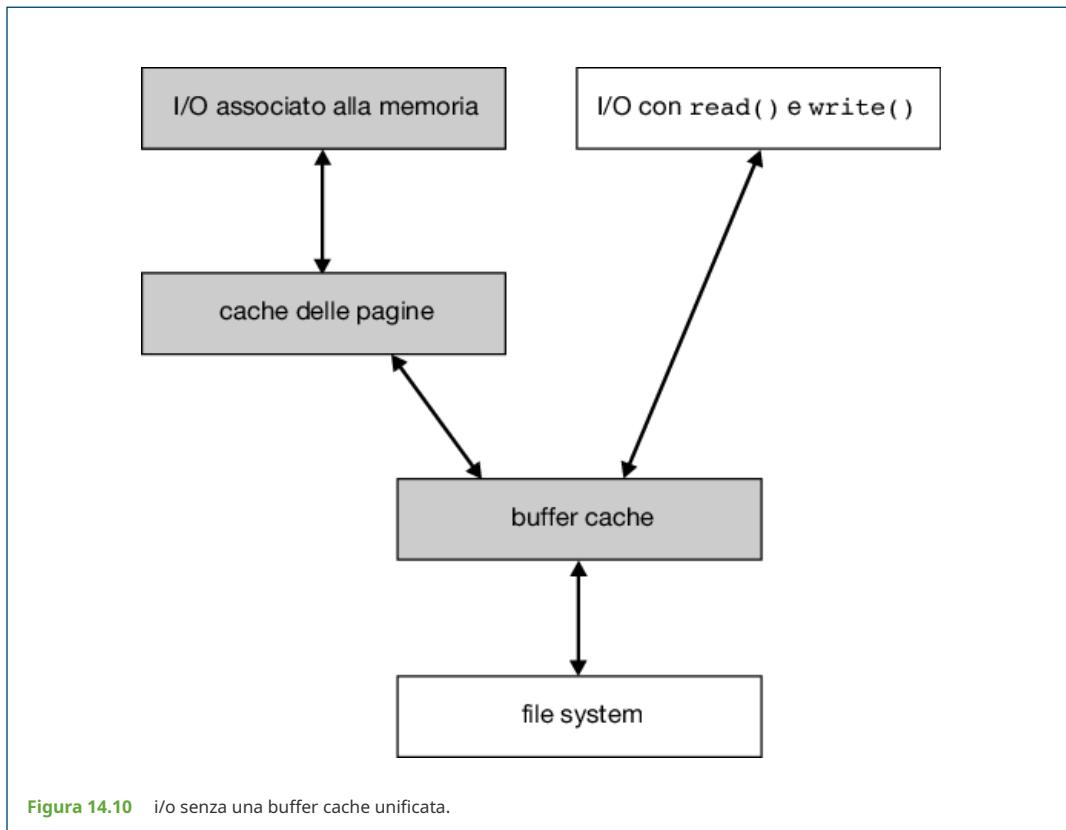
14.6.2 Prestazioni

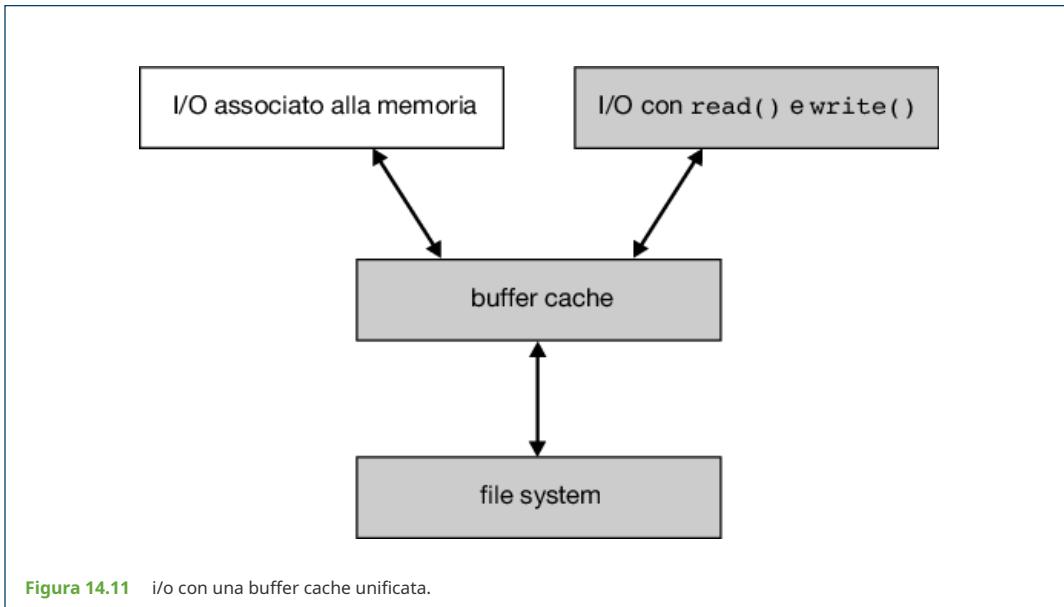
Anche dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi. Come si è osservato nel Capitolo 12, alcuni controllori di unità a disco contengono una memoria locale sufficiente per la creazione di una cache interna al controllore abbastanza grande da memorizzare intere tracce del disco alla volta. Eseguito il posizionamento della testina, si legge la traccia nella cache del controllore del disco a partire dal settore sotto cui si viene a trovare la testina (riducendo il tempo di latenza). Il controllore trasferisce quindi al sistema operativo tutte le richieste di settori. Quando i blocchi sono trasferiti dal controllore del disco alla memoria centrale, il sistema operativo ha la possibilità di inserirli in una propria cache nella memoria centrale.

Alcuni sistemi riservano una sezione separata della memoria centrale come buffer cache (*cache del disco*), dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una cache delle pagine per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi del file system; l'uso degli indirizzi virtuali per effettuare il caching dei dati dei file è molto più efficiente dell'effettuare il caching dei blocchi fisici di

disco, in quanto gli accessi avvengono attraverso la memoria virtuale e non attraverso il file system. Diversi sistemi, compreso Solaris, Linux e Windows usano le cache delle pagine, sia per le pagine relative ai processi sia per i dati dei file. Questo metodo è noto come memoria virtuale unificata.

Alcune versioni di unix e Linux prevedono la cosiddetta buffer cache unificata. Per illustrarne i vantaggi si considerino le due possibilità di aprire un file e accedervi: l'uso della mappatura dei file in memoria (Paragrafo 13.5) e l'uso delle ordinarie chiamate di sistema `read()` e `write()`. Senza una buffer cache unificata, si verifica una situazione simile a quella illustrata nella Figura 14.10. In questo caso, le chiamate di sistema `read()` e `write()` passano attraverso la buffer cache. La chiamata con i/o memory-mapped, tuttavia, richiede l'uso di due cache, la cache delle pagine e la buffer cache. Il memory-mapping prevede la lettura dei blocchi di disco dal file system e la loro memorizzazione nella buffer cache. Poiché il sistema di memoria virtuale non si interfaccia direttamente con la buffer cache, si deve copiare nella cache delle pagine il contenuto del file presente nella buffer cache. Questa situazione è nota come double caching proprio perché i dati del file system richiedono un doppio passaggio di cache. Non solo ciò comporta uno spreco di memoria, ma anche uno spreco di cicli della cpu e di i/o dovuti a un ulteriore trasferimento di dati nella memoria del sistema. Inoltre, eventuali incoerenze tra le due cache possono generare errori nella memorizzazione dei dati nei file. Invece, con una buffer cache unificata, sia il memory-mapping sia le chiamate di sistema `read()` e `write()` usano la stessa cache delle pagine, con il vantaggio di evitare il double caching e di permettere al sistema di memoria virtuale di gestire dati del file system. La Figura 14.11 illustra l'uso della buffer cache unificata.





Indipendentemente dalla gestione delle cache per blocchi di disco oppure per pagine (o per entrambi), l'algoritmo lru (Paragrafo 10.4.4) sembra essere un ragionevole algoritmo generale per la sostituzione dei blocchi o delle pagine. Tuttavia, l'evoluzione degli algoritmi di gestione delle cache delle pagine usati dal sistema operativo Solaris rivela le difficoltà nella scelta di un algoritmo ottimale. Tale sistema operativo permette ai processi e alla cache delle pagine di condividere la memoria inutilizzata; prima della versione 2.5.1, non si facevano distinzioni tra l'allocazione delle pagine a un processo o alla cache delle pagine, con la conseguenza che un sistema che eseguiva molte operazioni di i/o usava la maggior parte della memoria disponibile per la cache delle pagine. A causa dell'alta frequenza delle operazioni di i/o, quando la memoria libera diventa troppo esigua, il modulo di scansione delle pagine (Paragrafo 10.10.3) sottraeva pagine ai processi anziché alla cache delle pagine. In Solaris 2.6 e in Solaris 7 è stata realizzata in forma opzionale la tecnica di *paginazione con priorità*, secondo la quale il modulo di scansione delle pagine dà la priorità alle pagine dei processi rispetto a quelle della cache delle pagine. Il sistema operativo Solaris 8 ha applicato un limite prefissato alle pagine dei processi e alla cache delle pagine per il file system, impedendo a ciascun meccanismo di sottrarre totalmente la memoria all'altro. Con Solaris 9 e 10 sono stati nuovamente modificati gli algoritmi per migliorare l'uso della memoria e contenere il fenomeno del thrashing.

Un altro aspetto che può influenzare le prestazioni di i/o è l'utilizzo di scritture sincrone o asincrone nel file system. Le scritture sincrone avvengono nell'ordine in cui le riceve il sottosistema per la gestione del disco e non subiscono la memorizzazione transitoria. Quindi la procedura chiamante prima di proseguire deve attendere che i dati raggiungano l'unità disco. Nelle scritture asincrone si memorizzano i dati nella cache e si restituisce immediatamente il controllo alla procedura chiamante. Le scritture sono per lo più asincrone, anche se le scritture dei metadati, tra le altre, possono essere sincrone. I sistemi operativi spesso includono un flag nella chiamata di sistema `open()` per permettere a un processo di richiedere che le operazioni di scrittura si eseguano in modo sincrono. I sistemi di gestione delle basi di dati per esempio usano questa funzione per realizzare le transazioni atomiche, in modo da assicurare che i dati raggiungano la memoria stabile nell'ordine richiesto.

Alcuni sistemi ottimizzano la cache delle pagine adottando differenti algoritmi di sostituzione a seconda del tipo d'accesso ai file. Le pagine relative a un file da leggere o scrivere in modo sequenziale non si dovrebbero sostituire nell'ordine lru, infatti la pagina usata più di recente sarà usata nuovamente per ultima, o forse mai. Gli accessi sequenziali si potrebbero invece ottimizzare con tecniche note come rilascio all'indietro e lettura anticipata. Il rilascio all'indietro (*free-behind*) rimuove una pagina dal buffer non appena si verifica una richiesta della pagina successiva; le pagine precedenti con tutta probabilità non saranno più usate e quindi sprecano spazio nel buffer. Con la lettura anticipata (*read-ahead*) si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive: è probabile che queste pagine siano richieste una volta terminata l'elaborazione della pagina corrente. Il recupero di questi dati dal disco con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole. Si noti che la presenza nel controllore di una cache per le tracce non elimina la necessità di adottare la tecnica di lettura anticipata in un sistema multiprogrammato, infatti, a causa dell'elevata latenza e dell'overhead determinato dai tanti piccoli trasferimenti dalla cache per le tracce alla memoria centrale, il ricorso alla *lettura anticipata* resta vantaggioso.

La cache delle pagine, il file system e i driver del disco interagiscono in modi interessanti. Quando i dati vengono scritti in un file su disco, le pagine sono memorizzate nella cache e il driver del disco ordina la propria coda di dati in base all'indirizzo sul disco. Queste due azioni consentono al driver del disco di minimizzare gli spostamenti della testina del disco e fanno sì che la scrittura dei dati rispetti i tempi ottimali per la rotazione del disco. A meno che le scritture richieste siano sincrone, un processo, per scrivere sul disco, scrive semplicemente nella cache e il sistema trasferisce i dati su disco, in maniera asincrona, quando lo ritiene opportuno. Dal punto di vista del processo utente, le scritture sembreranno estremamente rapide. Durante la lettura, il sistema di i/o per i blocchi su disco effettua qualche lettura anticipata; ma, in realtà, le scritture sono molto più asincrone delle letture. Per grandi quantità di dati, quindi, la scrittura su disco tramite il file system è spesso più veloce della lettura, contrariamente a ciò che suggerirebbe l'intuizione.

14.7 Ripristino

Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati o la loro incoerenza.

Una caduta del sistema può causare incoerenze tra le strutture dati del file system su disco, come le strutture delle directory, i puntatori ai blocchi liberi e i puntatori agli fcb liberi. Molti file system applicano delle modifiche direttamente a queste strutture. Operazioni comuni come la creazione di un file possono comportare molti cambiamenti strutturali all'interno del file system di un disco. Le strutture delle directory vengono modificate, gli fcb e i blocchi di dati allocati e i contatori degli elementi liberi per tutti questi blocchi diminuiti. Quando queste modifiche sono interrotte da una caduta del sistema, ne possono derivare incoerenze tra le strutture. Per esempio, il contatore degli fcb liberi potrebbe indicare che un fcb è stato allocato, ma la struttura della directory potrebbe non avere un puntatore a quel fcb. L'utilizzo della cache che i sistemi operativi adottano per ottimizzare le prestazioni di i/o aggrava questo problema. Alcuni cambiamenti potrebbero andare direttamente sul disco, mentre altri possono essere memorizzati nella cache. Se i cambiamenti nella cache non raggiungono il disco prima che si verifichi una caduta, è possibile che la situazione peggiori ulteriormente.

Inoltre, anche i bachi nell'implementazione del file system, i controllori del disco, e persino le applicazioni utente possono indurre incoerenze nel file system. I file system hanno svariati metodi per affrontare queste circostanze, a seconda delle strutture dati e degli algoritmi che utilizzano. Ci occuperemo adesso di questi temi.

14.7.1 Verifica della coerenza

Quale che sia la causa degli errori, un file system deve prima scoprire i problemi e poi correggerli. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Sfortunatamente, questo procedimento può richiedere diversi minuti, o addirittura delle ore, e dovrebbe avvenire tutte le volte che il sistema si avvia. In alternativa, un file system può registrare il suo stato all'interno dei metadati. All'inizio di ogni serie di modifiche dei metadati è impostato un bit di stato per indicare che i metadati sono in stato di modifica. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tuttavia il bit dello stato rimane settato, entra in funzione un verificatore della coerenza.

Il verificatore della coerenza – un programma di sistema come `fsck` in unix – confronta i dati delle directory con i blocchi dati dei dischi, tentando di correggere ogni incoerenza. Gli algoritmi di allocazione e di gestione dello spazio libero determinano il genere di problemi che questo programma può riconoscere e con quanto successo riuscirà a risolverli. Per esempio, se si adotta uno schema di allocazione concatenata con un puntatore da ciascun blocco al successivo, si può ricostruire l'intero file e ricreare il corrispondente elemento nella directory analizzando i blocchi di dati. Al contrario, la perdita di un elemento di una directory in un sistema ad allocazione indicizzata potrebbe essere disastrosa, poiché ogni blocco di dati non contiene alcuna informazione sugli altri blocchi di dati. Per questo motivo, unix gestisce cache gli elementi delle directory per le letture, mentre qualsiasi operazione di scrittura che produca l'allocazione di spazio, o altre modifiche dei metadati, è svolta in modo sincrono, prima della scrittura dei corrispondenti blocchi di dati. Naturalmente possono ancora insorgere dei problemi se una scrittura sincrona viene interrotta da una caduta del sistema.

Alcuni dispositivi di memoria NVM contengono una batteria o supercondensatori ad alta capacità in grado di fornire energia durante una perdita di potenza o trasferire dati dai buffer di dispositivo ai supporti di memorizzazione per evitare la perdita di dati. Tuttavia anche queste precauzioni non evitano i danni di una caduta del sistema.

14.7.2 File system con log delle modifiche

Spesso nell'informatica si adottano algoritmi e tecnologie anche in aree diverse da quelle per le quali sono stati progettati. È il caso degli algoritmi per il ripristino, utilizzati per le basi di dati, basati sulla registrazione (*log*) delle modifiche. Questi algoritmi sono stati applicati con successo al problema della verifica della coerenza, realizzando i file system orientati alle transazioni e basati sul log delle modifiche (*log-based transaction-oriented file system*), noti anche come file system annotati (*journaling file system*).

Si osservi che l'approccio alla verifica della coerenza esaminato precedentemente permette in sostanza alle strutture di esibire incoerenze successivamente corrette nel ripristino. Questa strategia comporta tuttavia vari problemi. Per esempio, l'incoerenza potrebbe rivelarsi irreparabile: il verificatore della coerenza potrebbe non essere in grado di ripristinare le strutture, con una conseguente perdita di file o addirittura di intere directory. Ancora, il verificatore della coerenza potrebbe richiedere l'intervento umano per risolvere i conflitti, il che causa inconvenienti: in mancanza di assistenza da parte di qualcuno, il sistema potrebbe rimanere inutilizzabile finché una persona non gli indichi come procedere. Infine, il verificatore della coerenza sottrae risorse al sistema: per controllare terabyte di dati possono essere necessarie molte ore.

La soluzione consiste nell'applicare agli aggiornamenti dei metadati del file system metodi di ripristino basati sul log delle modifiche. Sia il file system ntfs sia il Veritas usano questo metodo, che è anche incluso nelle recenti versioni di ufs su Solaris, un metodo ormai comune in molti sistemi operativi.

Fondamentalmente, tutte le modifiche dei metadati si registrano in modo sequenziale in un file di log. Ogni insieme di operazioni che esegue uno specifico compito si chiama transazione. Una volta che le modifiche sono riportate nel file di log, le operazioni si considerano portate a termine con successo (*committed*) e la chiamata di sistema può restituire il controllo al processo utente, permettendogli di proseguire la sua esecuzione. Nel frattempo, si applicano alle effettive strutture del file system le operazioni scritte nel log, e man mano che si eseguono si aggiorna un puntatore che indica quali azioni sono state completate e quali sono ancora incomplete. Quando un'intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare. I buffer circolari scrivono fino alla fine dello spazio disponibile, e poi ricominciano dall'inizio, sovrascrivendo i vecchi contenuti. Naturalmente, si devono prendere delle misure per evitare che dati non ancora salvati siano sovrascritti. Il log si può

mantenere in una sezione separata del file system, o anche in un disco separato. E più efficiente, anche se è più complesso, averlo sotto testine di lettura e scrittura separate, poiché si riducono le situazioni di contesa della testina e i tempi di ricerca (*seek time*).

Se si verifica una caduta del sistema, nel log ci potranno essere zero o più transazioni. Le transazioni presenti non sono mai state ultimate nel file system, anche se il sistema operativo le definisce portate a termine con successo, e quindi si devono completare. Le transazioni si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento, in modo che le strutture del file system rimangano coerenti. L'unico problema che si può presentare è il caso in cui una transazione sia fallita (*aborted*), cioè non sia stata dichiarata terminata con successo prima della caduta del sistema. In questo caso, si devono annullare tutti i cambiamenti che erano stati applicati al file system dalla transazione, mantenendo anche in questo caso la coerenza del file system. Questo ripristino è tutto ciò che è necessario fare dopo una caduta del sistema, eliminando tutti i problemi della verifica della coerenza.

Un vantaggio indiretto dell'uso del logging degli aggiornamenti dei metadati dei dischi è che gli aggiornamenti sono molto più rapidi di quando si applicano direttamente alle strutture dati nei dischi. La ragione di questo miglioramento sta nel vantaggio, dal punto di vista delle prestazioni, dell'i/o ad accesso sequenziale rispetto a quello ad accesso diretto. Le onerose operazioni di scrittura sincrona ad accesso diretto sui metadati vengono sostituite con molto meno gravose operazioni di scrittura sequenziali sincrone nell'area di log di un file system annotato. I cambiamenti determinati da quelle operazioni si riportano successivamente in modo asincrono nelle strutture appropriate nei dischi attraverso operazioni di scrittura ad accesso diretto. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file su disco rigido.

14.7.3 Altre soluzioni

Un'altra alternativa alla verifica della coerenza è impiegata dal file system wafl di Network Appliance e da zfs di Solaris. Entrambi i sistemi non sovrascrivono mai i blocchi con nuovi dati; al contrario, una transazione scrive tutti i cambiamenti di dati e metadati su nuovi blocchi. Quando la transazione viene completata, le strutture dei metadati che puntavano alla vecchia versione di questi blocchi sono aggiornate in modo da puntare ai nuovi. Il file system può quindi rimuovere i vecchi puntatori e i vecchi blocchi per renderli nuovamente disponibili. Se i vecchi puntatori e i vecchi blocchi vengono mantenuti, viene creata una snapshot (*instantanea*), cioè un'immagine del file system prima dell'ultimo aggiornamento. Questa soluzione non dovrebbe richiedere una verifica della coerenza se l'aggiornamento del puntatore è eseguito atomicamente. Ciononostante, il wafl possiede comunque un controllore della coerenza, perché alcuni tipi di malfunzionamento possono ancora causare un errore nei metadati (per dettagli sul sistema wafl si veda il Paragrafo 14.8).

Lo zfs adotta un approccio ancora più innovativo alla coerenza del disco. Come wafl, zfs non sovrascrive mai i blocchi, ma è in grado di andare oltre fornendo il checksumming di tutti i metadati e i blocchi di dati. Questa soluzione (se combinata con raid) assicura che i dati siano sempre corretti. zfs non possiede quindi un controllore della coerenza. (Maggiori dettagli su zfs sono presenti nel Paragrafo 11.8.6.)

14.7.4 Copie di riserva e recupero dei dati

I dischi magnetici sono soggetti a guasti, ed è necessario preoccuparsene e provvedere affinché in tal caso i dati non vadano persi definitivamente. A questo scopo si possono usare programmi di sistema che consentano di fare delle copie di riserva (*backup*) dei dati residenti nei dischi su altri dispositivi di memorizzazione, come nastri magneticici o hard disk supplementari. Il ripristino della situazione antecedente la perdita di un singolo file, o del contenuto di un intero disco, richiederà il recupero (*restore*) dei dati dalle copie di backup.

Al fine di ridurre al minimo la quantità di dati da copiare, è possibile sfruttare le informazioni contenute nell'elemento della directory associato a ogni file. Per esempio, se il programma di creazione delle copie di riserva si quando è stata eseguita l'ultimo backup di un file, e se la data di ultima scrittura di quel file, registrata nella directory, indica che il file da quel momento non ha subito variazioni, non sarà necessario copiare nuovamente il file. Quella che segue è una tipica sequenza di gestione dei backup.

- Giorno 1. Copiatura nel supporto di backup di tutti i file contenuti nel disco; questo è detto backup completo.
- Giorno 2. Copiatura su un altro supporto di tutti i file modificati dal Giorno 1; questo è un backup incrementale.
- Giorno 3. Copiatura su un altro supporto di tutti i file modificati dal Giorno 2.
- ...
- Giorno n. Copiatura su un altro supporto di tutti i file modificati dal Giorno $n - 1$. Ritorno al Giorno 1.

Il nuovo ciclo può scrivere le nuove copie riutilizzando il precedente insieme di supporti di backup, oppure in un nuovo insieme.

Utilizzando questo metodo si ha la possibilità di recuperare il contenuto dell'intero disco iniziando le operazioni di recupero dalla copia di backup completa e proseguendo con i backup incrementali. Naturalmente, più grande è n , maggiore sarà il numero di nastri o dischi da leggere per un completo recupero. Un ulteriore vantaggio di questo ciclo di creazione di copie di riserva è la possibilità di recuperare qualsiasi file accidentalmente cancellato durante il ciclo, recuperandolo dalle copie del giorno precedente.

La lunghezza del ciclo è un compromesso tra la quantità di supporti necessari per i backup e il numero di giorni coperti da un'operazione di recupero. Una possibilità per diminuire la quantità di nastri che è necessario leggere per portare a termine il ripristino è di eseguire inizialmente un backup completo, per poi eseguire ogni giorno il backup di tutti i file modificati dal giorno del backup completo. In questo modo, il ripristino può fondarsi sull'ultimo backup incrementale, insieme all'ultimo backup completo, senza necessità di altri backup incrementali. Qui il compromesso da tenere a mente è che il numero di file modificati aumenta giornalmente: ogni nuovo backup incrementale, quindi, richiede più spazio.

Un utente potrebbe accorgersi dopo molto tempo che si è perso o si è danneggiato un particolare file. Per questa ragione si è soliti pianificare di tanto in tanto un backup completo che sarà conservato in modo permanente. È opportuno conservare tali backup permanenti lontano dalle copie ordinarie per proteggerle dai vari pericoli, come un incendio che può distruggere il calcolatore e tutte le copie di backup. Se il ciclo di creazione delle copie di backup prevede il reimpiego dei mezzi che le contengono, è anche necessario aver cura di non usarli troppe volte: se dovessero danneggiarsi per l'uso, potrebbe essere impossibile ripristinare i dati in essi contenuti. Nello show televisivo "Mr. Robot", gli hacker non hanno attaccato solamente le fonti primarie dei dati bancari, ma anche i loro siti di backup. Avere più siti di backup potrebbe non essere una cattiva idea quando i tuoi dati sono importanti.

14.8 Esempio: il file system WAFL

L'i/o da e verso il disco si riflette significativamente sulle prestazioni del sistema. Di conseguenza, i progettisti devono esercitare grande cura sia nella progettazione sia nell'implementazione del file system. Alcuni file system sono concepiti per finalità generali, ossia sono in grado di offrire prestazioni accettabili e funzionalità adatte a file che differiscono per tipo e dimensione, e a carichi di i/o diversi. Altri sono ottimizzati per compiti specifici, nel tentativo di fornire, per alcune applicazioni dedicate, prestazioni migliori di quelle dei sistemi a carattere generale. Un'ottimizzazione di questo genere è rappresentata dal file system wafl di NetApp Inc. wafl, acronimo di *write-anywhere file layout* ("modello di file per la scrittura ovunque"), è un file system potente ed elegante, ottimizzato per le scritture casuali.

È utilizzato in esclusiva dai file server di rete prodotti da NetApp, ed è pensato per essere utilizzato come file system distribuito. Benché sia stato originariamente progettato per i soli protocolli nfs e cfs, consente l'invio di file ai client tramite nfs, cifs, ftp e http. Quando molti client contattano un file server attraverso questi protocolli, le richieste di lettura casuali, e ancor di più quelle relative a scritture casuali, aumentano sensibilmente. I protocolli nfs e cifs utilizzano il caching dei dati provenienti dalle operazioni in lettura: per i file server, dunque, è la scrittura che assume la massima importanza.

L'impiego del wafl presuppone che i file server dispongano di una cache nvram per le scritture. I progettisti del wafl hanno sfruttato il vantaggio di lavorare su un'architettura specifica, con una cache per la memorizzazione stabile dei dati, al fine di ottimizzare il file system per l'i/o ad accesso casuale. Uno dei principi che hanno ispirato il wafl è la facilità d'uso. I suoi autori, inoltre, hanno aggiunto una nuova funzionalità di duplicazione istantanea (*snapshot*), per creare in più momenti, come vedremo, diverse copie a sola lettura del file system.

Il file system è simile al Berkeley Fast, con varie modifiche; è basato sui blocchi e usa gli inode per la descrizione dei file. Ciascun inode contiene 16 puntatori ad altrettanti blocchi (o blocchi indiretti), che appartengono al file descritto dall'inode. Ciascun file system possiede un inode radice. Come si vede nella Figura 14.12, tutti i metadati sono custoditi all'interno di file: un file per gli inode, un altro per la mappa dei blocchi liberi e un terzo per la mappa degli inode liberi. Poiché si tratta di file ordinari, i blocchi di dati non hanno un indirizzo predefinito, ma possono risiedere dovunque. Se un file system viene ampliato con l'aggiunta di dischi, esso aumenterà in modo automatico la lunghezza di questi file per i metadati.

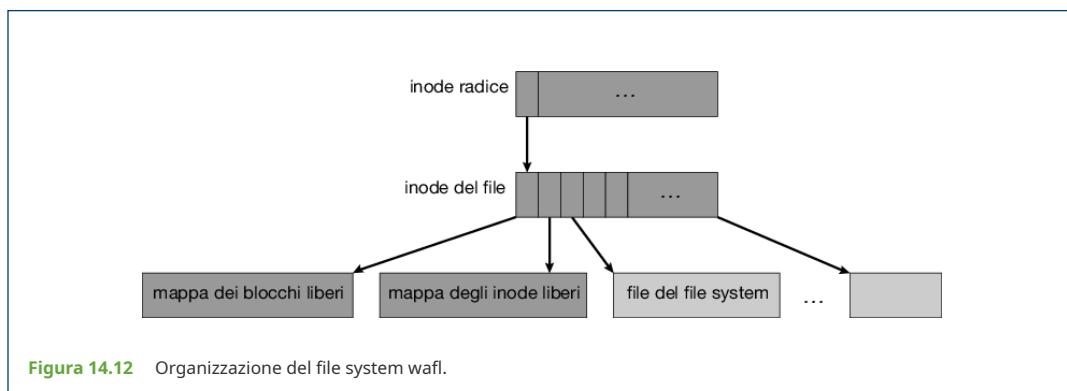


Figura 14.12 Organizzazione del file system wafl.

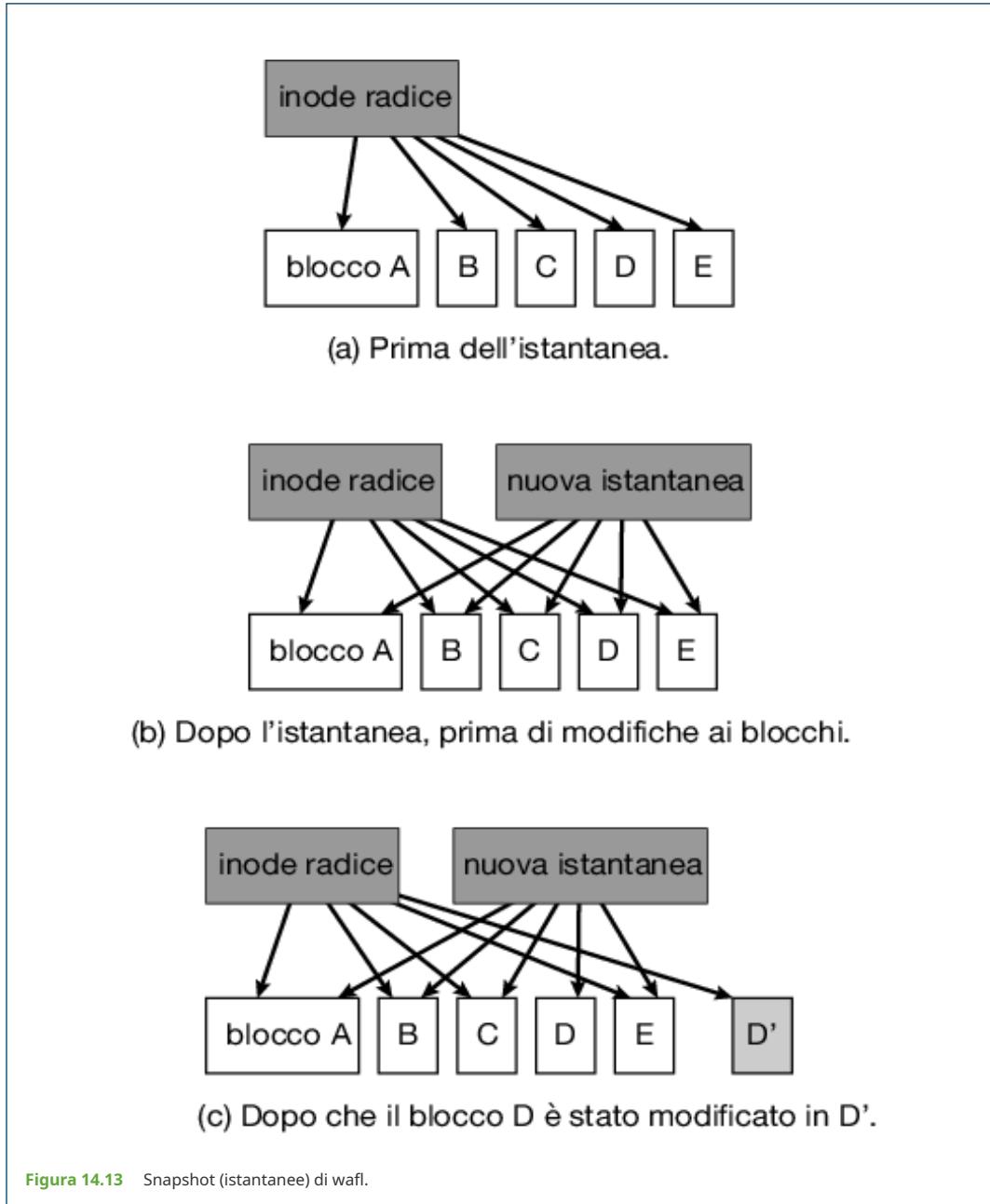
Pertanto, un file system wafl è un albero di blocchi che si dipartono dall'inode radice; per creare uno snapshot, wafl crea una copia dell'inode radice. In seguito a ciò, ogni aggiornamento dei file o dei metadati occuperà blocchi nuovi, anziché sovrascrivere i relativi blocchi esistenti. Il nuovo inode radice punta ai metadati e ai dati nella loro versione aggiornata. Nello stesso tempo, il vecchio inode radice continua a puntare ai blocchi precedenti, non aggiornati, e in tal modo dà accesso a un'immagine del file system che ricalca esattamente il momento in cui era stato fotografato; lo spazio occupato sul disco per questa operazione è davvero esiguo. In sostanza, lo snapshot richiede un supplemento di spazio sul disco equivalente ai soli blocchi che siano stati modificati dal momento in cui si è scattata l'istantanea.

Una differenza di rilievo in confronto a file system più tradizionali è data dalla mappa dei blocchi liberi, che possiede più di un bit per blocco. Si tratta di una maschera di bit che prevede un bit impostato a uno a ogni istantanee che stia utilizzando il blocco. Quando tutte le istantanee che stavano utilizzando un blocco sono cancellate, la maschera di bit associata è formata da zeri, e il blocco può essere riutilizzato.

I blocchi usati non sono mai sovrascritti, di modo che le scritture avvengono a gran velocità, visto che le operazioni in scrittura possono sfruttare il blocco libero più vicino alla posizione corrente della testina. Vi sono, nel wafl, molti altri meccanismi per ottimizzare le prestazioni.

Possono coesistere allo stesso tempo numerose istantanee: se ne può scattare una per ogni ora del giorno e ogni giorno del mese. Gli utenti abilitati ad accedere a queste istantanee, hanno accesso a ciascuno dei file per come era nei momenti in cui è stato fotografato. Questa funzionalità è anche utile per il backup, il testing, per gestire diverse versioni di un progetto, e altro ancora. L'implementazione degli snapshot in wafl è molto efficiente, dato che non richiede neppure di duplicare con la copiatura su scrittura ciascun blocco di

dati prima che sia modificato. Altri file system offrono la medesima funzionalità, ma spesso con minor efficienza. Le istantanee del wafl sono descritte nella Figura 14.13.



Le versioni più recenti del wafl permettono istantanee di lettura e scrittura, note come cloni. Come le istantanee, anche i cloni sono efficienti, in quanto utilizzano le stesse tecniche. In questo caso, uno snapshot di sola lettura cattura lo stato del file system e il clone fa riferimento a quello snapshot. Eventuali scritture sul clone sono memorizzate in nuovi blocchi e i puntatori del clone vengono aggiornati per fare riferimento ai nuovi blocchi. L'istantanea originale non è modificata, mantenendo così l'immagine del file system prima dell'aggiornamento del clone. I cloni possono essere “promossi” a sostituti del file system originale e ciò comporta l'eliminazione dei vecchi puntatori e di ogni vecchio blocco a essi associato. I cloni sono utili per il testing e gli aggiornamenti; la versione originale rimane infatti immutata e il clone viene cancellato quando il test è stato effettuato o quando l'aggiornamento fallisce.

IL FILE SYSTEM APPLE

Nel 2017 Apple ha rilasciato un nuovo file system per sostituire hfs+, introdotto più di 30 anni or sono. hfs+ è stato ampliato negli anni per aggiungere molte nuove funzionalità, ma come spesso accade questo processo ha aggiunto complessità e linee di codice e reso più difficile l'aggiunta di ulteriori funzionalità. Ripartire da zero consente a un progetto di iniziare da subito con le tecnologie e le metodologie più attuali e di fornire esattamente le funzionalità necessarie.

Apple File System (apfs) è un buon esempio di tale impostazione. Il suo obiettivo è quello di funzionare su tutti i dispositivi Apple attuali, da Apple Watch, a iPhone, ai computer Mac. Creare un file system che funzioni in watchos, ios, tvos e macos è certamente una sfida. apfs è ricco di funzionalità, inclusi puntatori a 64 bit, i cloni per file e directory, le istantanee, la condivisione dello spazio, il dimensionamento rapido delle directory, le primitive per operazioni atomiche sicure, il copy-on-write, la crittografia (singola e multipla), e la fusione (*coalescing*) dell'i/o. Inoltre è in grado di funzionare sia su nvm sia su hdd.

La maggior parte di queste funzioni è già stata discussa, ma ci sono alcuni nuovi concetti che vale la pena di esplorare.

La condivisione dello spazio è una funzionalità simile a quella di cui dispone zfs, in cui lo spazio di archiviazione è reso disponibile in forma di uno o più spazi liberi di grandi dimensioni (**container**, o **contenitori**) dai quali i file system possono effettuare allocazioni (consentendo ai volumi con formattazione apfs di aumentare e diminuire in dimensione).

Il dimensionamento rapido delle directory permette il calcolo veloce dello spazio utilizzato e il suo aggiornamento. Il **salvataggio atomico sicuro** è una primitiva (disponibile tramite api, non tramite i comandi del file system) che esegue la rinomina di file, di gruppi di file e di directory come singole operazioni atomiche. La fusione dell'i/o è un'ottimizzazione per i dispositivi nvm in cui diverse scritture di piccola entità vengono fuse insieme in un'unica scrittura di grandi dimensioni al fine di ottimizzare le prestazioni.

Apple ha scelto di non implementare il raid come parte del nuovo apfs, ma piuttosto di continuare a dipendere dal preesistente strumento Apple per la gestione di volumi raid via software. apfs è inoltre compatibile con hfs+, consentendo così la facile conversione di implementazioni esistenti.

Un'altra caratteristica che si ottiene naturalmente dall'implementazione del file system wafl è la replica, ovvero la duplicazione e la sincronizzazione di un insieme di dati su un altro sistema attraverso una rete. Per prima cosa, lo snapshot di un file system wafl viene duplicato su un altro sistema. Quando si genera un altro snapshot del sistema sorgente, per aggiornare il sistema remoto è sufficiente inviare tutti i blocchi contenuti nel nuovo snapshot. Questi blocchi sono quelli che hanno subito modifiche nell'intervallo di tempo tra lo scatto delle due istantanee. Il sistema remoto aggiunge questi blocchi al file system e aggiorna i propri puntatori. Il nuovo sistema è dunque un duplicato del sistema sorgente nel momento in cui è stata generato il secondo snapshot. La ripetizione di questo processo fa sì che il sistema remoto sia una copia quasi aggiornata del primo sistema. Le repliche sono utili per il *disaster recovery*. Nel caso in cui il primo sistema sia vittima di una perdita totale di dati, la maggior parte dei suoi dati sarebbe comunque disponibile nella replica sul sistema remoto.

Infine è opportuno ricordare che il file system zfs supporta snapshot, cloni e un sistema di repliche altrettanto efficienti, e che queste caratteristiche stanno diventando sempre più comuni nei moderni file systems.

14.9 Sommario

- La maggior parte dei file system risiede permanentemente in memoria secondaria, progettata per contenere, permanentemente, una grande quantità di dati. Il più comune mezzo di memoria secondaria è il disco, ma sta crescendo notevolmente anche l'utilizzo dei dispositivi nvm.
- I dischi si possono segmentare in partizioni, allo scopo di controllarne l'uso e consentire più file system, anche di tipo diverso, per ogni disco. Questi file system si montano su una architettura di file system logico per renderli disponibili all'uso.
- I file system spesso si realizzano secondo una struttura stratificata o modulare: i livelli più bassi hanno a che fare con le caratteristiche fisiche dei dispositivi di memorizzazione; i livelli più alti hanno a che fare con i nomi simbolici e le caratteristiche logiche dei file; i livelli intermedi fanno corrispondere le caratteristiche logiche dei file alle caratteristiche fisiche dei dispositivi.
- Lo spazio dei dischi può essere allocato ai file in tre modi: allocazione contigua, concatenata e indicizzata. L'allocatione contigua può risentire di frammentazione esterna. L'accesso diretto ai file è molto inefficiente con l'allocatione concatenata. L'allocatione indicizzata, infine, può richiedere un notevole overhead per il proprio blocco indice. Questi algoritmi si possono ottimizzare in molti modi. Lo spazio contiguo si può allargare attraverso delle estensioni allo scopo di aumentare la flessibilità e ridurre la frammentazione esterna. L'allocatione indicizzata si può realizzare in cluster per incrementare il throughput e ridurre il numero di elementi dell'indice necessari. L'indicizzazione in cluster di grandi dimensioni è simile all'allocatione contigua con estensioni.
- I metodi di allocatione dello spazio libero influenzano anche l'efficienza d'uso dello spazio dei dischi, le prestazioni del file system e l'affidabilità della memoria secondaria. I metodi usati comprendono i vettori di bit e le liste concatenate. Le ottimizzazioni comprendono il raggruppamento, il conteggio e la fat, che colloca la lista concatenata in una singola area contigua.
- Le procedure di gestione delle directory devono tener conto dell'efficienza, delle prestazioni e dell'affidabilità. La tabella hash è un metodo usato comunemente; è veloce ed efficiente. Sfortunatamente, il danneggiamento di una tabella o la caduta del sistema possono causare incoerenze tra le informazioni contenute nelle directory e il contenuto del disco.
- Un verificatore di coerenza può essere utilizzato per riparare la struttura danneggiata di un file system. Gli strumenti del sistema operativo per la creazione di copie di backup consentono la copiatura nelle unità a nastro dei dati contenuti nei dischi allo scopo di poterli ripristinare in seguito a perdite dovute a malfunzionamenti dei dispositivi fisici, errori del sistema operativo, o a errori degli utenti.
- A causa del ruolo fondamentale che i file system hanno nel funzionamento di un sistema, le loro prestazioni e affidabilità sono fondamentali. Tecniche come quelle che impiegano le cache e i file di log migliorano le prestazioni, mentre i log e le tecniche raid migliorano l'affidabilità. Il sistema wafl è un esempio di ottimizzazione delle prestazioni per rispondere a uno specifico carico di i/o.

Esercizi di ripasso

14.1 Prendete in considerazione un file costituito da 100 blocchi. Assumete che il blocco di controllo del file (e il blocco dell'indice, in caso di allocazione indicizzata) sia già in memoria. Calcolate quante operazioni di i/o del disco sono necessarie con le strategie di allocazione contigue, concatenate e indicizzate (singolo livello), se, per un blocco, valgono le condizioni che seguono. Nel caso di allocazione contigua, assumete che non ci sia spazio di crescita all'inizio, ma solo alla fine. Assumete inoltre che il blocco di informazione da aggiungere sia salvato in memoria.

- a. Il blocco viene aggiunto all'inizio.
- b. Il blocco viene aggiunto al centro.
- c. Il blocco viene aggiunto alla fine.
- d. Il blocco viene rimosso dall'inizio.
- e. Il blocco viene rimosso dal centro.
- f. Il blocco viene rimosso dalla fine.

14.2 Perché la mappa di bit per l'allocazione dei file deve essere conservata nella memoria di massa, e non nella memoria principale?

14.3 Considerate un sistema che supporta le strategie di allocazione contigua, concatenata e indicizzata. Quali criteri dovrebbero essere impiegati per decidere quale strategia è migliore per un dato file?

14.4 Un problema dell'allocazione contigua consiste nel fatto che l'utente deve preallocare abbastanza spazio per ogni file. Se il file diventa più grande dello spazio che gli è stato allocato, devono essere intraprese delle azioni specifiche. Questo problema può essere risolto definendo una struttura di file che consiste in un'area inizialmente contigua (di una dimensione specificata.) Se l'area viene riempita, il sistema operativo definisce automaticamente un'area di overflow linkata all'area contigua iniziale. Se l'area di overflow viene riempita, si alloca una seconda area di overflow. Paragonate questa implementazione con le versioni standard della allocazione contigua e concatenata.

14.5 Come possono le cache contribuire a migliorare le prestazioni? Perché i sistemi non utilizzano un maggior numero di cache, oppure cache più grandi, se esse sono così utili?

14.6 Perché è vantaggioso per l'utente che il sistema operativo allochi dinamicamente le proprie tabelle interne? Quali sono le conseguenze negative in cui incorrono i sistemi operativi che si comportano in questo modo?

Esercizi

14.7 Considerate un file system che adopera un metodo di allocazione contigua modificato, comprensivo di estensioni. Un file contiene diverse estensioni, ognuna delle quali corrisponde a un insieme contiguo di blocchi. Un aspetto cruciale, per tali sistemi, è il grado di variabilità nella dimensione delle estensioni. Quali sono i vantaggi e gli svantaggi nel caso che:

- a. tutte le estensioni siano della stessa grandezza, che è predeterminata;
- b. le estensioni possono essere di qualunque misura e sono allocate dinamicamente;
- c. le estensioni variano tra poche misure fisse, che sono predeterminate.

14.8 Confrontate le prestazioni delle tre tecniche per l'allocazione dei blocchi di un disco (allocazione contigua, concatenata e indicizzata), sia per file ad accesso sequenziale sia per file ad accesso casuale.

14.9 Quali vantaggi offre la variante dell'allocazione concatenata che utilizza una fat per collegare i blocchi di un file?

14.10 Considerate un sistema che tenga traccia dello spazio libero in una lista apposita.

- a. Supponete di perdere il puntatore alla lista. Il sistema è in grado di ricostruire la lista dello spazio libero? Argomentate la vostra risposta.
- b. Considerate un file system simile a quello con allocazione indicizzata impiegato da unix. Quante operazioni di i/o del disco potrebbero essere necessarie per leggere i contenuti di un piccolo file locale posto in /a/b/c? Si assume che nessuno dei blocchi del disco sia stato memorizzato nella cache.
- c. Proponete una soluzione che impedisca la perdita del puntatore dovuta a un guasto della memoria.

14.11 In alcuni file system è possibile allocare lo spazio sul disco con diverse granularità. Un file system, per esempio, può allocare 4 kb di spazio sul disco scegliendo un blocco unico da 4 kb oppure otto blocchi da 512 byte. In quale modo si può trarre vantaggio da questa flessibilità per migliorare le prestazioni? Quali modifiche si dovrebbero introdurre nel modello di gestione dello spazio libero per includervi questa caratteristica?

14.12 Discutete di come i meccanismi di ottimizzazione delle prestazioni dei file system potrebbero generare difficoltà nel mantenimento della coerenza dei sistemi, nel caso di un crash di sistema.

14.13 Discutete i vantaggi e gli svantaggi di supportare link a file che attraversano i punti di montaggio (ossia di link che si riferiscono a file memorizzati in volumi differenti).

14.14 Considerate un file system in un disco con dimensioni dei blocchi logici e fisici di 512 byte. Supponete che le informazioni su ciascun file siano già in memoria. Per ciascuno dei tre metodi di allocazione (contigua, concatenata e indicizzata) fornite le risposte alle seguenti domande:

- a. dite come in questo sistema si fanno corrispondere gli indirizzi logici agli indirizzi fisici (per l'allocazione indicizzata supponete che la lunghezza di un file sia sempre inferiore a 512 blocchi);
- b. se l'ultimo accesso è stato fatto al blocco 10 (posizione corrente), dite quanti blocchi fisici si devono leggere dal disco per accedere al blocco logico 4.

14.15 Considerate un file system che utilizza degli inode per rappresentare i file. I blocchi del disco hanno una dimensione di 8 kb e un puntatore a un blocco del disco richiede 4 byte. Questo file system ha 12 blocchi diretti sul disco, ma anche blocchi indiretti singoli, doppi e tripli. Qual è la dimensione massima di file che può essere memorizzata nel sistema?

14.16 In un dispositivo di memoria si può eliminare la frammentazione ricompattando le informazioni. I tipici dispositivi a disco non dispongono di registri di rilocazione o registri di base (come quelli usati per compattare la memoria); in questa situazione dite come si possono rilocare i file. Fornite tre motivi per i quali la ricompattazione e la rilocazione dei file vengono spesso evitate.

14.17 Spiegate perché l'annotazione degli aggiornamenti dei metadati assicura il ripristino di un file system dopo una caduta del sistema.

14.18 Considerate il seguente schema di creazione di copie di riserva.

- Giorno 1. Copiatura nel supporto di backup delle copie di riserva di tutti i file contenuti nel disco.
- Giorno 2. Copiatura in un altro supporto di tutti i file modificati dal giorno 1.
- Giorno 3. Copiatura in un altro supporto di tutti i file modificati dal giorno 1.

Tale schema differisce dalla sequenza data nel Paragrafo 14.7.4 per il fatto che tutte le copiature riguardano tutti i file modificati dopo la prima copiatura completa. Dite quali sono i vantaggi e gli svantaggi di questo sistema rispetto a quello del Paragrafo 14.7.4 e se le operazioni di recupero sono più semplici o più difficili. Motivate le vostre risposte.

CAPITOLO 15

Dettagli interni del file system

Come illustrato nel Capitolo 13, il file system fornisce il meccanismo per la memorizzazione e l'accesso al contenuto dei file, compresi dati e programmi. Questo capitolo riguarda principalmente le strutture e le operazioni interne del file system. Si esaminano varie modalità d'uso dei file, l'allocazione dello spazio dei dischi, il recupero dello spazio liberato, la registrazione delle locazioni dei dati, e l'interfaccia di altri componenti del sistema operativo alla memoria secondaria.

15.1 I file system

Nessun computer general-purpose memorizza un unico file: ci sono di solito migliaia, milioni, e persino miliardi di file archiviati all'interno di un computer. I file sono memorizzati su dispositivi di archiviazione ad accesso casuale, per esempio dischi rigidi, dischi ottici e dispositivi nvme.

Come discusso nei precedenti capitoli, un sistema informatico a uso generale dispone di dispositivi di memorizzazione multipli, che possono essere suddivisi in partizioni, le quali contengono i volumi che, a loro volta, contengono i file system. A seconda del gestore di volumi, un volume può estendersi su più partizioni. La Figura 15.1 mostra una tipica organizzazione del file system.

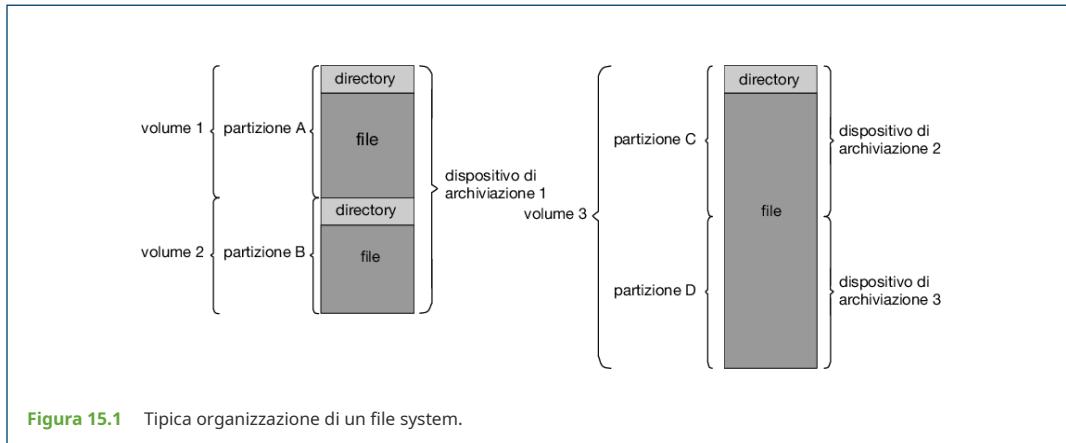
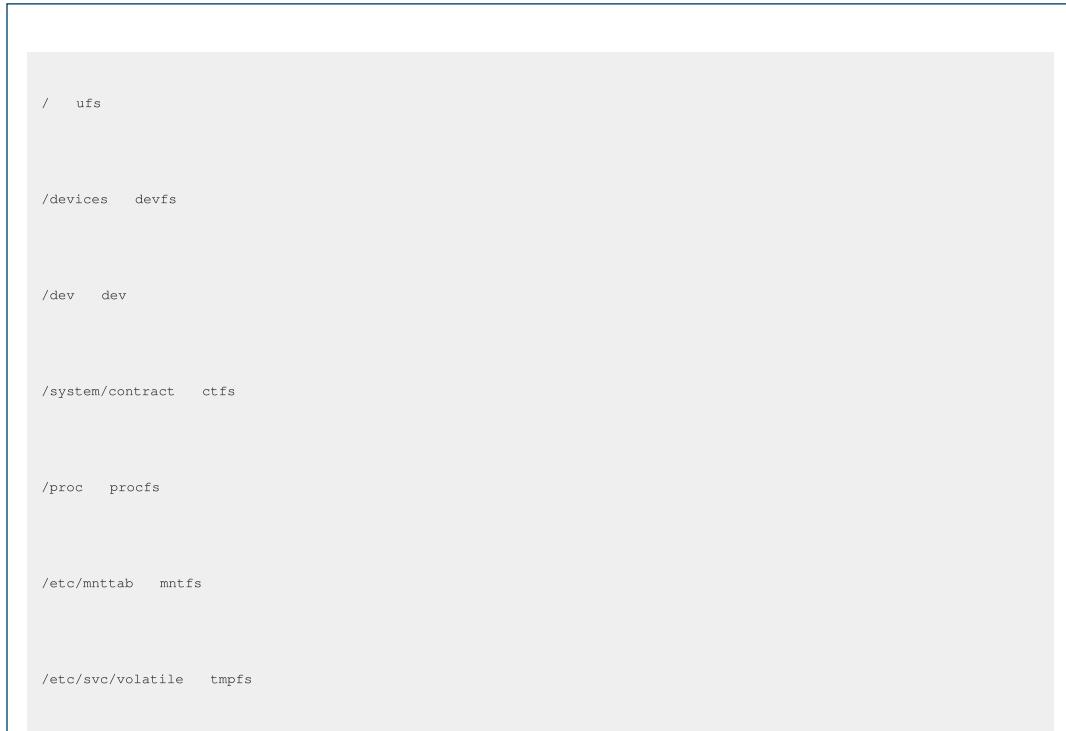


Figura 15.1 Tipica organizzazione di un file system.

Il numero di file system in un sistema informatico può variare da zero a molti, e possono essere di tipo diverso. Per esempio, un tipico sistema Solaris può avere molti file system di una dozzina di tipi diversi, come evidenziato nella Figura 15.2.



```
/system/object    objfs
```

```
/lib/libc.so.1    lofs
```

```
/dev/fd    fdfs
```

```
/var    ufs
```

```
/tmp    tmpfs
```

```
/var/run    tmpfs
```

```
/opt    ufs
```

```
/zpbge    zfs
```

```
/zpbge/backup    zfs
```

```
/export/home    zfs
```

```
/var/mail    zfs
```

```
/var/spool/mqueue    zfs
```

```
/zpbg    zfs
```

```
/zpbg/zones    zfs
```

Figura 15.2 File system in Solaris.

In questo libro prenderemo in considerazione solo file system di utilizzo generico. Vale la pena però di notare che esistono molti file system per scopi specifici. Consideriamo i tipi di file system dell'esempio menzionato precedentemente riguardante Solaris:

- tmpfs – un file system “temporaneo” creato nella memoria centrale volatile e i cui contenuti vengono cancellati se il sistema si riavvia o si blocca;
- objfs – un file system “virtuale” (essenzialmente un’interfaccia con il kernel che è simile a un file system) che permette agli strumenti che eseguono il debug di accedere ai simboli del kernel;

- cefs – un file system virtuale che mantiene le informazioni “contrattuali” per gestire quali sono i processi che partono quando il sistema si avvia e quali devono continuare a essere eseguiti durante il funzionamento del sistema;
- lofs – un file system di tipo “loop back” che permette di accedere a un file system al posto di un altro;
- procfs – un file system virtuale che presenta le informazioni su tutti i processi presenti come se esse risiedessero su un file system;
- ufs, zfs – file system a uso generale.

I file system dei computer possono quindi essere numerosi. Persino all'interno di un file system è utile dividere i file in gruppi da gestire e sui quali agire collettivamente. Un'organizzazione di questo tipo richiede l'utilizzo di directory (si veda il Paragrafo 14.3).

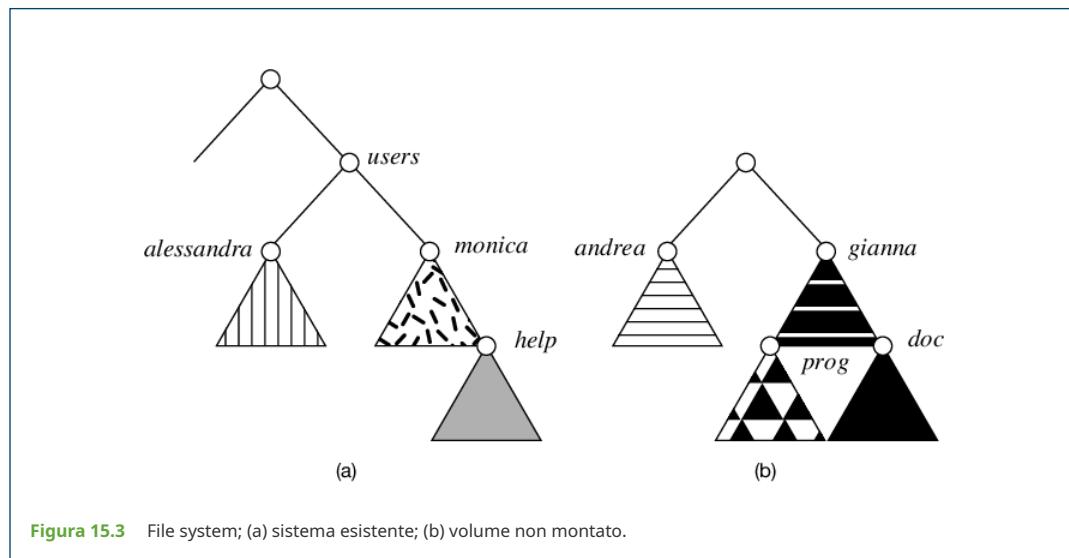
15.2 Montaggio di un file system

Così come si deve *aprire* un file per poterlo usare, per essere reso accessibile ai processi di un sistema, un file system deve essere *montato*. In particolare deve essere costruita la struttura della directory, composta di volumi, che devono essere montati affinché siano disponibili nello spazio dei nomi del file system.

La procedura di montaggio è molto semplice: si fornisce al sistema operativo il nome del dispositivo e la sua locazione (detta punto di montaggio) nella struttura di file e directory alla quale agganciare il file system. Alcuni sistemi operativi richiedono che venga specificato il tipo di file system, mentre altri ispezionano le strutture del dispositivo e determinano il tipo del file system presente. Di solito, un punto di montaggio è una directory vuota. Per esempio, in un sistema unix, un file system contenente le directory iniziali degli utenti si potrebbe montare come `/home`; quindi per avere accesso alla struttura della directory all'interno di quel file system, si premette `/home` ai nomi della directory (per esempio `/home/jane`). Se lo stesso file system si montasse come `/users` il percorso per quella stessa directory sarebbe `/users/jane`.

Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system contenuto nel dispositivo. La verifica si compie chiedendo al driver del dispositivo di leggere la directory di dispositivo e controllando che tale directory abbia il formato previsto. Infine, il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato. Questo schema permette al sistema operativo di attraversare la sua struttura della directory, passando da un file system all'altro, anche di tipi diversi, secondo le necessità.

Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella Figura 15.3, in cui i triangoli rappresentano i sottoalberi di directory di interesse. La Figura 15.3(a), mostra un file system esistente, mentre nella Figura 15.3(b) è raffigurato un volume non ancora montato che risiede in `/device/dsk`. A questo punto, si può accedere solo ai file del file system esistente. Nella Figura 15.4 si possono vedere gli effetti dell'operazione di montaggio del volume residente in `/device/dsk` al punto di montaggio `/users`. Se si smonta il volume, il file system ritorna alla situazione rappresentata nella Figura 15.3.



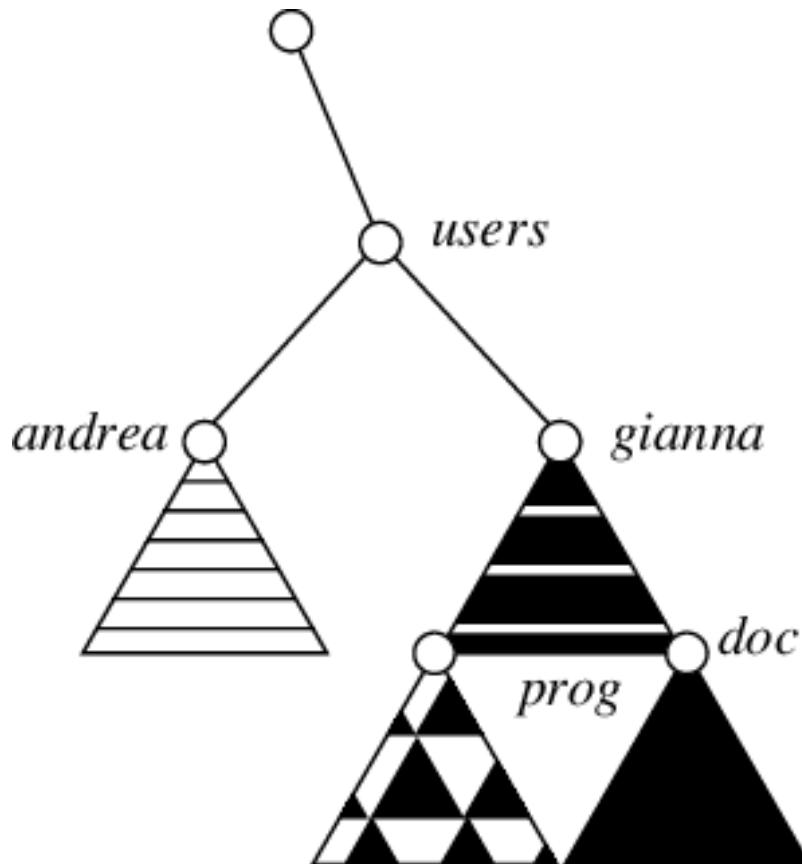


Figura 15.4 Volume montato sulla directory /users.

Per precisare le funzionalità, i sistemi operativi impongono regole semantiche a queste operazioni. Per esempio, un sistema potrebbe vietare il montaggio in una directory contenente file, o rendere disponibile il file system montato in tale directory e nascondere i file preesistenti nella directory finché non si *smonti* il file system, concludendone l'uso e permettendo l'accesso ai file originariamente presenti in tale directory. Come ulteriore esempio, un sistema potrebbe permettere il montaggio ripetuto dello stesso file system in diversi punti di montaggio, o potrebbe imporre una sola possibilità di montaggio per ciascun file system.

Si considerino le azioni del sistema operativo macos. Ogni volta che il sistema rileva per la prima volta un disco (sia nella fase d'avviamento che mentre il sistema è in esecuzione), il sistema operativo macos cerca un file system nel dispositivo. Se ne trova uno, esegue automaticamente il montaggio del file system nella directory `/volumes`, aggiungendo un'icona di cartella sul desktop, etichettata con il nome del file system (secondo quel che è memorizzato nella directory di dispositivo). A questo punto l'utente può selezionare l'icona con il mouse e quindi vedere il contenuto del nuovo file system appena montato.

La famiglia di sistemi operativi Microsoft Windows mantiene una struttura della directory a due livelli estesa, con una lettera di unità associata a dispositivi e volumi. I volumi hanno una struttura della directory a grafo generale associata a una lettera di unità. Il percorso completo per uno specifico file è quindi della forma `lettera_di_unità:\percorso\file`. Le versioni più recenti di Windows permettono di montare un file system su qualunque punto dell'albero delle directory, esattamente come unix. I sistemi Windows rilevano automaticamente tutti i dispositivi e montano tutti i file system rilevati all'avvio della macchina. In altri sistemi, per esempio unix, i comandi di montaggio sono esplicativi. Un file di configurazione del sistema contiene una lista di dispositivi (e relativi punti di montaggio) da montare automaticamente all'avvio, ma altri montaggi possono essere eseguiti manualmente.

Le questioni riguardanti il montaggio di file system sono approfondite nei Paragrafi 15.3 e C.7.5 (Appendice C reperibile sulla piattaforma MyLab).

15.3 Partizioni e montaggio

Un disco si può strutturare in vari modi, a seconda del sistema operativo che lo gestisce. Si può suddividere in più partizioni, oppure un volume può comprendere più partizioni su molteplici dischi. Qui trattiamo il primo caso, mentre il secondo, che si può considerare più correttamente un caso particolare di organizzazione raid, è trattato nel Paragrafo 11.8.

Ciascuna partizione può essere priva di struttura logica (*raw partition*), oppure può contenere un file system. Si usa un disco privo di struttura logica (*raw disk*) se nessun file system è appropriato all'utilizzo che se ne vuole fare. Il sistema operativo unix impiega una partizione priva di struttura per l'area d'avvicendamento dei processi; per questo scopo usa un formato specifico. Allo stesso modo alcuni sistemi di gestione di basi di dati usano dischi non strutturati e formattano i dati secondo le proprie necessità. Un disco privo di struttura logica può anche contenere informazioni necessarie per sistemi raid di gestione dei dischi, per esempio le mappe di bit che indicano quali blocchi sono duplicati in altri dischi, e in quali sono state effettuate modifiche che si devono duplicare. Analogamente, può contenere una piccola base di dati di informazioni sulla configurazione raid, per esempio, quali dischi appartengono a ciascun insieme raid. Il Paragrafo 11.5.1 affronta altri aspetti concernenti l'uso dei raw disk.

Se una partizione contiene un file system di avvio, con sistema operativo installato e configurato correttamente, allora deve contenere anche le informazioni relative all'avviamento del sistema, come descritto nel Paragrafo 11.5.2. Queste informazioni devono avere un proprio formato, poiché nella fase d'avviamento il sistema non ha ancora caricato il codice del file system e quindi non può interpretarne il formato. Le informazioni consistono in una serie sequenziale di blocchi, che si carica in memoria come un'immagine. L'esecuzione dell'immagine comincia a una locazione prefissata, per esempio il primo byte. Questo boot loader ha invece abbastanza informazioni sul file system da essere in grado di caricare il kernel e avvarne l'esecuzione.

Esso può contenere più informazioni di quelle che servono per un singolo sistema operativo. Su molti sistemi, per esempio, è possibile l'installazione di più sistemi operativi. Come può il sistema sapere quale sistema operativo deve caricare? In questo caso l'area d'avviamento contiene un boot loader, capace di interpretare diversi file system e diversi sistemi operativi. Una volta caricato, può avviare uno dei sistemi operativi disponibili nei dispositivi di memorizzazione. I dispositivi possono avere più partizioni, ognuna contenente un diverso tipo di file system e un sistema operativo differente. Si noti che se il boot loader non fosse in grado di capire un particolare formato di file system, il sistema operativo memorizzato su quel file system non sarebbe avviabile. Anche per questa ragione solo alcuni file system sono supportati come file system di root per un dato sistema operativo.

Nella fase di caricamento del sistema operativo, si esegue il montaggio della partizione radice (*root partition*), che contiene il kernel del sistema operativo e in alcuni casi altri file di sistema. A seconda del sistema operativo, il montaggio di altri volumi avviene automaticamente in questa fase oppure si può compiere successivamente in maniera manuale. Durante l'operazione di montaggio, il sistema verifica che il dispositivo contenga un file system valido chiedendo al dispositivo di leggere la directory di dispositivo e verificando che la directory abbia il formato corretto. Se così non fosse, è necessaria una verifica della coerenza della partizione e una eventuale correzione, con o senza l'intervento dell'utente. Infine, il sistema annota nella tabella di montaggio in memoria che un file system è stato montato e il tipo di file system. I dettagli di questa funzione dipendono dal sistema operativo.

I sistemi Microsoft Windows eseguono il montaggio di ogni volume in uno spazio di nomi separato, identificato da una lettera seguita dai due punti (:). Per esempio, per memorizzare che un file system è stato montato in F:, il sistema operativo introduce un puntatore al file system in un campo della struttura del dispositivo corrispondente a F:. Quando un processo specifica la lettera dell'unità, il sistema operativo trova il puntatore al file system appropriato e attraversa le strutture della directory in quel dispositivo per trovare lo specifico file o directory. Le più recenti versioni di Windows permettono il montaggio di un file system in qualsiasi punto all'interno della struttura della directory esistente.

In unix il montaggio di un file system si può compiere in qualsiasi directory. Questa funzione si realizza impostando un flag nella copia in memoria dell'*inode* di quella directory, segnalando che la directory è un punto di montaggio. Un campo dell'*inode* punta a un elemento nella tabella di montaggio, che indica quale dispositivo è montato in quella posizione. L'elemento della tabella di montaggio contiene un puntatore al superblocco del file system in quel dispositivo. Questo schema permette al sistema operativo di attraversare facilmente la propria struttura della directory, attraversando in maniera trasparente file system di tipo diverso.

15.4 Condivisione di file

La capacità di condividere i file è particolarmente utile per utenti che vogliono collaborare per ridurre le risorse richieste per raggiungere un certo obiettivo. Quindi, nonostante le difficoltà inerenti alla condivisione, i sistemi operativi orientati agli utenti devono soddisfare questa esigenza.

In questo paragrafo si considerano diversi aspetti della condivisione dei file, partendo dalle problematiche generali che nascono quando più utenti condividono dei file. Una volta che più utenti possono condividere file, l'obiettivo diventa estendere la condivisione a più file system, compresi i file system remoti. Infine, ci possono essere diverse interpretazioni delle azioni conflittuali intraprese su file condivisi. Per esempio, se più utenti stanno scrivendo nello stesso file, ci si può chiedere se il sistema dovrebbe permettere tutte le operazioni di scrittura, oppure dovrebbe proteggere le azioni di ciascun utente da quelle degli altri.

15.4.1 Utenti multipli

Se un sistema operativo permette l'uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e alla loro protezione. Data una struttura della directory che permette la condivisione di file da parte degli utenti, il sistema deve intermediare questa condivisione. Il sistema può permettere a ogni utente di accedere ai file degli altri utenti per default, oppure richiedere che un utente debba esplicitamente concedere i permessi di accesso ai file. Questi aspetti sono alla base dei temi del controllo degli accessi e della protezione, trattati nel Paragrafo 13.4.

Per realizzare i meccanismi di condivisione e protezione, il sistema deve memorizzare e gestire più attributi per le directory e i file rispetto a un sistema che consente un singolo utente. Sebbene storicamente siano stati proposti molti metodi per la realizzazione di questi meccanismi, la maggior parte dei sistemi è arrivata ad adottare – per ciascun file o directory – i concetti di *proprietario* (owner) o *utente e gruppo*. Il proprietario è l'utente che può cambiare gli attributi, concedere l'accesso e che, in generale, ha il maggior controllo sul file. L'attributo di gruppo di un file si usa per definire il sottoinsieme di utenti autorizzati a condividere il file. Per esempio, il proprietario di un file in un sistema unix può fare qualsiasi operazione sul file, mentre i membri del gruppo possono compiere un sottoinsieme di queste operazioni e il resto degli utenti un altro sottoinsieme. Il proprietario del file può definire l'esatto insieme di operazioni che i membri del gruppo e gli altri utenti possono eseguire. Maggiori dettagli sugli attributi che regolano i permessi sono trattati nel paragrafo successivo.

Gli identificatori (id) del gruppo e del proprietario di un certo file o directory sono memorizzati insieme con gli altri attributi del file. Quando un utente richiede di compiere un'operazione su un file, per verificare se l'utente richiedente è il proprietario del file si può confrontare l'id utente con l'attributo che identifica il proprietario. Analogamente si confrontano gli id di gruppo. Ne risultano i permessi applicabili, che il sistema considera per consentire o impedire l'operazione richiesta.

Molti sistemi operativi hanno più file system locali, inclusi volumi di un unico disco, o più volumi in diversi dischi connessi al sistema. In questi casi, la verifica degli id e il confronto dei permessi si possono fare facilmente dopo l'operazione di montaggio dei file system. Si consideri ora un disco esterno che può essere spostato tra vari sistemi: che cosa succede se gli id sui sistemi sono diversi? È necessario prestare attenzione per garantire che ci sia corrispondenza tra gli id in seguito allo spostamento dei dispositivi tra i sistemi, oppure che la proprietà dei file venga resettata quando si verifica un tale spostamento (per esempio, si può creare un nuovo id utente a cui assegnare tutti i file sul disco esterno, in modo da essere sicuri che nessun file sia accidentalmente accessibile agli utenti esistenti).

15.5 File system virtuali

Come abbiamo visto, i sistemi operativi moderni devono gestire contemporaneamente tipi di file system diversi. Per capire come si può realizzare questa funzione occorre tuttavia considerare il modo in cui un sistema operativo può consentire l'integrazione di diversi tipi di file system in un'unica struttura della directory, così da permettere agli utenti di spostarsi senza problemi da un tipo di file system all'altro, mentre percorrono tutta la struttura.

Un metodo ovvio ma non ottimale per realizzare più tipi di file system è scrivere procedure di gestione di file e directory differenti per ciascun tipo di file system. Al contrario, la maggior parte dei sistemi operativi, compreso unix, impiega tecniche orientate agli oggetti per semplificare, organizzare e rendere modulare la soluzione. L'uso di queste tecniche rende possibile l'implementazione, nella stessa struttura, di tipi di file system molto diversi tra loro, compresi i file system di rete, come l' nfs. Gli utenti possono accedere a file contenuti in diversi file system nei dischi locali, o anche in file system disponibili tramite la rete.

Per isolare le funzioni di base delle chiamate di sistema dai dettagli di implementazione si adoperano apposite strutture dati e procedure. In questo modo la realizzazione del file system si articola in tre strati principali, riportati in modo schematico nella Figura 15.5. Il primo strato è l'interfaccia del file system, basata sulle chiamate di sistema `open()`, `read()`, `write()` e `close()` e sui descrittori di file.

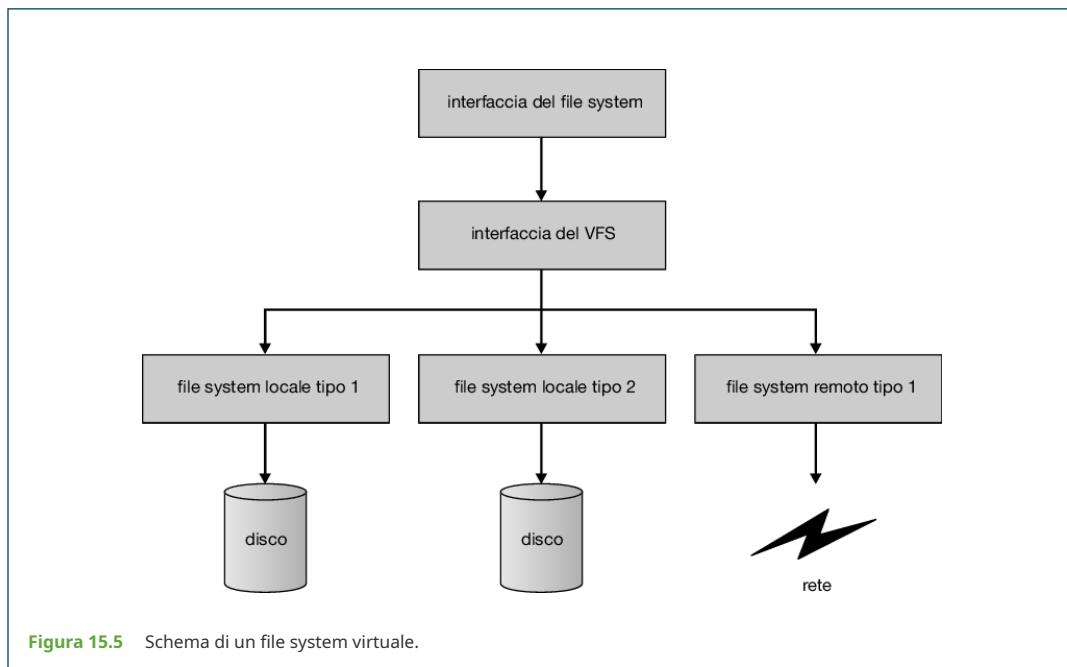


Figura 15.5 Schema di un file system virtuale.

Il secondo strato si chiama strato del file system virtuale (*virtual file system*, vfs) e svolge due funzioni importanti.

1. Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia vfs uniforme. Nello stesso calcolatore possono coesistere più implementazioni dell'interfaccia vfs, che permettono un accesso trasparente a diversi tipi di file system montati localmente.
2. Permette la rappresentazione univoca di un file su tutta una rete. Il vfs è basato su una struttura di rappresentazione dei file detta vnode che contiene un indicatore numerico unico per tutta la rete per ciascun file. (Gli *inode* di unix sono unici solo all'interno di un singolo file system.) Tale unicità per tutta la rete è richiesta per la gestione dei file system di rete. Il kernel contiene una struttura *vnode* per ciascun nodo attivo, sia che si tratti di un file sia che si tratti di una directory.

Quindi, il vfs distingue i file locali da quelli remoti, e distingue i file locali a seconda del relativo tipo di file system.

Il vfs, a seconda del tipo di file system, attiva le operazioni specifiche di un file system per gestire le richieste locali, e invoca le procedure del protocollo nfs per le richieste remote. Gli handle del file si costruiscono a partire dai *vnode* relativi e s'inviano a queste procedure come argomenti. Lo strato che realizza il codice di uno specifico file system o il protocollo di file system remoto è il terzo dell'architettura.

Esaminiamo succintamente l'architettura vfs di Linux. I quattro tipi più importanti di oggetti definiti nel vfs di Linux sono:

- l'oggetto *inode*, che rappresenta il singolo file;
- l'oggetto *file*, che rappresenta un file aperto;
- l'oggetto *superblock*, che rappresenta un intero file system;

- l'oggetto dentry, che rappresenta il singolo elemento della directory.

Per ognuno di questi tipi, vfs specifica un insieme di operazioni da implementare. Ciascun oggetto di uno di questi tipi contiene un puntatore a una tabella di funzioni; questa, alla sua volta, contiene gli indirizzi delle effettive funzioni che implementano le operazioni definite per quel particolare oggetto. Per esempio, una versione abbreviata della api di alcune delle operazioni dell'oggetto file comprende:

- `int open(...)` – apre il file.
- `int close(...)` – chiude un file aperto.
- `ssize_t read(...)` – legge dal file.
- `ssize_t write(...)` – scrive sul file.
- `int mmap(...)` – mappa il file in memoria.

Ogni implementazione dell'oggetto file per uno specifico tipo di file deve implementare tutte le funzioni specificate nella definizione dell'oggetto file. (La definizione completa dell'oggetto file si trova nella struttura `struct file_operations`, nel file `/usr/include/linux/fs.h`).

Questo schema permette allo strato software vfs di eseguire operazioni su uno degli oggetti in questione invocando l'appropriata funzione della tabella delle funzioni dell'oggetto, senza dover conoscere i dettagli dello specifico oggetto. Per esempio, vfs non sa, né vuol sapere, se un certo inode rappresenti un file su disco, un file di directory o un file remoto. L'implementazione appropriata dell'operazione `read()` per l'oggetto in questione è comunque reperibile sempre nel medesimo punto all'interno della tabella delle funzioni: invocando tale implementazione, vfs può disinteressarsi dei dettagli legati a come vengono effettivamente letti i dati.

15.6 File system remoti

L'avvento delle reti (Capitolo 19) ha permesso la comunicazione tra calcolatori remoti. Le reti permettono la condivisione di risorse sparse nell'area di un campus o addirittura in diversi luoghi del mondo. Un'ovvia risorsa da condividere sono i dati, nella forma di file.

I metodi con i quali i file si condividono in una rete sono cambiati molto, seguendo l'evoluzione della tecnologia delle reti e dei file. Il primo metodo utilizzato consiste nel trasferimento dei file richiesto in modo esplicito dagli utenti, attraverso programmi come l'`ftp`. Un secondo metodo è quello del file system distribuito (*distributed file system*, dfs), che permette la visibilità nel calcolatore locale delle directory remote. Il terzo metodo, il World Wide Web, è, da un certo punto di vista, un ritorno al primo. Per accedere ai file remoti si usa un browser, e operazioni distinte – essenzialmente un involucro (*wrapper*) per l'`ftp` – per trasferirli. Un altro metodo, sempre più diffuso, per la condivisione di file è il cloud computing (Paragrafo 1.10.5).

L'`ftp` si usa sia per l'accesso anonimo sia per quello autenticato. L'accesso anonimo permette di trasferire file senza avere un account nel sistema remoto. Il World Wide Web usa quasi esclusivamente lo scambio di file anonimo. Un dfs comporta un'integrazione molto più stretta tra il calcolatore che accede ai file remoti e il calcolatore che fornisce i file. Tale integrazione incrementa la complessità, com'è illustrato nel prossimo paragrafo.

15.6.1 Modello client-server

I file system remoti permettono il montaggio di uno o più file system di uno o più calcolatori remoti in un calcolatore locale. In questo caso, il calcolatore contenente i file si chiama server, mentre il calcolatore che richiede l'accesso ai file si chiama client. La relazione tra client e server è piuttosto comune tra i calcolatori di una rete. In generale, il server dichiara che determinate risorse sono disponibili ai client, specificando esattamente quali (in questo caso, quali file) ed esattamente a quali client. Un server può gestire richieste provenienti da più client, e un client può accedere a più server, secondo l'implementazione del particolare sistema client-server.

Il server in genere specifica i file disponibili su di un volume o livello di directory. L'identificazione dei client è più difficile; un client può essere identificato tramite i relativi nomi simbolici di rete, o tramite altri identificatori come un indirizzo ip, ma questi possono essere facilmente imitati (*spoofing*). Di conseguenza un client non autorizzato può accedere a un server. Tra le soluzioni più sicure ci sono quelle che prevedono l'autenticazione del client tramite chiavi di cifratura. Sfortunatamente, l'introduzione di tecniche per la sicurezza introduce nuovi problemi, per esempio la necessità della compatibilità tra client e server (si devono impiegare gli stessi algoritmi di cifratura) e dello scambio sicuro delle chiavi (l'intercettazione delle chiavi può permettere accessi non autorizzati). Questi problemi sono sufficientemente difficili da far sì che nella maggioranza dei casi si usino metodi di autenticazione insicuri.

Nel caso di unix e del suo file system di rete (*network file system*, nfs), l'autenticazione avviene, per default, tramite le informazioni di rete relative al client. In questo schema, gli identificatori (id) dell'utente devono coincidere nel client e nel server; diversamente, il server non può determinare i diritti d'accesso ai file. Si consideri per esempio un utente con un identificatore uguale a 1000 nel client e a 2000 nel server. Una richiesta per uno specifico file dal client al server non potrà essere gestita correttamente, perché il server cercherà di determinare se l'utente 1000 ha i permessi d'accesso al file, invece di usare il reale id dell'utente che è 2000. L'accesso sarà concesso o negato secondo un'informazione di autenticazione sbagliata. Il server deve fidarsi del client e assumere che quest'ultimo gli presenti l'identificatore corretto. Si noti che il protocollo nfs permette relazioni da molti a molti; cioè più server possono fornire file a più client. Infatti, un calcolatore può comportarsi sia da server per altri client nfs, sia da client di altri server nfs.

Una volta montato il file system remoto, le richieste delle operazioni su file sono inviate al server, attraverso la rete, per conto dell'utente, usando il protocollo dfs. Normalmente, una richiesta di apertura di file si invia insieme con l'id dell'utente richiedente. Il server quindi applica i normali controlli d'accesso per determinare se l'utente ha le credenziali per accedere al file nel modo richiesto; se tali controlli hanno esito positivo, riporta un handle d'accesso al file all'applicazione client, che la usa per eseguire sul file operazioni di lettura, scrittura e altro. Il client chiude il file quando non deve più accedervi. Il sistema operativo può applicare una semantica simile a quella adottata per il montaggio di un file system locale, oppure una semantica diversa.

15.6.2 Sistemi informativi distribuiti

Per semplificare la gestione dei servizi client-server, i sistemi di informazione distribuiti, noti anche come servizi di naming distribuiti, sono stati concepiti per fornire un accesso unificato alle informazioni necessarie per il calcolo remoto. Il sistema dei nomi di dominio (*domain name system*, dns) fornisce le traduzioni dai nomi dei calcolatori agli indirizzi di rete per l'intera Internet. Prima che il dns si diffondesse capillarmente nella rete, si scambiavano tra i calcolatori, per posta elettronica o `ftp`, file contenenti le stesse informazioni. Questo metodo, ovviamente, non poteva adattarsi dinamicamente all'aumento delle dimensioni della rete Internet. Il dns è trattato ulteriormente nel Paragrafo 19.3.1.

Altri sistemi di informazione distribuiti forniscono uno spazio identificato da *nome utente/parola d'ordine/identificatore utente/identificatore di gruppo* per un servizio distribuito. I sistemi unix hanno adottato un'ampia varietà di metodi per l'informazione distribuita. Sun Microsystems (che ora è parte della Oracle Corporation) ha introdotto il sistema *yellow pages* (poi ribattezzato *network information service*, nis), adottato da gran parte dell'industria. Questo servizio centralizza la memorizzazione dei nomi degli utenti e dei calcolatori, delle informazioni sulle stampanti e altro. Sfortunatamente, usa metodi di autenticazione insicuri, per esempio l'invio di parole d'ordine dell'utente non cifrate (*in chiaro*) e l'identificazione dei calcolatori attraverso gli indirizzi ip. Il sistema nis+ di Sun era una versione del nis più sicura, ma anche molto più complessa e non ha avuto una gran diffusione.

Nel caso del *common internet file system* (cifs) di Microsoft, le informazioni di rete si usano insieme con gli elementi di autenticazione dell'utente (nome dell'utente e parola d'ordine) per creare un login di rete che il server usa per decidere se permettere o negare l'accesso a un file system richiesto. Affinché questa autenticazione sia valida, i nomi utente devono essere uguali nelle varie macchine (come per l' nfs). Per fornire un unico spazio di nomi per gli utenti, Microsoft usa l'active directory. Una volta impostata, la

funzione di naming è usata per autenticare gli utenti da tutti i client e da tutti i server mediante la versione di Microsoft del protocollo di autenticazione di rete Kerberos (<https://web.mit.edu/kerberos>).

L'industria si sta orientando verso il protocollo ldap (*lightweight directory-access protocol*) come meccanismo sicuro per il naming distribuito. Lo stesso *active directory* è basato sull'ldap. Oracle Solaris e altri importanti sistemi operativi includono ldap e permettono l'uso di questo protocollo per l'autenticazione degli utenti e per altri servizi di ricerca di informazioni a livello dell'intero sistema, per esempio la disponibilità delle stampanti. È pensabile che un'organizzazione possa usare una singola directory ldap distribuita per memorizzare le informazioni su tutti gli utenti e le risorse di tutti i calcolatori dell'organizzazione stessa. Si avrebbe un unico punto di autenticazione sicura (*single sign-on*) per gli utenti, che inserirebbero una sola volta le proprie informazioni di autenticazione per avere accesso a tutti i calcolatori dell'organizzazione. Questa soluzione semplificherebbe anche i compiti degli amministratori di sistema, concentrando in un unico punto informazioni ora sparse in vari file in ciascun sistema o in diversi servizi di informazione distribuiti.

15.6.3 Tipi di malfunzionamento

I file system locali possono presentare malfunzionamenti per varie cause: problemi dei dischi che li contengono, alterazione dei dati relativi alle strutture delle directory o a informazioni necessarie alla gestione dei dischi (chiamate collettivamente metadati), malfunzionamenti dei controlleri dei dischi, problemi ai cavi di connessione o agli adattatori. Anche un errore di un utente o dell'amministratore di sistema può causare la perdita di file, d'intere directory o addirittura la cancellazione di volumi. Molti di questi malfunzionamenti portano alla caduta del sistema (*crash*), all'emissione di una condizione d'errore e alla necessità di un intervento umano per risolvere il problema.

L'uso di file system remoti implica ulteriori tipi di malfunzionamento; a causa della complessità dei sistemi di rete e della necessità di interazioni tra calcolatori remoti, i problemi che possono interferire con il corretto funzionamento dei file system remoti sono infatti molto più numerosi. Nel caso delle reti, si possano verificare interruzioni del collegamento tra due calcolatori, dovute a guasti hardware o a improprie configurazioni dell'architettura, oppure a problemi di implementazione della rete. Sebbene alcune reti includano meccanismi di tolleranza ai guasti, compresi cammini multipli tra ogni coppia di calcolatori, altre non li prevedono. Qualsiasi malfunzionamento potrebbe interrompere il flusso dei comandi del dfs.

Si consideri un client mentre usa un file system remoto. Il client ha qualche file remoto aperto; tra le varie attività potrebbe percorrere le directory remote per aprire aprire file, svolgere operazioni di lettura e scrittura e chiudere i file. Si consideri ora un malfunzionamento della rete, una caduta del server remoto, oppure anche uno spegnimento programmato di quel server: improvvisamente il file system remoto è inaccessibile. Questo scenario è piuttosto comune, quindi il client non dovrebbe comportarsi come nel caso di una perdita del file system locale. Piuttosto, il sistema dovrebbe terminare tutte le operazioni sul server non più raggiungibile, oppure posticiparle finché il server sarà nuovamente disponibile. Questa semantica di trattamento dei malfunzionamenti si definisce e si realizza come parte del protocollo di file system remoto. La terminazione di tutte le operazioni può portare alla perdita di dati (e della pazienza) da parte degli utenti. Quindi la maggior parte dei protocolli dfs impone o permette la posticipazione delle operazioni sul file system remoto, con la speranza che il calcolatore remoto diventi nuovamente disponibile.

Per realizzare questo tipo di recupero dai malfunzionamenti è necessario mantenere alcune informazioni di stato sia sui client sia sui server. Se sia i client sia i server tengono traccia delle loro attività correnti e dei loro file aperti, entrambi possono riprendersi senza problemi da un malfunzionamento. Nel caso in cui il server "cada" ma debba tracciare il montaggio remoto di file system e i file aperti, l'nfs segue un criterio semplice realizzando un dfs senza stato. Sostanzialmente, assume che una richiesta di un client per la lettura o scrittura di un file non sarebbe avvenuta, a meno che il file system non sia stato montato in modo remoto e il file in questione aperto prima della richiesta. Il protocollo nfs trasporta tutte le informazioni necessarie per localizzare il file appropriato e per svolgere l'operazione richiesta sul file. Allo stesso modo, non tiene traccia di quali client abbiano montato i propri volumi esportati, assumendo anche in questo caso che, se perviene una richiesta, debba essere legittima. Sebbene questo metodo senza stato renda l'nfs tollerante ai guasti e piuttosto facile da realizzare, lo rende insicuro. Per esempio un server nfs potrebbe permettere richieste contraddette di lettura o scrittura. Questioni del genere sono regolamentate dallo standard industriale nfs versione 4, in cui nfs è dotato di stato per migliorare le proprie funzionalità, prestazioni e sicurezza.

15.7 Semantica della coerenza

La semantica della coerenza è un importante criterio per la valutazione di qualsiasi file system che consenta la condivisione dei file. Questa semantica specifica il modo in cui più utenti devono accedere contemporaneamente a un file condiviso. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere osservate da altri utenti. La semantica è tipicamente realizzata come parte del codice del file system.

La semantica della coerenza è direttamente correlata agli algoritmi di sincronizzazione dei processi del Capitolo 6. Tuttavia, a causa delle lunghe latenze e delle basse velocità di trasferimento dei dischi e delle reti, i complessi algoritmi descritti in tale capitolo di solito non s'impiegano per l'i/o su file. Per esempio, l'esecuzione di una transazione atomica su dischi remoti può coinvolgere molte comunicazioni di rete e molte letture e scritture nei dischi. I sistemi che tentano una così completa serie di funzioni tendono ad avere scarse prestazioni. Una realizzazione riuscita di una complessa semantica della condivisione si trova nel file system Andrew.

Nella trattazione seguente si suppone che una serie d'accessi, cioè letture e scritture, tentati da un utente allo stesso file sia sempre compresa tra una coppia di operazioni `open()` e `close()`. Tale serie d'accessi si chiama sessione di file. Per illustrare questo concetto si descrivono sommariamente qui di seguito alcuni importanti esempi di semantica della coerenza.

15.7.1 Semantica unix

Il file system di unix, descritto nel Capitolo 19, usa la seguente semantica della coerenza.

- Le scritture in un file aperto da parte di un utente sono immediatamente visibili ad altri utenti che hanno lo stesso file contemporaneamente aperto.
- Un metodo di condivisione permette agli utenti di condividere il puntatore alla locazione corrente nel file. Quindi l'avanzamento del puntatore da parte di un utente influenza su tutti gli utenti che condividono il file. In questo caso il file ha una singola immagine e tutti gli accessi si alternano (intercalandosi) a prescindere dalla loro origine.

Nella semantica unix un file è associato a una singola immagine fisica, accessibile come una risorsa esclusiva. La contesa su quest'immagine singola determina ritardi nei processi utente.

15.7.2 Semantica delle sessioni

Il file system Andrew (Openafs) usa la seguente semantica della coerenza:

- le scritture in un file aperto da un utente non sono visibili immediatamente ad altri utenti che hanno lo stesso file contemporaneamente aperto;
- una volta chiuso il file, le modifiche apportate sono visibili solo nelle sessioni che iniziano successivamente. Le istanze del file già aperte non riportano queste modifiche.

Secondo questa semantica, un file può essere contemporaneamente associato a parecchie immagini, probabilmente diverse. Di conseguenza, più utenti possono eseguire accessi concorrenti di lettura o scrittura sulla rispettiva immagine del file senza subire ritardi. Non si impone quasi alcun vincolo nella gestione degli accessi.

15.7.3 Semantica dei file condivisi immutabili

Un approccio particolare è quello dei file condivisi immutabili; una volta che un file è stato dichiarato *condiviso* dal suo creatore, non può essere modificato. Un file immutabile presenta due caratteristiche chiave: il suo nome non si può riutilizzare e il suo contenuto non può essere modificato. Quindi il nome di un file immutabile indica che i contenuti di quel file sono fissi. Come si descrive nel Capitolo 19, la realizzazione di questa semantica in un sistema distribuito è semplice; infatti la condivisione è molto disciplinata poiché consente la sola lettura.

15.8 NFS

I file system di rete sono ampiamente diffusi; in genere si integrano con l'intera struttura della directory e l'interfaccia del sistema client. L'nfs è un buon esempio di file system di rete, client-server, ampiamente usato e ben realizzato. In questo paragrafo lo utilizzeremo per esplorare i dettagli di implementazione dei file system di rete.

L'nfs è sia una realizzazione sia una specifica di un sistema software per l'accesso a file remoti attraverso lan, o anche wan. Fa parte dell'onc+, adottato dalla maggior parte dei fornitori di sistemi operativi unix e in alcuni sistemi operativi per pc. La versione qui descritta fa parte del sistema operativo Solaris, che è a sua volta una versione modificata dello unix svr4. Esso usa il protocollo tcp/ip o udp/ip (a seconda della rete di comunicazione). Nella nostra descrizione dell'nfs, specifica e implementazione si accavallano: per ogni descrizione dettagliata si fa riferimento alla versione realizzata per Solaris; mentre ogni descrizione generale vale anche per la sua specifica.

Ci sono numerose versioni di nfs, l'ultima delle quali è la Versione 4. Esamineremo qui la Versione 3, attualmente la più utilizzata.

15.8.1 Generalità

Nel contesto dell'nfs si considera un insieme di stazioni di lavoro interconnesse come un insieme di calcolatori indipendenti con file system indipendenti. Lo scopo è quello di permettere un certo grado di condivisione tra i file system, su richiesta esplicita, in modo trasparente. La condivisione è basata su una relazione client-server. Un calcolatore può essere, come spesso accade, sia un client sia un server. La condivisione è ammessa tra ogni coppia di calcolatori. Per assicurare l'indipendenza dei calcolatori, la condivisione di un file system remoto ha effetto esclusivamente sul calcolatore client.

Affinché una directory remota sia accessibile in modo trasparente da un calcolatore particolare, per esempio C_1 , un client di quel calcolatore deve prima eseguire un'operazione di montaggio. La semantica dell'operazione richiede di montare una directory remota in corrispondenza di una directory di un file system locale. Una volta completata l'operazione di montaggio, la directory montata assume l'aspetto di un sottoalbero integrato nel file system locale, e sostituisce il sottoalbero che discende dalla directory locale; questa, a sua volta, rappresenta la radice della directory appena montata. La directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito: si deve fornire la locazione (nome del calcolatore) della directory remota. Tuttavia, da questo momento in poi gli utenti del calcolatore C_1 possono accedere ai file della directory remota in modo del tutto trasparente.

Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella Figura 15.6, in cui i triangoli rappresentano i sottoalberi di directory di interesse. La figura illustra tre file system di calcolatori indipendenti chiamati U , $S1$ e $S2$. A questo punto, in ogni calcolatore si può accedere solo a file locali. Nella Figura 15.7 (a) mostra l'effetto del montaggio di $s1:/usr/shared$ in $U:/usr/local$. In questa figura è illustrata la visione che gli utenti di U hanno del loro file system. Occorre osservare che, una volta completato il montaggio, essi possono accedere a qualsiasi file che si trovi nella directory $dir1$, usando il prefisso $/usr/local/dir1$. La directory originale $/usr/local$ di quel calcolatore non è più visibile.

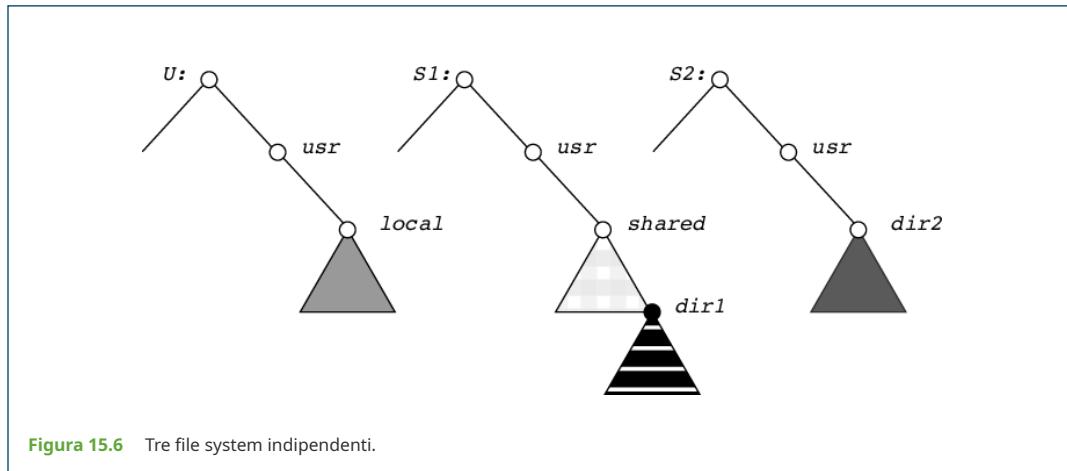


Figura 15.6 Tre file system indipendenti.

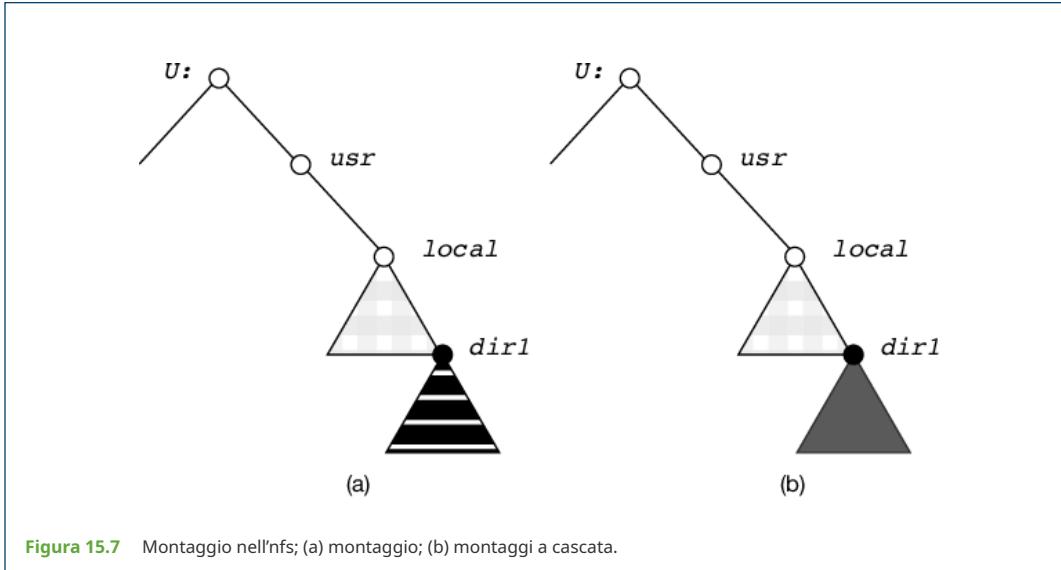


Figura 15.7 Montaggio nell' nfs; (a) montaggio; (b) montaggi a cascata.

Potenzialmente ogni file system o ogni directory in un file system, nel rispetto dei diritti d'accesso, si possono montare in modo remoto sopra qualsiasi directory locale. Le stazioni di lavoro prive di dischi possono persino montare i propri file system sulla radice prelevandoli dai server. In alcune versioni dell' nfs sono permessi anche i montaggi a cascata; ciò significa che un file system si può montare in corrispondenza di un altro file system non locale, ma già montato in modo remoto. Un calcolatore vede tuttavia gli effetti dei soli montaggi da esso richiesti. Montando un file system remoto, il client non acquisisce l'accesso ai file system che, eventualmente, erano montati sopra questo; così, il meccanismo di montaggio non ha la proprietà transitiva.

Nella Figura 15.7 (b) sono riportati i montaggi a cascata. Nella figura è riportato il risultato del montaggio di `s2:/usr/dir2` in `U:/usr/local/dir1`, che era già stato montato in modo remoto da S_1 . Gli utenti di U possono accedere ai file di `dir2` usando il prefisso `/usr/local/dir1`. Se si monta un file system condiviso in corrispondenza delle directory iniziali degli utenti di tutti i calcolatori di una rete, un utente può aprire una sessione in qualsiasi stazione di lavoro e prelevare il proprio ambiente di lavoro iniziale. Questa proprietà permette la mobilità dell'utente.

Uno degli scopi nella progettazione dell' nfs era quello di operare in un ambiente eterogeneo di calcolatori, sistemi operativi e architetture di rete. La definizione dell' nfs è indipendente da questi aspetti. Questa indipendenza si ottiene usando primitive rpc costruite su un protocollo di rappresentazione esterna dei dati (*external data representation, xdr*), usato tra due interfacce indipendenti dall'implementazione. Quindi, se il sistema è formato da calcolatori e file system eterogenei adeguatamente interfacciati all' nfs, si possono montare file system di diversi tipi, sia localmente sia in modo remoto.

La definizione dell' nfs distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi d'accesso ai file remoti. Di conseguenza, per questi servizi si definiscono due protocolli distinti: un protocollo di montaggio e un protocollo per gli accessi ai file remoti, il protocollo nfs. I protocolli sono definiti come insiemi di rpc. Queste rpc sono gli elementi di base usati per realizzare, in modo trasparente, l'accesso remoto ai file.

15.8.2 Protocollo di montaggio

Il protocollo di montaggio stabilisce la connessione logica iniziale tra un server e un client. In Solaris ogni calcolatore ha un processo server, esterno al kernel, che esegue le funzioni del protocollo.

Un'operazione di montaggio comprende il nome della directory remota da montare e il nome del calcolatore server in cui tale directory è memorizzata. La richiesta di montaggio si mappa sulla rpc corrispondente e s'invia al server di montaggio in esecuzione nello specifico calcolatore server. Il server conserva una lista di esportazione (la `/etc/dfs/dfstab` nel sistema Solaris, modificabile soltanto da un *superuser*) che specifica i file system locali esportati per il montaggio e i nomi dei calcolatori a cui tale operazione è permessa. La lista può comprendere anche i diritti d'accesso, come la sola scrittura. Per semplificare la manutenzione delle liste di esportazione e delle tabelle di montaggio, si può usare uno schema distribuito di naming per contenere queste informazioni e renderle disponibili agli appropriati client.

Occorre ricordare che qualsiasi directory all'interno di un file system esportato si può montare in modo remoto da un calcolatore accreditato. Quando il server riceve una richiesta di montaggio conforme alla propria lista di esportazione, riporta al client un handle del file da usare come chiave per ulteriori accessi ai file che si trovano all'interno del file system montato. L'handle del file contiene tutte le informazioni di cui ha bisogno il server per identificare un proprio file. Nei termini dell'ambiente unix, l'handle del file è composto da un identificatore di file system e da un numero di *inode* per identificare la specifica directory montata nell'ambito del file system esportato.

Il server contiene anche una lista dei calcolatori client e delle corrispondenti directory correntemente montate. Questa lista si usa soprattutto per scopi amministrativi, per esempio per informare i client che un server sta andando fuori servizio. L'aggiunta o la cancellazione di elementi da questa lista sono gli unici modi in cui il protocollo di montaggio può modificare lo stato del server.

Generalmente un sistema ha una configurazione di montaggio predefinita che si stabilisce nella fase d'avviamento (`/etc/vfstab` in Solaris); tale configurazione si può comunque modificare. Oltre alla procedura di montaggio effettiva, il protocollo di montaggio comprende numerose altre procedure, come lo smontaggio e la restituzione della lista d'esportazione.

15.8.3 Protocollo nfs

Il protocollo nfs offre un insieme di rpc per operazioni su file remoti che supportano le seguenti operazioni:

- ricerca di un file in una directory;
- lettura di un insieme di elementi di una directory;
- manipolazione di collegamenti e di directory;
- accesso ad attributi di file;
- lettura e scrittura di file.

Queste procedure si possono invocare soltanto dopo aver stabilito un handle del file per la directory montata in remoto.

L'omissione delle operazioni `open` e `close` è intenzionale. Una caratteristica importante dei server nfs è l'*assenza dell'informazione di stato*. I server *stateless* non conservano informazioni sui loro client da un accesso all'altro. Dal lato del server nfs non esistono equivalenti della tabella dei file aperti o delle strutture di file di unix, quindi ogni richiesta deve fornire un insieme completo di argomenti, tra cui un identificatore unico di file e un offset assoluto all'interno del file per svolgere le operazioni appropriate. L'architettura che ne deriva è robusta; non si devono prendere misure speciali per ripristinare un server dopo un guasto. Per tale ragione, le operazioni sui file devono essere idempotenti (ripeterle più volte non modifica l'effetto già ottenuto con la prima esecuzione dell'operazione). Per ottenere l'idempotenza, ciascuna richiesta dell'nfs ha un numero di sequenza, che permette al server di determinare duplicazioni o perdite nella sequenza delle richieste.

La presenza della suddetta lista di client sembra violare la proprietà dell'assenza di informazione di stato nel server. Tuttavia, essa non è essenziale ai fini del corretto funzionamento del client o del server, quindi non è necessario ricostruire tale lista dopo la caduta di un server; tale lista può contenere anche dati incoerenti e viene trattata come un semplice suggerimento.

Un'ulteriore implicazione della filosofia dei server senza informazione di stato e un risultato del funzionamento sincrono di una rpc consiste nel fatto che i dati modificati, tra cui riferimenti indiretti e blocchi di stato, si devono scrivere nei dischi del server prima che i risultati siano riportati al client. Un client può cioè ricorrere a una cache per i blocchi di scrittura, ma quando li invia al server, assume che abbiano raggiunto i dischi del server. Un server deve scrivere tutti i dati dell'nfs in modo sincrono. In questo modo la caduta di un server e il successivo ripristino saranno invisibili al client; tutti i blocchi che il server gestisce per il client resteranno intatti. La conseguente perdita di prestazioni può essere rilevante poiché si perdonano i vantaggi derivanti dall'impiego del caching. Le prestazioni si possono incrementare impiegando dispositivi di memoria secondaria con una propria cache non volatile (di solito si tratta di memorie alimentate da una batteria). Il controllore del disco riporta che la scrittura nel disco è avvenuta quando la scrittura è avvenuta nella cache non volatile. Essenzialmente, il calcolatore "vede" una scrittura sincrona molto rapida. Questi blocchi restano intatti anche dopo una caduta del sistema, e periodicamente vengono trasferiti da questa memoria stabile al disco.

Una singola procedura di scrittura dell'nfs è garantita essere l'atomicità e non è inframmezzata ad altre chiamate di scrittura nello stesso file. Tuttavia, il protocollo nfs non fornisce meccanismi di controllo della concorrenza. Poiché ogni chiamata di scrittura o lettura dell'nfs può contenere fino a 8 kb di dati e i pacchetti udp sono limitati a 1500 byte, può essere necessario dividere una chiamata di sistema `write()` in diverse rpc di scrittura. Quindi, due utenti che scrivono nello stesso file remoto possono riscontrare interferenze nei loro dati. Poiché la gestione di meccanismi di lock è inerentemente basata su informazioni di stato, si richiede che un servizio esterno all'nfs fornisca i meccanismi di locking, come nel caso di Solaris. Gli utenti sanno che per coordinare l'accesso ai file condivisi devono usare meccanismi che sono al di fuori dell'ambito dell'nfs.

L'nfs è integrato nel sistema operativo tramite un vfs. Per illustrarne l'architettura si può descrivere il modo in cui si gestisce un'operazione su un file remoto già aperto (Figura 15.8). Il client inizia l'operazione con un'ordinaria chiamata di sistema. Lo strato del sistema operativo mappa tale chiamata di sistema in un'operazione del vfs sull'opportuno `vnode`. Lo strato vfs identifica il file come remoto e invoca l'opportuna procedura dell'nfs. Avviene quindi una chiamata rpc allo strato del servizio nfs nel server remoto. Tale chiamata si reintroduce nello strato del vfs del sistema remoto, che riconosce essere locale e invoca l'appropriata operazione del file system. Questo cammino si ripercorre al contrario per restituire il risultato. Un vantaggio di tale architettura è che il client e il server sono identici; così un calcolatore può essere un client, un server o entrambi. L'effettivo servizio su ciascun server è eseguito da thread del kernel.

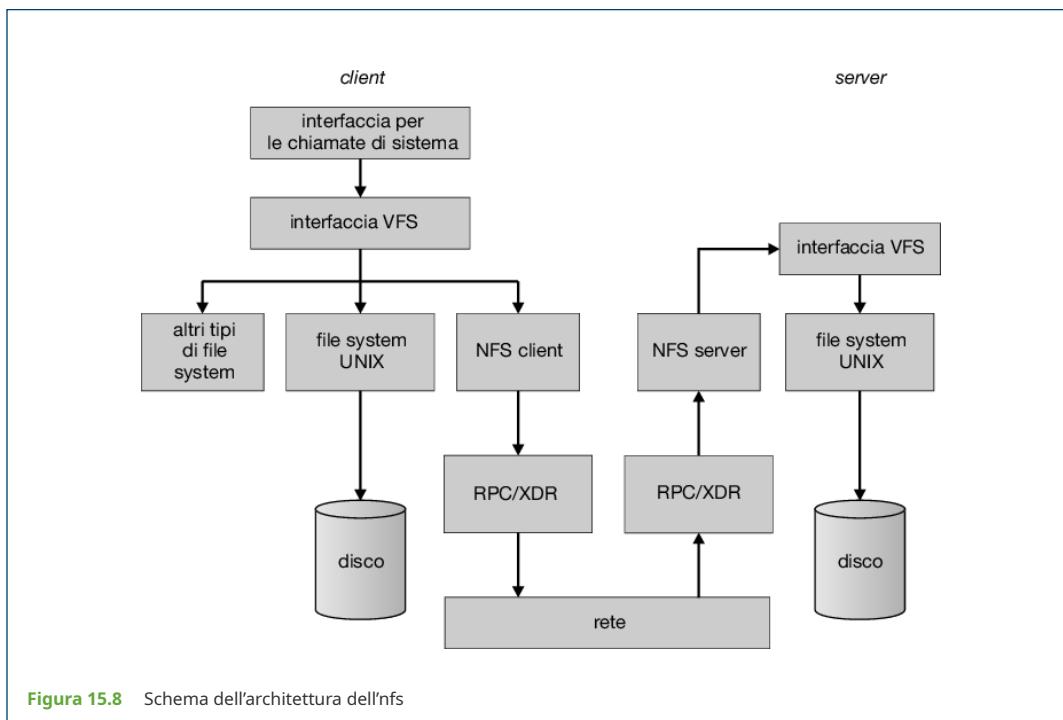


Figura 15.8 Schema dell'architettura dell'nfs

15.8.4 Traduzione dei nomi di percorso

La traduzione dei nomi di percorso (*path name translation*) del protocollo nfs prevede l'analisi di un nome di percorso – per esempio `/usr/local/dir1/file.txt` – al fine di estrarre i nomi delle singole directory, o componenti – nell'esempio: (1) `usr`, (2) `local` e (3) `dir1`. La traduzione dei nomi di percorso si compie suddividendo il percorso stesso in nomi di componenti ed eseguendo una chiamata `lookup()` dell'nfs separata per ogni coppia formata da un nome di componente e un *vnode* del directory. Quando si attraversa un punto di montaggio, ogni `lookup` di un componente causa una rpc separata al server. Questo schema di attraversamento del nome di percorso è costoso ma necessario, poiché ogni client ha un'unica configurazione del proprio spazio di nomi logico, determinata dai montaggi che ha eseguito. Sarebbe molto più efficiente consegnare un nome di percorso a un server e ricevere un *vnode* di destinazione una volta incontrato un punto di montaggio. Tuttavia dovunque nel percorso può essere presente un altro punto di montaggio per quel particolare client, sconosciuto al server stateless.

Una cache per la ricerca dei nomi delle directory, nel client, conserva i *vnode* per i nomi delle directory remote; in questo modo si accelerano i riferimenti ai file con lo stesso nome di percorso iniziale. Se gli attributi restituiti dal server non corrispondono agli attributi del *vnode* nella cache, si scarta il contenuto della cache della directory.

Occorre ricordare che alcune implementazioni dell'nfs permettono il montaggio di un file system remoto sopra un altro file system remoto già montato; si tratta del *montaggio a cascata*. Quando un client ha un montaggio a cascata, nel caso di un attraversamento di un nome di percorso può essere coinvolto più di un server. Tuttavia, quando un client fa una ricerca in una directory sulla quale il server ha montato un file system, il client vede la directory sottostante e non la directory montata.

15.8.5 Operazioni remote

Eccetto che per l'apertura e la chiusura dei file, tra le normali chiamate di sistema di unix per operazioni su file e le rpc del protocollo nfs esiste una corrispondenza quasi uno a uno. Quindi, un'operazione remota su un file si può tradurre direttamente nella rpc corrispondente. Dal punto di vista concettuale l'nfs aderisce al paradigma del servizio remoto, ma in pratica si usano tecniche di buffering e cache per migliorare le prestazioni. La corrispondenza tra un'operazione remota e una rpc non è diretta; invece le rpc prelevano blocchi e attributi del file e li memorizzano localmente nelle cache. Le successive operazioni remote usano i dati nella cache, purchè siano rispettati i vincoli di coerenza.

Esistono due cache: la cache degli attributi dei file (informazioni degli *inode*) e la cache dei blocchi del file. Quando un file viene aperto, il kernel fa un controllo col server remoto per stabilire se deve prelevare o riconvalidare gli attributi nella cache: i blocchi del file che sono in cache si usano solo se i corrispondenti attributi sono aggiornati nella loro cache. La cache degli attributi viene aggiornata ogni volta che arrivano nuovi attributi dal server. Gli attributi nella cache si scartano, per default, dopo 60 secondi. Tra il server e il client si usano le tecniche di lettura anticipata (*read-ahead*) e scrittura differita (*delayed-write*). Finché il server non ha confermato che i dati sono stati scritti nei dischi, i client non liberano i blocchi di scrittura differita. La scrittura differita viene mantenuta anche quando si apre un file concorrentemente, in modi conflittuali. Ne deriva che la semantica unix (Paragrafo 15.7.1) non si conserva.

Mettere a punto il sistema per migliorarne le prestazioni rende difficile caratterizzare la semantica della coerenza dell' nfs. File nuovi creati in un calcolatore possono non essere visibili in altri calcolatori per 30 secondi. Inoltre, le scritture eseguite in un file in un sito possono o meno essere visibili anche in altri siti che hanno aperto lo stesso file per la lettura. Le nuove aperture di un file consentono di osservare solo le modifiche già inviate al server, quindi l' nfs non fornisce né una stretta emulazione della semantica unix, né la semantica delle sessioni di Andrew (Paragrafo 15.7.2). Nonostante questi inconvenienti, l'utilità e le alte prestazioni del meccanismo ne fanno il sistema distribuito multi-vendor più usato.

15.9 Sommario

- I sistemi operativi general-purpose forniscono molti tipi di file system, da quelli destinati a scopi specifici a quelli più generali.
- I volumi contenenti i file system possono essere montati nello spazio dei file system del computer.
- A seconda del sistema operativo, lo spazio dei file system può essere unico (i file system montati sono integrati nella struttura delle directory) o distinto (a ogni file system montato viene assegnata una denominazione differente).
- Almeno un file system deve essere avviabile, ovvero deve contenere un sistema operativo, affinché il sistema sia in grado di avviarsi. Il boot loader viene eseguito per primo; si tratta di un programma semplice che è in grado di trovare il kernel nel file system, caricarlo e avviare la sua esecuzione. I sistemi possono contenere più partizioni avviabili, consentendo all'amministratore di sceglierne una all'avvio.
- La maggior parte dei sistemi è multiutente e deve pertanto fornire un meccanismo per la condivisione e la protezione dei file. I file e le directory includono spesso metadati, come le autorizzazioni di accesso per proprietario, utente e gruppo.
- Le partizioni dei dispositivi di archiviazione di massa vengono utilizzate per l'i/o di basso livello o per i file system. Ogni file system risiede in un volume, che può essere composto da una partizione o da più partizioni che lavorano insieme grazie a un gestore di volumi.
- Per semplificare l'implementazione di più file system distinti, un sistema operativo può utilizzare un approccio a strati, con un'interfaccia virtuale di file system in grado di nascondere le differenze tra file system possibilmente dissimili durante l'accesso.
- I file system remoti possono essere implementati in maniera semplice utilizzando programmi come ftp o server e client web, oppure tramite un modello client-server capace di offrire un maggior numero di funzionalità. Occorre autenticare le richieste di montaggio di un file system e l'id utente per impedire accessi non consentiti.
- Le strutture client-server non condividono le informazioni in modo nativo, ma può essere utilizzato un sistema di naming distribuito come il dns per consentire una tale condivisione, fornendo spazio dei nomi unificato, gestione delle password e identificazione del sistema. Per esempio, Microsoft cifs utilizza active directory, che sfrutta una versione del protocollo di autenticazione di rete Kerberos per fornire un insieme completo di servizi di naming e autenticazione tra i computer in una rete.
- Una volta che la condivisione dei file è resa possibile, deve essere scelto e implementato un modello di semantica della coerenza che permetta la gestione di accessi simultanei a uno stesso file. Tra questi modelli ci sono la semantica unix, la semantica delle sessioni e la semantica dei file condivisi immutabili.
- nfs è un esempio di un file system remoto, che fornisce ai client un accesso diretto alle directory, ai file e persino a interi file system. Un file system remoto con funzionalità complete include un protocollo di comunicazione con operazioni remote e traduzione dei nomi dei percorsi.

Esercizi di ripasso

15.1 Spiegate come lo strato vsf permetta al sistema operativo di supportare facilmente diversi tipi di file system.

15.2 Perché può essere utile avere più di un tipo di file system su uno stesso sistema?

15.3 Descrivete come utilizzare l'interfaccia procfs per esplorare lo spazio dei nomi dei processi su un sistema Unix o Linux (che implementi il file system procfs). Quali aspetti dei processi possono essere visualizzati tramite questa interfaccia? Come si possono raccogliere le stesse informazioni su un sistema privo del file system procfs?

15.4 Perché alcuni sistemi integrano i file system montati nella struttura dei nomi del file system di root, mentre altri usano un meccanismo di denominazione separato?

15.5 Vista l'esistenza di funzionalità di accesso ai file remoti come `ftp`, perché sono stati creati file system remoti come `nfs`?

Esercizi

15.6 Considerate l'integrazione seguente a un protocollo per l'accesso remoto ai file. Ciascun client gestisce una cache per i nomi, in cui memorizza le traduzioni dei nomi dei file nei corrispondenti handle. Quali problemi dovete tenere in considerazione nel realizzare la cache per i nomi?

15.7 In caso di arresto anomalo del sistema o di un'interruzione di corrente mentre un file system è montato e sono in corso operazioni di scrittura, dite che cosa occorre fare prima di montare di nuovo il file system nei seguenti casi.

- (a) File system con logging.
- (b) File system senza logging.

15.8 Perché i sistemi operativi montano automaticamente il file system di root all'avvio?

15.9 Perché i sistemi operativi richiedono di montare altri file system oltre a quello di root?

15.10 Discutete i vantaggi e gli svantaggi di associare ai file system remoti (memorizzati su file server) un insieme di semantiche di fallimento (*failure semantics*) differente da quello associato al file system locale.

15.11 Quali sono le implicazioni del supportare la semantica della coerenza di unix nell'accesso condiviso a file memorizzati su un file system remoto?

CAPITOLO 16

Sicurezza

Protezione e sicurezza sono di vitale importanza per i sistemi informatici. Distinguiamo tra questi due concetti: la sicurezza misura la fiducia nel fatto che l'integrità di un sistema e dei suoi dati siano preservati; la protezione è l'insieme di meccanismi che controllano l'accesso di processi e utenti alle risorse di un sistema informatico. In questo capitolo ci concentreremo sulla sicurezza, mentre la protezione sarà trattata nel Capitolo 17.

La sicurezza si occupa di preservare le risorse del computer da accessi non autorizzati, distruzione o alterazione dolosa e da involontaria introduzione di elementi di incoerenza. Queste risorse includono l'informazione memorizzata nel sistema sotto forma di dati e programmi, così come cpu, memoria, dischi, nastri e connessioni di rete che l'elaboratore gestisce. In questo capitolo partiamo esaminando le modalità con cui può verificarsi l'uso improprio delle risorse, sia esso intenzionale o fortuito. Prenderemo quindi in considerazione un fondamentale meccanismo di abilitazione della sicurezza: la crittografia. In conclusione studieremo i meccanismi in grado di riconoscere e neutralizzare gli attacchi.

16.1 Il problema della sicurezza

Per molte applicazioni la garanzia della sicurezza nei sistemi elaborativi merita un impegno di notevole portata. I grandi sistemi commerciali, contenenti le retribuzioni aziendali o altri documenti finanziari, sono obiettivi invitanti per i ladri. Sistemi che custodiscono i dati inerenti all'operatività di un'azienda possono suscitare l'interesse di concorrenti senza scrupoli. A ciò si aggiunga che la perdita involontaria di tali dati, o la loro sottrazione dolosa, può pregiudicare gravemente il funzionamento di un'azienda. Anche le risorse informatiche di base sono interessanti per gli aggressori, perché consentono il mining di bitcoin e l'invio di spam, e costituiscono un punto di partenza da cui attaccare in modo anonimo altri sistemi.

Nel Capitolo 17 sono trattati alcuni meccanismi offerti dai sistemi operativi (con l'ausilio di appropriate caratteristiche dell'hardware) per consentire agli utenti di proteggere le loro risorse (di solito programmi e dati). Questi meccanismi funzionano bene fintanto che gli utenti si conformano alle modalità d'uso e d'accesso previste per tali risorse. Si dice che un sistema è sicuro se, in ogni circostanza, vi si accede e si utilizzano le risorse solo secondo le modalità previste. Sfortunatamente non è possibile ottenere una sicurezza totale; ciononostante si deve poter disporre di meccanismi che rendano l'elusione della sicurezza un caso raro e non la norma.

Le violazioni della sicurezza del sistema si possono classificare come intenzionali (dolose) o accidentali. È più semplice proteggere un sistema contro le violazioni accidentali che contro quelle dolose. I meccanismi di protezione sono, nella maggior parte dei casi, la base per la difesa dalle violazioni di sicurezza. Nell'elenco che segue sono comprese sia le intrusioni accidentali sia le violazioni dolose. È bene notare che indicheremo con le espressioni *intruso*, *aggressore* e *pirata informatico* coloro che tentino di attuare infrazioni della sicurezza; inoltre chiameremo *minaccia* ogni potenziale pericolo per la sicurezza (quale, per esempio, la scoperta di una vulnerabilità), mentre il termine *attacco* denoterà il tentativo di infrangere le misure di sicurezza.

- **Violazione della riservatezza.** Questo tipo di violazione consiste nella lettura non autorizzata dei dati (o nel furto di informazioni), una finalità tipica degli intrusi. Con la sottrazione di dati segreti da un sistema, o l'intercettazione i dati in transito, un intruso può ricavare denaro da informazioni, quali i numeri delle carte di credito o dalle informazioni sull'identità dell'utente.
- **Compromissione dell'integrità.** Questa violazione si realizza con la modifica non autorizzata dei dati. Attacchi simili possono, per esempio, causare il trasferimento di una responsabilità a un soggetto innocente o la modifica del codice sorgente di un'importante applicazione commerciale.
- **Violazione della disponibilità.** Questo abuso consiste nella distruzione non autorizzata di dati. Alcuni pirati informatici (*cracker*), pur di vedere accresciuta la propria "reputazione", preferiscono infliggere danni devastanti ai sistemi piuttosto che guadagnare denaro. Il sabotaggio dei siti web è un classico esempio di questo tipo di infrazione.
- **Appropriazione del servizio.** Questa violazione è relativa all'uso non autorizzato delle risorse. A titolo di esempio, un intruso (o un programma di intrusione) potrebbe installare un demone che funga da file server all'interno di un sistema.
- **Rifiuto del servizio.** Questa violazione impedisce l'utilizzo legittimo del sistema. Gli attacchi di rifiuto del servizio, o dos (*Denial-Of-Service*), sono talvolta accidentali. Il primo worm di Internet si trasformò in un attacco dos, quando un baco causò la sua rapida proliferazione. Ritorneremo sugli attacchi dos più avanti, nel Paragrafo 16.3.2.

Gli aggressori ricorrono a numerosi metodi standard per tentare di infrangere la sicurezza. Il più comune è l'attacco mimetico (*masquerading*), che si realizza quando, in una comunicazione, un partecipante finge di essere qualcun altro (un'altra persona o un'altra macchina). Gli aggressori usano l'attacco mimetico per violare l'autenticazione (*authentication*), cioè la correttezza dell'identificazione; in questo modo possono ottenere permessi d'accesso che normalmente gli sarebbero negati, oppure scalare privilegi, appropriandosi di diritti per i quali non avrebbero titolo. La riproduzione di informazioni catturate nell'ambito di uno scambio di dati rappresenta un'altra modalità di attacco. Un attacco replay (*replay attack*) è basato sulla ripetizione dolosa o fraudolenta di informazioni valide già trasmesse. Talvolta l'attacco si esaurisce nella semplice riproduzione, con la reiterazione, per esempio, di una richiesta di trasferimento di denaro. Spesso invece è accompagnato da una modifica del messaggio (*message modification*) che mira, ancora una volta, ad accaparrarsi privilegi non dovuti. Si consideri il danno che potrebbe sorgere se, per una richiesta di autenticazione, si sostituissero alle vere informazioni su un utente quelle di un utente non autorizzato. Ancora diversa è la tipologia di attacco di interposizione (*man-in-the-middle attack*), che avviene quando, nel flusso di dati di una comunicazione, si intrmette un attaccante, che imita il trasmettitore nei confronti del ricevitore, e viceversa. In una comunicazione di rete, un attacco man-in-the-middle può essere preceduto da un dirottamento della sessione (*session hijacking*), con cui è intercettata una sessione di comunicazione attiva.

Un'altra vasta classe di attacchi è finalizzata all'acquisizione indebita dei privilegi. Ogni sistema assegna i privilegi agli utenti, anche qualora esistesse un solo utente e tale utente fosse l'amministratore. Generalmente, il sistema include diversi insiemi di privilegi, uno per ciascun account utente e alcuni per il sistema. I privilegi vengono spesso assegnati anche agli utenti non registrati sul sistema, come un generico utente della rete Internet che accede a una pagina web senza effettuare alcun login o l'utente anonimo di servizi come il trasferimento di file. Persino il mittente di un messaggio di posta elettronica diretto a un sistema remoto può essere considerato in possesso di privilegi: il privilegio di inviare un'e-mail all'utente destinatario su quel sistema. L'acquisizione indebita dei privilegi offre agli aggressori più privilegi di quanti ne dovrebbero avere. Per esempio, un'e-mail contenente uno script o una macro che viene eseguita sul sistema supera i privilegi che il mittente dell'e-mail dovrebbe avere. Tecniche come l'attacco mimetico e la modifica del messaggio, menzionate in precedenza, vengono spesso utilizzate per aumentare i privilegi. Ci sono molti altri esempi, poiché questo è un tipo di attacco molto comune e, in effetti, è difficile rilevare e prevenire tutti i tipi di attacchi che ricadono in questa categoria.

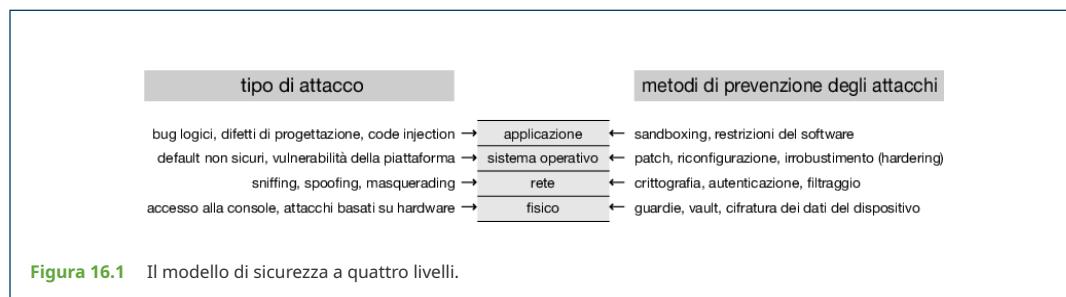
Come si è detto, una protezione totale del sistema dalle violazioni non è possibile, ma le si può rendere di complessità tale da scoraggiare la maggior parte degli intrusi. In alcuni casi, quali gli attacchi dos, anche se è preferibile evitare l'attacco, è comunque sufficiente che esso sia rilevato per adottare contromisure appropriate.

Per proteggere il sistema è necessario prendere misure di sicurezza a quattro livelli.

1. **Fisico.** I siti che ospitano i sistemi elaborativi devono essere protetti fisicamente contro gli accessi armati o furtivi da parte d'intrusi. Vanno protetti sia i luoghi che ospitano le macchine sia le stazioni di lavoro o i terminali che vi hanno accesso, per esempio limitando l'accesso all'edificio in cui risiedono o ancorandoli alle scrivanie sulle quali sono collocati.

2. Rete. La maggior parte dei sistemi informatici contemporanei, dai server ai dispositivi mobili, ai dispositivi Internet of Things (IoT), è collegata in rete. La rete fornisce un mezzo al sistema per accedere a risorse esterne, ma fornisce anche un potenziale vettore per l'accesso non autorizzato al sistema stesso.
3. Inoltre, i dati dei computer nei sistemi moderni viaggiano in linee di comunicazione private, linee condivise (come quelle della rete Internet), connessioni wireless o accessi dial-up. L'intercettazione di questi dati può essere tanto dannosa quanto l'intrusione in un calcolatore. L'interruzione di queste comunicazioni potrebbe costituire un attacco dos remoto, che riduce le possibilità d'uso da parte degli utenti e l'affidamento che si può fare sul sistema.
4. Sistema operativo. Il sistema operativo e il suo insieme integrato di applicazioni e servizi costituiscono un'enorme mole di codice che può ospitare molte vulnerabilità. Le impostazioni predefinite non sicure, gli errori di configurazione e i bug di sicurezza sono solo alcuni dei potenziali problemi. I sistemi operativi devono quindi essere sempre aggiornati (tramite una continua installazione di patch) e "fortificati", ovvero configurati e modificati per ridurre la superficie di attacco ed evitare la penetrazione. La superficie di attacco è l'insieme dei punti da cui un utente malintenzionato può tentare l'accesso al sistema.
5. Applicazione. Anche le applicazioni di terze parti possono comportare dei rischi, specialmente se possiedono privilegi significativi. Alcune applicazioni sono intrinsecamente dannose, ma anche le applicazioni benigne possono contenere bug di sicurezza. A causa del vasto numero di applicazioni di terze parti e dei loro diversi codici sorgenti, è praticamente impossibile garantire che tutte queste applicazioni siano sicure.

Questo modello di sicurezza a quattro livelli è mostrato nella Figura 16.1.



Il modello di sicurezza a quattro livelli è come una catena formata da anelli collegati: una vulnerabilità in uno qualsiasi dei suoi livelli può compromettere l'intero sistema. A tal proposito, è vero il vecchio adagio secondo cui la sicurezza è tanto forte quanto il suo anello più debole.

Un altro fattore che non può essere trascurato è quello umano. L'autorizzazione degli utenti richiede cautela, per garantire che accedano al sistema solo gli utenti che ne hanno diritto. Persino gli utenti autorizzati, tuttavia, potrebbero essere malintenzionati, oppure "incoraggiati" a cedere le loro credenziali ad altri volontariamente o mediante tecniche di ingegneria sociale (*social engineering*) che utilizzano l'inganno per convincere le persone a rivelare informazioni riservate. Una tipologia di attacco che si serve dell'ingegneria sociale è il phishing.¹ Questa tecnica consiste nel contraffare e-mail o pagine web, rendendole in tutto simili a quelle autentiche, per spingere gli utenti trattati in inganno a comunicare informazioni confidenziali. A volte è sufficiente un clic su un collegamento presente in una pagina del browser o in un'email per scaricare inavvertitamente un malware, compromettendo la sicurezza del sistema sul computer dell'utente. Di solito l'obiettivo finale non è quel PC, ma piuttosto una risorsa più preziosa: dal sistema compromesso vengono effettuati attacchi verso altri sistemi sulla lan o verso altri utenti.

Abbiamo visto finora che per mantenere la sicurezza devono essere presi in considerazione tutti e quattro i fattori nel modello a quattro livelli, più il fattore umano. Inoltre, il sistema deve fornire protezione (discussa in dettaglio nel Capitolo 17) per consentire l'implementazione di funzionalità di sicurezza.

Senza la facoltà di autorizzare utenti e processi, di controllarne l'accesso, e di registrare le loro attività, sarebbe impossibile, per un sistema operativo, un'esecuzione sicura o l'implementazione delle procedure di sicurezza. Le funzionalità di protezione fornite dall'hardware sono indispensabili per mettere in atto una complessiva strategia di protezione. Un sistema privo di protezione della memoria, per esempio, non può essere sicuro. Le più recenti funzionalità dell'hardware, come vedremo, favoriscono una migliore tutela dei sistemi.

Sfortunatamente la sicurezza ha ben poco di prevedibile. Contromisure di sicurezza vengono elaborate e messe in atto dopo che gli intrusi hanno sfruttato i punti deboli di un sistema; ciò spinge gli incursori a raffinare le modalità di attacco. Tra gli episodi recenti di violazione si può citare, per esempio, l'utilizzo di programmi *spyware*, che sfruttano sistemi ignari per inviare posta elettronica indesiderata, detta *spam* (illustreremo questa prassi nel Paragrafo 16.2). È probabile che questa rincorsa continuerà, con la necessità di incrementare gli strumenti per bloccare tecniche e attività perfezionate dagli incursori.

Nel seguito del capitolo tratteremo la sicurezza a livello del sistema operativo; la sicurezza ai livelli umano e fisico, benché importante, va oltre gli scopi di questo libro. La sicurezza nei sistemi operativi e tra sistemi operativi si realizza in diversi modi: dalle password per l'autenticazione alla vigilanza contro i virus, fino alla scoperta delle intrusioni. Per cominciare, tratteremo le minacce alla sicurezza.

16.2 Minacce legate ai programmi

I processi, insieme al kernel, costituiscono per un elaboratore gli unici strumenti per portare a termine del lavoro. Di conseguenza, scrivere un programma che crei una falla nella sicurezza o introdurre anomalie nell'esecuzione di un processo per ottenere il medesimo risultato, è un normale obiettivo dei pirati informatici. In realtà, molte infrazioni alla sicurezza indipendenti dai programmi hanno spesso l'obiettivo di generare minacce ai programmi stessi. Benché sia utile, per esempio, accedere a un sistema senza autorizzazione, è di gran lunga più vantaggioso lasciarsi alle spalle un demone di back-door (letteralmente: *porta sul retro*), in grado di fornire informazioni o procurare facilmente l'accesso, anche se l'attacco principale è stato bloccato. In questo paragrafo analizziamo i metodi più comuni con cui i programmi possono generare violazioni della sicurezza. Poiché vi è notevole discrepanza nelle definizioni convenzionali delle fallo di sicurezza, useremo i termini più comuni o più descrittivi.

16.2.1 Malware

Il malware è un software progettato per sfruttare, disabilitare o danneggiare i sistemi informatici. Esistono molti modi per realizzare questi scopi e qui ne esploriamo le principali varianti.

Molti sistemi dispongono di meccanismi che permette agli utenti di usare programmi scritti da altri. Se questi programmi si eseguono in un dominio che fornisce i diritti d'accesso dell'utente che esegue il programma, gli altri utenti possono abusare di questi diritti. Un programma che agisce in modo clandestino o malevolo, anziché eseguire semplicemente la sua funzione dichiarata, è chiamato cavallo di Troia. In alcune condizioni, un programma può aumentare i suoi privilegi: come esempio, si consideri un'app per dispositivi mobili che sembra fornire funzionalità utili, per esempio un'app torcia, ma che nel frattempo accede furtivamente ai contatti o ai messaggi dell'utente e li invia a un server remoto.

Una variante classica del cavallo di Troia è un programma (detto "trojan mule") che emula una procedura di login: l'ignaro utente, nella fase d'accesso a un terminale, crede di aver scritto erroneamente la propria password; prova ancora e, questa volta, ha successo. Ciò che realmente accade in un caso come questo è che un emulatore della procedura d'accesso alla sessione di lavoro sottrae il nome utente e la password dell'utente. L'emulatore copia la password e mostra un messaggio d'errore nell'inserimento dei dati, quindi interrompe la propria esecuzione e lascia l'utente di fronte alla vera procedura d'accesso. Questo tipo d'attacco può essere respinto dal sistema operativo mostrando un messaggio informativo alla fine di ogni sessione di lavoro interattiva o utilizzando per il login una sequenza di caratteri non "catturabile" inviata dalla tastiera, come la sequenza **Ctrl-Alt-Canc** impiegata dai moderni sistemi operativi Windows. Inoltre, l'utente può assicurarsi che l'url digitato sia corretto e valido.

Un'altra variante del cavallo di Troia è il cosiddetto spyware. Lo spyware talvolta accompagna un programma che l'utente ha scelto di installare. Nel caso più frequente, è annesso ai programmi per uso gratuito (*freeware*) o privi di licenza commerciale (*shareware*), ma può essere incluso anche in quelli commerciali. La finalità di un programma spyware è di visualizzare annunci pubblicitari sullo schermo dell'utente, creare finestre a comparsa nel browser quando si visitano alcuni siti, o prelevare informazioni dal sistema dell'utente per trasmetterle a un sito di raccolta. Quest'ultima modalità rientra nella categoria di attacchi noti in genere come canali coperti (*covert channel*), in cui si stabiliscono comunicazioni nascoste. Per esempio, un programma in apparenza inoffensivo, installabile su un sistema Windows, potrebbe portare al caricamento di un demone spyware. Lo spyware potrebbe contattare un sito centrale, ricevere da esso un messaggio con una lista di indirizzi dei destinatari, e recapitare spam a tali utenti dalla macchina Windows. Questo meccanismo si ripete fintanto che l'utente scopre lo spyware: spesso, però, non si giunge neppure a scoprirla. Nel 2010 si calcola che il 90 per cento dello spam sia stato inviato in questa maniera. Nella maggior parte dei paesi questa appropriazione del servizio non è neanche considerata reato!

Uno sviluppo abbastanza recente e sgradito è costituito dai malware, detti ransomware, che non rubano informazioni, ma che sono in grado di crittografare (parzialmente o totalmente) le informazioni presenti sul computer di destinazione e renderle inaccessibili al proprietario. L'informazione oggetto dell'attacco ha di solito poco valore per l'aggressore, ma molto valore per il proprietario. L'idea alla base di questi attacchi è di costringere il proprietario a pagare un riscatto (in inglese "ransom") per ottenere la chiave necessaria per decifrare i dati. Come avviene generalmente nei rapporti con i criminali, ovviamente, il pagamento del riscatto non garantisce di avere nuovamente accesso ai propri dati.

I cavalli di Troia e gli altri malware hanno particolare successo nei casi in cui vi sia una violazione del principio del minimo privilegio. Ciò si verifica in genere quando il sistema operativo consente per impostazione predefinita più privilegi rispetto a quelli di cui un normale utente ha bisogno o quando l'utente è, per default, in esecuzione come amministratore di sistema (come è avvenuto in tutti i sistemi operativi Windows fino a Windows 7). In questi casi, il sistema immunitario del sistema operativo – permessi e protezioni di vario genere – non è in grado di entrare in azione e il malware può persistere e sopravvivere anche in seguito a riavvi, oltre a poter estendere la sua portata sia a livello locale sia attraverso la rete.

La violazione del principio del minimo privilegio è un caso di scarsa capacità decisionale nella progettazione del sistema operativo. Un sistema operativo (e, in effetti, il software in generale) dovrebbe consentire un controllo accurato dell'accesso e della sicurezza, in modo che solo i privilegi necessari per eseguire un'attività siano disponibili durante l'esecuzione dell'attività. Inoltre, le funzioni di controllo devono essere facili da gestire e da capire. Le misure di sicurezza scomode da utilizzare, inadeguate e incomprensibili sono destinate a essere aggirate, causando un indebolimento generale della sicurezza che le stesse misure sono state progettate per implementare.

IL PRINCIPIO DEL MINIMO PRIVILEGIO

“Il principio del minimo privilegio: in un sistema, ogni programma e ogni utente dotato di privilegi dovrebbero operare con il minimo privilegio necessario per completare il proprio lavoro, allo scopo di ridurre il numero di potenziali interazioni tra programmi privilegiati al minimo necessario per poter operare correttamente, in modo che si possa essere ragionevolmente fiduciosi del non verificarsi di usi non intenzionali, indesiderati o impropri del privilegio.” Jerome H. Saltzer, nella descrizione di un principio di progettazione del sistema operativo Multics nel 1974:

<https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf>.

In un’altra forma di malware, il progettista di un programma o di un sistema può lasciare nel programma un “buco” segreto che lui solo è in grado di utilizzare. Questo tipo di violazione della sicurezza, detto trabocchetto (*trap door*), è stato mostrato nel film *War Games*. Il codice può per esempio cercare uno specifico userid, o una determinata password, e può quindi aggirare le normali procedure di sicurezza. Alcuni programmatore sono stati arrestati per aver truffato banche inserendo errori d’arrotondamento nel loro codice per ottenere l’accredito nei propri conti dei risultanti mezzi centesimi di dollaro. Considerato il numero di transazioni eseguite da una grande banca, con tali accrediti si possono raggiungere delle cospicue quantità di denaro.

Una trap door può essere impostata per operare solo al verificarsi di un dato insieme di condizioni logiche, nel qual caso viene chiamata bomba logica (*logic bomb*). Le back door di questo tipo sono particolarmente difficili da rilevare, poiché possono rimanere latenti per un lungo periodo, anche anni, prima di essere scoperte, di solito dopo che il danno è stato fatto. È accaduto, per esempio, che un amministratore di rete impostasse l’esecuzione di una riconfigurazione distruttiva della rete della sua azienda nel caso in cui il programma avesse rilevato che non era più impiegato presso la società.

Un abile trabocchetto si potrebbe inserire in un compilatore, che in questo caso potrebbe generare sia un codice oggetto normale sia un codice oggetto contenente un trabocchetto, a prescindere dal codice sorgente da compilare. Si tratta di un’attività particolarmente nefasta, poiché un’ispezione del codice sorgente del programma non rivelerebbe alcun problema e solo il reverse engineering del codice del compilatore permetterebbe di individuare il trabocchetto. Questo tipo di attacco può essere eseguito anche sfruttando patch per il compilatore o per le librerie di compilazione. Per esempio, nel 2015, un malware destinato alla suite di compilatori XCode di Apple (soprannominato “XCodeGhost”) ha colpito molti sviluppatori di software che hanno utilizzato versioni compromesse di XCode non scaricate direttamente dal sito Apple.

I trabocchetti costituiscono un problema difficile: per individuarli è necessario analizzare tutto il codice sorgente dei componenti di un sistema. Poiché i sistemi possono essere composti da milioni di righe di codice, queste analisi non si fanno spesso, e spesso non si fanno mai. Una metodologia di sviluppo del software che può aiutare a contrastare questo tipo di lacune nella sicurezza è la revisione del codice: lo sviluppatore invia il codice implementato a una codebase, e uno o più sviluppatori riesaminano il codice e lo approvano, oppure forniscono commenti. Una volta che un determinato gruppo di revisori approva il codice (a volte dopo che i problemi rilevati sono stati risolti e il codice è stato ripresentato e riesaminato), il codice è ammesso nella codebase e viene quindi compilato, debuggato e infine rilasciato per l’uso. Molti buoni sviluppatori di software utilizzano sistemi di controllo delle versioni che forniscono strumenti per la revisione del codice, per esempio git (<https://github.com/git/>). Si noti, inoltre, che esistono strumenti automatici di revisione e scansione del codice, progettati per trovare difetti, incluse eventuali falliche di sicurezza. In genere, però, i bravi programmatore sono i migliori revisori del codice.

Per coloro che non sono direttamente coinvolti nello sviluppo del codice, la revisione del codice è utile per trovare e segnalare difetti, o per trovarli e sfruttarli. Nella maggior parte dei casi, il codice sorgente di un software non è disponibile e la revisione del codice da parte dei non sviluppatori è molto più difficile.

16.2.2 Code injection

La maggior parte dei software non è malevola, ma può comunque rappresentare una seria minaccia per la sicurezza a causa di attacchi di tipo code injection (“iniezione di codice”), per mezzo dei quali viene aggiunto o modificato codice eseguibile. Un software benigno può ospitare vulnerabilità che, se sfruttate, consentono a un utente malintenzionato di assumere il controllo del codice del programma, alterando il flusso del codice esistente o riprogrammandolo interamente con un nuovo codice.

Gli attacchi code injection sono quasi sempre il risultato di paradigmi di programmazione scadenti o non sicuri, e sono comuni in linguaggi di basso livello come C o C++, che consentono l’accesso diretto alla memoria tramite puntatori. L’accesso diretto alla memoria, unito alla necessità di decidere con cura le dimensioni dei buffer e fare attenzione a non superarle, può portare alla corruzione della memoria qualora i buffer non fossero gestiti correttamente.

Si consideri, per esempio, il vettore di code injection più semplice: l’overflow di un buffer. Il programma nella Figura 16.2 illustra un tale overflow, che si verifica a causa di un’operazione di copia non limitata, la chiamata a `strcpy()`. La funzione esegue la copia senza tener conto della dimensione del buffer in questione, interrompendola solo quando si incontra un byte null (\0). Se un tale byte viene raggiunto prima della dimensione massima del buffer (`buffer_size`) il programma si comporta come previsto, ma che cosa succede se la copia prosegue oltre la dimensione del buffer?

```

#include <stdio.h>

#include <string.h>

#define BUFFER_SIZE 0

int main(int argc, char *argv[])

{
    int j = 0;

    char buffer[BUFFER_SIZE];

    int k = 0;

    if (argc < 2) {return -1;}

    strcpy(buffer, argv[1]);

    printf("K is %d, J is %d, buffer is %s\n", j, k, buffer);

    return 0;
}

```

Figura 16.2 Programma C che esemplifica il buffer overflow.

La risposta è che l'esito di un overflow dipende in gran parte dalla dimensione e dal contenuto della parte in eccesso (Figura 16.3). Inoltre, l'esito varia molto a seconda del codice generato dal compilatore, che può essere ottimizzato in modi che influenzano il risultato: le ottimizzazioni infatti implicano spesso modifiche al layout di memoria (comunemente riposizionando o effettuando il padding delle variabili).

1. Se l'overflow è molto piccolo (se va di poco oltre la dimensione `buffer_size`), ci sono buone probabilità che passi del tutto inosservato. Ciò è dovuto al fatto che l'allocazione di `buffer_size` byte viene spesso completata con byte aggiuntivi, fino a un limite specificato dall'architettura (generalmente 8 o 16 byte). Questo completamento (*padding*) costituisce memoria inutilizzata e un overflow in questa zona di memoria, anche se tecnicamente fuori dai limiti imposti da `buffer_size`, non ha alcun effetto negativo.
2. Se l'overflow supera il riempimento, la successiva variabile automatica presente nello stack viene sovrascritta dal contenuto in eccesso. Il risultato dipenderà in questo caso dalla posizione esatta della variabile e dalla sua semantica (per esempio, se è impiegata in una condizione logica che può essere quindi alterata). In assenza di controlli questo overflow può causare un arresto anomalo del programma, poiché un valore imprevisto in una variabile può portare a un errore non correggibile.
3. Se l'overflow supera di molto il riempimento, tutto il record di attivazione della funzione corrente viene sovrascritto. Nella parte superiore del record si trova l'indirizzo di ritorno della funzione, a cui si accede quando la funzione deve restituire il controllo al chiamante. Il flusso del programma viene quindi stravolto e può essere reindirizzato dall'attaccante verso un'altra area di memoria, inclusa la memoria da lui controllata (per esempio, lo stesso buffer di input, o lo stack oppure l'heap). Il codice iniettato viene quindi eseguito: ciò consente all'autore dell'attacco di eseguire codice arbitrario mantenendo l'id del processo.

Si noti che un programmatore attento avrebbe incluso nel codice controlli sulla dimensione di `argv[1]` usando la funzione `strncpy()` invece di `strcpy()`: la riga `strcpy(buffer, argv[1])` sarebbe allora diventata `strncpy(buffer, argv[1], sizeof(buffer)-1)`. Purtroppo, però, le buone pratiche di controllo sugli indici degli array sono l'eccezione, non la regola. La funzione `strcpy()` appartiene a una nota classe di funzioni vulnerabili che include `sprintf()`, `gets()` e altre funzioni che non tengono conto delle dimensioni del buffer. Tuttavia, anche le varianti di queste funzioni che prendono in considerazione le dimensioni del buffer possono contenere vulnerabilità se accoppiate con operazioni aritmetiche su interi di lunghezza finita, che possono portare a un overflow.

A questo punto, dovrebbero risultare evidenti i pericoli insiti in una semplice svista nella gestione di un buffer. Brian Kerningham e Dennis Ritchie (nel loro libro² *The C Programming Language*) descrivevano il possibile esito come un “comportamento indefinito” ma, in realtà, l’aggressore può forzare un comportamento perfettamente prevedibile, come dimostrato dal Morris Worm (e documentato in rfc1135: <https://tools.ietf.org/html/rfc1135>). Solo diversi anni dopo, tuttavia, un articolo apparso sul numero 49 della rivista *Phrack* (“Smashing the Stack for Fun and Profit” <http://phrack.org/issues/49/14.html>) ha divulgato questa tecnica alle masse, scatenando il moltipliarsi dei tentativi di attacco.

Per ottenere una code injection deve essere presente un codice iniettabile. Per prima cosa, l’autore dell’attacco scrive un breve segmento di codice come il seguente:

```

void func (void) {

```

```
execvp ("/bin/sh", "/bin/sh", NULL);  
  
}  

```

Tramite la chiamata di sistema `execvp()`, questo segmento di codice crea un processo shell. Se il programma sotto attacco è in esecuzione con permessi di root, la shell appena creata otterrà l'accesso completo al sistema. Ovviamente, il segmento di codice può fare qualsiasi cosa consentita dai privilegi del processo attaccato. Il codice viene successivamente compilato e il risultante codice assembly viene trasformato in un flusso binario. Il codice compilato viene spesso definito shellcode, a causa della sua funzione tipica di generare una shell, anche se il significato del termine si è ampliato fino a comprendere qualsiasi tipo di codice, incluso il codice più avanzato utilizzato per aggiungere nuovi utenti a un sistema, per riavviare o persino per connettersi alla rete e attendere istruzioni remote ("reverse shell"). Un exploit shellcode è mostrato nella Figura 16.4. Il codice che viene usato per tempi molto brevi solamente per reindirizzare l'esecuzione verso qualche altra posizione è molto simile a un trampolino che fa "rimbalzare" il flusso del codice da un punto all'altro.

Esistono compilatori di shellcode (un esempio degno di nota è il progetto "MetaSploit") che si preoccupano anche di aspetti specifici come assicurare che il codice sia compatto e non contenga byte null (per quando si sfrutta la copia di stringhe, che terminerebbe sui null). Un simile compilatore può persino mascherare lo shellcode con caratteri alfanumerici.

Se l'autore dell'attacco è riuscito a sovrascrivere l'indirizzo di ritorno (o qualsiasi puntatore a funzione, come quello di una `vTable`), allora tutto ciò che occorre fare, nel caso più semplice, è modificare l'indirizzo per puntare allo shellcode, che viene comunemente caricato come parte dell'input dell'utente, attraverso una variabile di ambiente, o come input da file o da rete. Supponendo che non esistano mitigazioni (come sarà descritto più avanti), questo è sufficiente per l'esecuzione dello shellcode e per il successo dell'attacco dell'hacker. Le questioni relative all'allineamento sono spesso gestite aggiungendo una sequenza di istruzioni `nop` prima dello shellcode. Il risultato è noto come un `nop-sled`, in quanto provoca uno slittamento verso il basso dell'esecuzione in cui si eseguono istruzioni `nop` fino a quando si incontra il payload e lo si esegue.

Questo esempio di attacco di tipo buffer overflow mostra che sono necessarie notevoli conoscenze e abilità di programmazione per riconoscere il codice vulnerabile e per sfruttarlo. Sfortunatamente non servono grandi programmati per lanciare attacchi alla sicurezza, perché dopo che un hacker determina il bug e scrive un exploit, chiunque disponga di abilità informatiche rudimentali e accesso all'exploit (i cosiddetti script kiddie) può provare a lanciare l'attacco ai sistemi di destinazione.

L'attacco buffer overflow è particolarmente deleterio, perché può muoversi tra sistemi e può viaggiare sui canali di comunicazione consentiti. Attacchi di questo tipo possono agire all'interno di protocolli che dovrebbero essere utilizzati per comunicare con la macchina di destinazione e pertanto possono essere difficili da rilevare e prevenire. Inoltre, possono anche superare la sicurezza aggiunta dai firewall (Paragrafo 16.6.6).

Si noti che i buffer overflow sono solo uno dei numerosi vettori che possono essere manipolati per l'iniezione di codice. Gli overflow possono essere sfruttati anche quando si verificano nell'heap. Anche l'uso dei buffer dopo aver liberato la memoria o l'invocazione di più chiamate `free()` su uno stesso buffer può favorire l'iniezione di codice.

16.2.3 Virus e worm

Un'altra minaccia basata su programmi è costituita dai virus. Un virus è un frammento di codice inserito in un programma legittimo. I virus si autoriproducono e sono concepiti in modo da "contagiare" altri programmi; possono recare danni enormi a un sistema, modificandone o distruggendone i file, oltre a causare difetti di funzionamento nei programmi e crash del sistema. Come molti attacchi invasivi, i virus sfruttano le peculiarità di una singola architettura, sistema operativo o applicazione: per gli utenti dei pc rappresentano un problema temibile. unix e gli altri sistemi operativi multiutente, in genere, non sono soggetti ai virus, poiché i loro programmi eseguibili sono protetti dalla scrittura. Se anche un virus infetta tali programmi ha, in linea di massima, una capacità offensiva limitata, perché altri aspetti del sistema sono protetti.

Normalmente i virus si trasmettono per posta elettronica, tramite posta indesiderata (*spam*), e utilizzando tecniche di phishing. Possono diffondersi, inoltre, attraverso lo scambio di dischi infetti tra utenti, o quando si scaricano programmi contaminati dai servizi di condivisione dei file su Internet. Si può fare una distinzione tra i virus, che richiedono attività da parte dell'uomo, e i worm, che usano una rete per replicarsi, senza l'aiuto dell'uomo.

Come esempio di come un virus "infetti" un host, considerate i file di Microsoft Office. Questi file possono contenere delle cosiddette *macro* (cioè, programmi in Visual Basic) che vengono eseguite automaticamente dai programmi del pacchetto Office (Word, PowerPoint ed Excel). Dal momento che tali programmi sono eseguiti sotto l'account dell'utente, le macro possono agire quasi senza controllo (e possono cancellare in maniera indiscriminata i file dell'utente, per esempio). Di norma il virus si autotrasmette, con l'invio di messaggi di posta elettronica agli indirizzi contenuti nella rubrica dell'utente. Dall'esempio di codice seguente si può constatare come sia facile scrivere una macro in Visual Basic, con la quale un virus potrebbe formattare il disco rigido di un calcolatore Windows, non appena il file contenente la macro fosse aperto:

```
Sub AutoOpen()  
  
Dim oFS  
  
Set oFS = CreateObject("Scripting.FileSystemObject")
```

```
    vs = Shell("c: command.com /k format c:",vbHide)  
  
End Sub
```

Di solito i worm inviano anche e-mail autonomamente agli utenti contenuti nella lista dei contatti.

Come funzionano i virus? Dopo che un virus raggiunge la macchina presa di mira, un programma chiamato portatore di virus (*virus dropper*) inserisce il virus nel sistema. Il portatore di virus è solitamente un cavallo di Troia che, sebbene apparentemente eseguito per altre ragioni, ha per vero obiettivo l'installazione del virus. Una volta installato, il virus può fare una serie di cose. Esistono migliaia di virus, che tuttavia possono essere ricondotti ad alcune categorie principali. Molti di loro appartengono a più di una categoria.

- File. Un virus di questo tipo infetta il sistema aggiungendosi in coda a un file. Esso modifica le istruzioni iniziali del programma in modo da far eseguire il proprio codice. Dopo essere stato eseguito restituisce il controllo al programma, in modo da non essere notato. I virus di file sono talora denominati virus parassiti, perché non si lasciano alle spalle alcun file completo, preservando la funzionalità del programma ospitante.
- Avvio. Un virus di avvio contagia la partizione d'avvio del sistema: va in esecuzione ogni volta che si avvia il sistema e prima che il sistema operativo sia caricato. Ricerca quindi altri dispositivi dotati di boot sector e li infetta. Questi virus sono noti anche come virus della memoria, dato che non compaiono nel file system. La Figura 16.5 illustra il funzionamento di un virus di avvio. I virus di avvio si sono inoltre adattati per infettare il firmware, come gli ambienti pxe della scheda di rete ed efi (*Extensible Firmware Interface*).
- Macro. La maggior parte dei virus è scritta in un linguaggio a basso livello, quale l'assembly o il C. I virus delle macro sono invece creati con un linguaggio ad alto livello, per esempio Visual Basic. Essi sono innescati all'avvio di un programma in grado di eseguire le macro. Un virus delle macro, per esempio, potrebbe essere contenuto in un foglio di calcolo.
- Rootkit. Originariamente coniato per descrivere backdoor pensati per fornire un facile accesso di root su sistemi unix, il termine si è poi esteso a virus e malware che si infiltrano nel sistema operativo stesso. Il risultato è una completa compromissione del sistema operativo, in cui nessun aspetto del sistema può più essere considerato attendibile. Quando il malware infetta il sistema operativo può controllare tutte le funzioni del sistema, comprese quelle che normalmente faciliterebbero il suo rilevamento.
- Codice sorgente. Questo tipo di virus tenta di insediarsi all'interno di codice sorgente, e vi apporta modifiche che ne favoriscono la diffusione.
- Polimorfico. Onde evitare l'azione di rilevamento dei programmi antivirus, questo virus cambia aspetto ogni volta che viene installato. Le modifiche non alterano le sue caratteristiche, ma la sua firma. La firma del virus è una sequenza di informazioni da cui può essere identificato; generalmente si tratta di una serie di byte nel codice del virus.
- Cifrato. I virus cifrati comprendono il codice di decifrazione e il codice cifrato del virus vero e proprio; lo scopo è ancora quello di evitare la rilevazione. Le istruzioni contenute nel virus sono prima decifrare, quindi eseguite dal virus stesso.
- Clandestino (stealth). Questo virus ingannevole tenta di sottrarsi al rilevamento modificando le parti del sistema che potrebbero essere usate per individuarlo. Per esempio, potrebbe modificare la chiamata di sistema `read` in modo da ottenerne, leggendo il file che ha alterato, il codice nella versione originale, anziché il codice infetto.
- Tunnel. Questo virus tenta di sfuggire alla sorveglianza dei programmi antivirus, installandosi nella catena di gestione delle interruzioni. Virus di natura analoga si installano nei driver dei dispositivi.
- Composito. Un virus di questo tipo è in grado di infettare numerose parti di un sistema, tra cui i settori di avviamento, la memoria e i file. Questa circostanza lo rende difficile da scoprire e arginare.
- Corazzato. Un virus corazzato ha un codice particolarmente ostico da capire per i ricercatori che devono scrivere un antivirus. Per evitare di essere rilevato e neutralizzato, può anche presentarsi in forma compressa. Inoltre, i portatori di virus e gli altri file coinvolti nell'attacco virale, sono spesso nascosti tramite l'impostazione degli attributi dei file o la scelta di nomi di file non visibili.

Tale ampia varietà di virus sembra destinata a crescere. Per esempio nel 2004 fu scoperto un virus nuovo e assai esteso, il cui modo di operare sfruttava tre bachi differenti. Questo virus inizialmente infestò centinaia di server (tra cui molti siti ritenuti sicuri), in cui era installato il Microsoft Internet Information Server (iis). Ogni browser Microsoft Explorer vulnerabile era a rischio: non appena uno di loro visitava tali siti scaricava un virus per il browser. Il virus del browser installava poi numerosi programmi back-door, tra i quali un keystroke logger, in grado di registrare tutto ciò che è digitato sulla tastiera (inclusi numeri delle carte di credito e password). Esso, inoltre, installava un primo demone, per consentire agli intrusi l'accesso remoto senza limiti, e un secondo demone, grazie al quale gli intrusi potevano smistare messaggi abusivi di posta elettronica, inviandoli dal calcolatore infetto.

Un dibattito sulla sicurezza piuttosto acceso all'interno della comunità informatica riguarda l'esistenza di una monocultura, in cui molti sistemi eseguono lo stesso hardware, sistema operativo e software applicativo. Questa monocultura sarebbe presumibilmente costituita dai prodotti Microsoft. Una domanda che ci si pone è se una tale monocultura esista ancora oggi; un'altra domanda è se, nel caso, essa sia causa di un aumento delle minacce e dei danni causati da virus e da altre intrusioni che minano la sicurezza. Le informazioni sulle vulnerabilità vengono acquistate e vendute in luoghi come il dark web (sistemi www raggiungibili tramite configurazioni non convenzionali dei client o altri metodi). Un attacco informatico è tanto più forte quanto maggiore è il numero di sistemi influenzati dall'attacco.

16.3 Minacce relative al sistema e alla rete

Le minacce relative a un programma rappresentano di per sé gravi rischi per la sicurezza, ma i rischi crescono notevolmente quando un sistema è connesso a una rete. La connettività su scala mondiale rende un sistema vulnerabile ad attacchi che possono provenire da ovunque.

Più un sistema è aperto, ossia più servizi e funzioni rende disponibili, più è probabile che si possa trovare un baco da sfruttare. I sistemi operativi cercano sempre di più di essere sicuri per default: per esempio Solaris 10 è passato da un modello in cui molti servizi (fra cui ftp e telnet) erano automaticamente abilitati all'installazione del sistema, a un modello in cui quasi tutti i servizi sono inizialmente disabilitati e devono essere abilitati esplicitamente dall'amministratore di sistema. Questi cambiamenti riducono la superficie di attacco, ossia l'insieme dei modi in cui un attaccante può provare a violare il sistema.

Tutti gli hacker lasciano delle tracce, attraverso i percorsi del traffico di rete, a causa di tipologie insolite di pacchetti, o in altro modo. Per questo motivo, gli hacker lanciano frequentemente i loro attacchi da sistemi detti zombie, ovvero sistemi o dispositivi indipendenti che pur essendo stati compromessi dagli hacker continuano a servire i loro proprietari, ignari del fatto che il sistema viene contemporaneamente utilizzato per scopi malevoli, tra cui attacchi denial-of-service e invio di spam. I sistemi zombie rendono gli hacker particolarmente difficili da rintracciare, perché mascherano la fonte originale dell'attacco e l'identità di chi lo compie. Questo è uno dei molti motivi per proteggere i sistemi "irrilevanti", e non solamente quei sistemi che contengono informazioni o servizi "preziosi": i primi possono infatti trasformarsi nelle roccaforti degli hacker.

L'uso diffuso di banda larga e Wi-Fi ha ulteriormente aggravato la difficoltà nel rintracciare gli aggressori: anche un semplice computer desktop, facilmente comprometibile da un malware, può diventare uno strumento prezioso se utilizzato per l'accesso alla rete e per la sua larghezza di banda. Le reti wireless facilitano il lancio di attacchi da parte di utenti malintenzionati che si inseriscono in una rete pubblica in modo anonimo o localizzano una rete privata non protetta ("WarDriving").

16.3.1 Attaccare il traffico di rete

Le reti sono obiettivi abituali e allettanti per gli hacker, che hanno molte opzioni per effettuare i loro attacchi. Come mostrato nella Figura 16.6, un utente malintenzionato può scegliere di rimanere passivo e intercettare il traffico di rete (questo attacco è comunemente indicato come sniffing), ottenendo spesso informazioni utili sui tipi di sessioni condotte tra i sistemi o sul contenuto delle sessioni. In alternativa, il malintenzionato può assumere un ruolo più attivo, mascherandosi come una delle parti (spoofing), o diventando un man-in-the-middle (uomo nel mezzo) completamente attivo, che intercetta ed eventualmente modifica le transazioni tra due soggetti comunicanti.

Nel paragrafo successivo descriviamo un tipo comune di attacco di rete, l'attacco denial-of-service (DoS). Si noti che è possibile proteggersi dagli attacchi con mezzi come la crittografia e l'autenticazione, che sono discussi più avanti in questo capitolo. I protocolli Internet, tuttavia, non supportano per default né la crittografia né l'autenticazione.

16.3.2 Attacchi denial-of-service

Come accennato in precedenza, gli attacchi dos non mirano a ottenere informazioni o a sottrarre risorse, bensì a impedire l'uso corretto di un sistema o di una funzionalità. La maggior parte degli attacchi basati sul rifiuto del servizio è diretta a sistemi che l'aggressore non è riuscito a violare. Infatti, non di rado risulta più facile sferrare un attacco che alteri l'uso corretto di una macchina, anziché penetrarvi all'interno.

Gli attacchi denial-of-service hanno origine, solitamente, dalla rete. Appartengono a due categorie. Nel primo caso l'aggressore occupa un numero così alto di risorse di un servizio da bloccarne completamente la funzionalità. Un click su un sito web, per esempio, potrebbe comportare lo scaricamento di una applet Java, che potrebbe poi consumare tutto il tempo di cpu disponibile, o creare finestre all'infinito. Il secondo caso riguarda il sabotaggio di una rete che ospita un servizio. Numerosi attacchi denial-of-service sono andati a segno con queste modalità, ai danni di grossi siti web. Essi derivano dall'abuso di alcune funzioni fondamentali del protocollo tcp/ip. Se l'aggressore, per esempio, invia la parte del protocollo che afferma "voglio stabilire una connessione tcp", ma a questa non fa mai seguire il consueto messaggio: "la connessione è ora completa", ne potrebbero risultare varie sessioni tcp parzialmente avviate. Se ripetuto per molte sessioni, questo meccanismo può erodere le risorse di rete del sistema, impedendo l'instaurazione di una connessione tcp legittima. A causa di attacchi simili, che possono durare ore o giorni, la funzionalità colpita può rimanere parzialmente o totalmente indisponibile. Gli attacchi di questo genere sono bloccati, di solito, intervenendo a livello della rete, almeno finché i sistemi operativi non siano aggiornati e resi meno vulnerabili.

Generalmente è impossibile impedire gli attacchi denial-of-service, poiché essi sfruttano gli stessi meccanismi del funzionamento normale. Perfino più difficili da prevenire e controbattere sono gli attacchi denial-of-service distribuiti (*distributed denial-of-service*, ddos). Questi attacchi sono scagliati verso un bersaglio comune simultaneamente da numerosi siti, che spesso sono siti zombie. Gli attacchi ddos sono diventati più comuni e spesso sono associati a tentativi di ricatto: un sito viene attaccato, e l'attaccante richiede denaro per fermare l'attacco.

Talvolta l'attacco riesce persino a passare inosservato; può essere difficile stabilire se il rallentamento di un sistema sia dovuto al suo uso intensivo oppure a un attacco. Basti pensare a come una campagna pubblicitaria incisiva che aumenti il traffico verso un sito possa essere facilmente scambiata per un attacco ddos.

Vi sono altri aspetti interessanti in merito agli attacchi dos; tra le altre cose, è necessario, per i programmati e i gestori di sistemi, comprendere in profondità gli algoritmi e le tecnologie che utilizzano. Se un algoritmo di autenticazione, dopo vari tentativi falliti, impedisce l'accesso a un utente per un periodo di tempo, un aggressore potrebbe servirsi di questo meccanismo per bloccare i processi di autenticazione di tutti gli utenti. Analogamente, un firewall che blocca automaticamente alcune tipologie di traffico può essere indotto a bloccare quel traffico quando invece dovrebbe lasciarlo transitare. Infine, le esercitazioni all'interno dei corsi universitari di informatica sono vere e proprie fucine di attacchi dos involontari. Si pensi alle prime esercitazioni di programmazione

in cui gli studenti imparano a creare processi figli o thread. Un baco tra i più comuni riguarda la proliferazione a oltranza dei processi figli: per la memoria libera del sistema e la cpu non c'è via di scampo.

16.3.3 Scansione delle porte

La scansione delle porte non è un attacco, ma piuttosto un mezzo impiegato dai pirati informatici per sondare le vulnerabilità di un sistema, in vista di un attacco. Anche gli addetti alla sicurezza utilizzano la scansione delle porte, per esempio per rilevare i servizi che non sono necessari o che non dovrebbero essere in esecuzione. La scansione delle porte funziona in automatico, grazie a un dispositivo che tenta di stabilire una connessione tcp/ip o di inviare un pacchetto udp verso una porta specifica (o una serie di porte).

La scansione delle porte è spesso parte di una tecnica di ricognizione nota come fingerprinting, in cui un utente malintenzionato tenta di dedurre il tipo di sistema operativo in uso e l'insieme dei servizi offerti al fine di individuare le vulnerabilità note. Molti server e client semplificano questo compito rivelando il loro numero esatto di versione come parte delle intestazioni del protocollo di rete (per esempio, le intestazioni http "Server:" e "User-Agent:"). Anche un'analisi dettagliata di comportamenti peculiari da parte dei gestori di protocollo può aiutare l'aggressore a capire quale sistema operativo è in uso sull'obiettivo, un passaggio necessario per un attacco di successo.

Gli scanner di vulnerabilità di rete sono venduti come prodotti commerciali. Esistono anche strumenti che eseguono un sottoinsieme delle funzionalità di uno scanner completo. Per esempio, l'applicazione open-source `nmap` (si veda <http://www.insecure.org/nmap/>), permette di esplorare le reti e tracciare rapporti sulla sicurezza con grande versatilità. Una volta puntata su un obiettivo, essa rivela quali servizi sono in esecuzione, compresi i nomi delle applicazioni e le versioni, e può rivelare quale sistema operativo sia in uso. L'applicazione permette, inoltre, di riconoscere le tecniche di difesa adottate: per esempio, quali firewall siano posti a difesa del bersaglio. Non sfrutta alcun baco noto. Tuttavia, altri strumenti (come Metasploit) iniziano la loro azione dove termina quella degli scanner delle porte, fornendo strutture di payload che possono essere utilizzate per testare le vulnerabilità o sfruttarle creando un payload specifico in grado di innescare il bug.

L'articolo fondamentale sulle tecniche di scansione delle porte può essere trovato all'indirizzo <http://phrack.org/issues/49/15.html>. Queste tecniche sono in continua evoluzione, così come le misure per rilevarle (che costituiscono la base per i sistemi di rilevamento delle intrusioni di rete, discussi più avanti).

16.4 Crittografia come strumento per la sicurezza

Le difese contro gli attacchi informatici sono varie, e spaziano da quelle metodologiche alle più tecnologiche. La crittografia è lo strumento più generale di cui progettisti di sistema e utenti possono disporre. In questo paragrafo discuteremo della crittografia e le sue applicazioni nella sicurezza dei calcolatori. Si noti che la nostra trattazione della crittografia è semplificata per scopi didattici. Evitate di usare direttamente gli schemi qui descritti in situazioni pratiche. Esistono ottime librerie per la crittografia e queste librerie, facilmente reperibili, costituiscono una buona base per il progetto di applicazioni.

In un calcolatore isolato il sistema operativo può determinare con sicurezza chi è il mittente e chi il destinatario di qualsiasi comunicazione tra processi, poiché esso controlla tutti i canali di comunicazione all'interno del calcolatore. In una rete di calcolatori la situazione è completamente diversa. Un calcolatore in una rete riceve bit dai *cavi*, senza aver alcun modo immediato e affidabile per determinare quale calcolatore o applicazione abbia inviato quei bit. In modo analogo, un calcolatore invia bit nella rete senza sapere con certezza chi effettivamente li riceverà. Inoltre, sia in fase di invio che in fase di ricezione, il sistema non ha modo di sapere se qualcuno sta ascoltando di nascosto la comunicazione.

Di solito, per risalire ai potenziali mittenti e destinatari dei messaggi si usano gli indirizzi di rete. I pacchetti di rete arrivano con un indirizzo di sorgente, come un indirizzo ip, e quando un calcolatore invia un messaggio specifica esplicitamente il destinatario nell'indirizzo di destinazione. Tuttavia, per le applicazioni per le quali la sicurezza è importante, non è possibile fidarsi del fatto che gli indirizzi di sorgente e destinazione di un pacchetto identifichino con certezza chi lo ha inviato o chi lo riceve. Un calcolatore in mano a un pirata informatico potrebbe inviare un messaggio con un indirizzo sorgente falsificato, e molti altri calcolatori, oltre a quello specificato nell'indirizzo di destinazione, potrebbero ricevere il pacchetto (cosa che di solito effettivamente avviene). Per esempio, tutti i router nel cammino tra sorgente e destinazione riceveranno il pacchetto. Dunque, come può il sistema operativo decidere se soddisfare una richiesta se non può considerare fidata l'identità del richiedente? E come può fornire la protezione per una richiesta o per dei dati se non è in grado di determinare con esattezza chi riceverà la risposta o il contenuto del messaggio che invia nella rete?

In generale, si considera impossibile costruire una rete di qualsiasi dimensione in cui gli indirizzi di sorgente e di destinazione dei pacchetti si possano considerare *fidati* in questo senso. Quindi, l'unica possibilità è cercare di eliminare in qualche modo la necessità di considerare come fidata la rete. Per questo è essenziale il ruolo della crittografia. Da un punto di vista astratto, la crittografia si usa per definire i potenziali mittenti e destinatari di un messaggio. La crittografia moderna si fonda su codici segreti, chiamati chiavi, che si distribuiscono selettivamente ai calcolatori di una rete e si usano per elaborare i messaggi. La crittografia permette al destinatario di un messaggio di verificare che il messaggio sia stato creato da un calcolatore che possiede una certa chiave. In modo analogo, un processo mittente può codificare il suo messaggio in modo tale che solo un calcolatore in possesso di una certa chiave possa decifrare il messaggio. Tuttavia, a differenza degli indirizzi di rete, le chiavi sono progettate in modo che sia computazionalmente impossibile derivarle a partire dai messaggi generati tramite esse e da qualsiasi altra informazione pubblica. Dunque, questo meccanismo costituisce un mezzo molto più fidato per definire i mittenti e i destinatari dei messaggi.

La crittografia è uno strumento potente e il suo utilizzo genera talvolta controversie. Alcuni paesi ne vietano l'uso in determinate forme o limitano la lunghezza delle chiavi, in altri paesi sono in corso dibattiti sul fatto che i venditori di tecnologia (per esempio i venditori di smartphone) debbano fornire una backdoor alla crittografia utilizzata, consentendo alle forze dell'ordine di aggirare la privacy che i dispositivi forniscono. Molti osservatori sostengono, tuttavia, che le backdoor costituiscano un'intenzionale indebolimento della sicurezza che potrebbe essere sfruttato dagli aggressori o usato abusivamente dai governi.

Si noti infine che la crittografia è un ambito di studio a sé stante, caratterizzato da grandi e piccole complessità e sottigliezze. Ciò che esploriamo qui sono gli aspetti principali di quelle parti della crittografia che riguardano i sistemi operativi.

16.4.1 Cifratura

Poiché risolve un'ampia varietà di problemi di sicurezza delle comunicazioni, la cifratura (*encryption*) è frequentemente utilizzata in molti ambiti della moderna computazione. La cifratura è utilizzata per inviare messaggi sulla rete in maniera sicura e per proteggere da letture non autorizzate i dati contenuti in un database, in un file o in un intero disco. Un algoritmo di cifratura permette al mittente di un messaggio di imporre che solo un calcolatore che possiede una certa chiave possa leggere il messaggio. Un algoritmo di cifratura permette anche di assicurare che un dato possa essere letto solo da chi l'ha scritto. La cifratura dei messaggi, come si sa, è una pratica antica; alcuni algoritmi di cifratura risalgono all'antichità. In questo paragrafo ci occuperemo degli algoritmi e dei principi fondamentali dell'era moderna.

Un algoritmo di cifratura comprende i seguenti componenti.

- Un insieme K di chiavi.
- Un insieme M di messaggi.
- Un insieme C di testi cifrati.
- Una funzione di cifratura $E : K \rightarrow (M \rightarrow C)$. Cioè, per ogni $k \in K$, E_k è una funzione che genera testi cifrati a partire dai messaggi. Sia E sia E_k devono essere funzioni calcolabili in modo efficiente per ogni k . In genere, E_k mappa in maniera randomizzata i messaggi in testi cifrati.
- Una funzione di decifratura $D : K \rightarrow (C \rightarrow M)$. Cioè, per ogni $k \in K$, D_k è una funzione che genera messaggi a partire dai testi cifrati. Sia D sia D_k devono essere funzioni calcolabili in modo efficiente per ogni k .

La proprietà fondamentale che un algoritmo di autenticazione deve possedere è la seguente: dato un testo cifrato $c \in C$, un calcolatore può calcolare m tale che $E_k(m) = c$ solo se m è in possesso di k . Quindi, un calcolatore che ha k può decifrare i testi cifrati ottenendo i testi in chiaro corrispondenti. Tuttavia, un calcolatore che non possiede k non può decifrarli. Poiché i testi cifrati sono di solito visibili (per esempio, s'inviano in una rete), è essenziale che sia infatti possibile derivare k dai testi cifrati.

Ci sono due tipi principali di algoritmi di cifratura: simmetrica e asimmetrica. Nei paragrafi successivi ci occuperemo sia dell'una sia dell'altra.

16.4.1.1 Cifratura simmetrica

In un algoritmo di cifratura simmetrica la stessa chiave è usata per cifrare e decifrare; ciò significa che la segretezza di k deve essere protetta. La Figura 16.7 mostra un esempio di due utenti che comunicano in maniera sicura su un canale insicuro per mezzo di una cifratura simmetrica. Si noti che lo scambio della chiave può avvenire direttamente tra le due parti oppure per mezzo di una terza parte fidata (un'autorità di certificazione), come illustrato nel Paragrafo 16.4.1.4.

Negli ultimi decenni, l'algoritmo di cifratura simmetrica più usato negli Stati Uniti d'America per scopi civili è stato il des (*data encryption standard*), adottato dal National Institute of Standards and Technology (nist). Il des funziona prendendo blocchi in chiaro di 64 bit e una chiave, della lunghezza di 56 bit, ed effettuando una serie di trasformazioni basate su sostituzioni e permutazioni. Poiché agisce su un blocco di bit per volta, l'algoritmo des è un codice a blocchi e le sue trasformazioni sono quelle tipiche della cifratura a blocchi. Nella cifratura a blocchi, l'utilizzo di una sola chiave per cifrare una quantità eccessiva di dati presterebbe il fianco a eventuali attacchi.

Il des è ormai considerato insicuro per molte applicazioni, perché le sue chiavi possono essere provate esaustivamente anche con risorse di calcolo modeste. (Si noti che, nonostante ciò, des è ancora molto utilizzato). Anziché abbandonarlo, tuttavia, il nist ha elaborato una variante del des, chiamata des triplo, in cui l'algoritmo originale è ripetuto per tre volte (due cifrature e una decifrazione) sul medesimo testo in chiaro, usando due o tre chiavi – per esempio, $c = E_{k3}(D_{k2}(E_{k1}(m)))$. Quando si usano tre chiavi, la lunghezza effettiva della chiave è di 168 bit. Il des triplo è attualmente molto usato.

Per sostituire il des, nel 2001 il nist ha adottato un nuovo algoritmo di cifratura, detto aes (*advanced encryption standard*, ossia “standard di cifratura avanzata”). Laes (noto anche come Rijndael) è stato standardizzato in FIPS-197 (<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>). Utilizza chiavi con lunghezza pari a 128, 192 o 256 bit e opera su blocchi di 128 bit. In genere, è un algoritmo compatto ed efficiente.

La cifratura a blocchi non è di per sé uno schema di cifratura sicuro. In particolare, essa non gestisce direttamente messaggi che superano le dimensioni di blocco richieste. Vi sono tuttavia molti metodi di cifratura basati su codici stream che possono essere utilizzati per crittografare messaggi più lunghi in maniera sicura.

I codici stream sono metodi di cifratura volti a crittografare e decrittografare non già un blocco, ma una sequenza di byte o di bit; ciò è utile quando i messaggi in chiaro sono talmente lunghi da rendere la cifratura a blocchi troppo lenta. La chiave è immessa in un generatore di bit pseudo-casuali, ossia un algoritmo che tenta di produrre sequenze casuali di bit. Ciò che restituisce il generatore, quando riceve una chiave, è un insieme infinito di bit, detto keystream, che può essere utilizzato per la codifica di un testo semplicemente calcolando lo xor (lo xor, o or esclusivo, è un'operazione che confronta due bit in ingresso e restituisce 0 se i bit sono uguali, 1 se sono diversi) tra il testo e il keystream. Le suite di crittografia basate su aes includono anche codici stream e sono le più comuni oggi.

16.4.1.2 Cifratura asimmetrica

In un algoritmo di cifratura asimmetrica vi sono chiavi diverse per la cifratura e la decifrazione. Un'entità che desidera ricevere una comunicazione cifrata crea due chiavi e ne mette una (chiamata chiave pubblica) a disposizione di chiunque lo voglia. Ciascun mittente può utilizzare la chiave per cifrare la comunicazione, ma solo il creatore della chiave è in grado di decifrarla. Questo schema, noto come crittografia a chiave pubblica, ha costituito una svolta nella crittografia (descritta da Diffie e Hellman in <https://www-ee.stanford.edu/hellman/publications/24.pdf>). Non è più necessario che la chiave venga tenuta segreta e consegnata in modo sicuro, ma chiunque può inviare un messaggio cifrato al soggetto ricevente e non importa chi altro è in ascolto, perché solo il ricevente può decifrare il messaggio.

Come esempio di crittografia a chiave pubblica descriviamo l'algoritmo noto come rsa, dal nome dei suoi inventori (Rivest, Shamir e Adleman). rsa è l'algoritmo asimmetrico più ampiamente usato. Negli ultimi tempi, tuttavia, registrano crescente diffusione gli algoritmi asimmetrici basati sulle curve ellittiche, che mantengono lo stesso livello di efficacia crittografica con chiavi più brevi.

In rsa, k_e è la chiave pubblica e k_d è la chiave privata. N è il prodotto di due numeri primi grandi, scelti casualmente, p e q (p e q per esempio misurano ognuno 512 bit). Deve essere computazionalmente impossibile ricavare $k_{d,N}$ da $k_{e,N}$, affinché k_e non debba essere tenuto segreto e possa essere divulgato. L'algoritmo di cifratura è $E_{k_e,N}(m) = m^{k_e} \text{ mod } N$, dove k_e soddisfa $k_e k_d \text{ mod } (p-1)(q-1) = 1$. L'algoritmo di decifrazione è allora $D_{k_d,N}(c) = c^{k_d} \text{ mod } N$.

La Figura 16.8 rappresenta un esempio che fa uso di valori piccoli. Si suppone $p = 7$ e $q = 13$, e si calcola $N = 7*13 = 91$ e $(p-1)(q-1) = 72$. Si sceglie quindi un k_e strettamente minore di 72 tale che 72 e k_e siano relativamente primi, per esempio, $k_e = 5$. Si calcola poi k_d tale che valga $k_e k_d \text{ mod } 72 = 1$, ottenendo $k_d = 29$. Si ottiene così la chiave pubblica, $k_{e,N} = 5,91$, e la chiave privata, $k_{d,N} = 29,91$. La cifratura del messaggio 69 con la chiave pubblica genera il messaggio 62, che viene quindi decodificato dal destinatario grazie alla chiave privata.

L'uso della cifratura asimmetrica ha inizio con la pubblicazione della chiave pubblica del destinatario. Se la comunicazione è a due vie, anche il mittente deve pubblicare la propria chiave pubblica. Per “pubblicazione” si può intendere la semplice trasmissione di una copia elettronica della chiave, oppure qualcosa di più complesso. La chiave privata (detta anche “segreta”) deve essere custodita con estrema cura, dato che chiunque ne entri in possesso può decrittografare ogni messaggio creato con la chiave pubblica corrispondente.

La differenza nell'uso delle chiavi, tra crittografia asimmetrica e simmetrica, a prima vista trascurabile, è in realtà piuttosto netta. La crittografia asimmetrica esige risorse computazionali di gran lunga superiori per la propria esecuzione. Un elaboratore può codificare e decodificare un testo molto più velocemente mediante gli usuali algoritmi simmetrici, che non usando un algoritmo asimmetrico. Perché, dunque, usare un algoritmo asimmetrico? In realtà, questi algoritmi non sono affatto applicati alla cifratura di grandi quantità di dati per scopi generali. Essi sono invece usati sia per la cifratura di piccole quantità d'informazione, sia per l'autenticazione, la sicurezza e la distribuzione delle chiavi, come vedremo nei paragrafi seguenti.

16.4.1.3 Autenticazione

Abbiamo visto che la cifratura permette di limitare l'insieme dei potenziali destinatari di un messaggio. Garantire l'insieme dei potenziali mittenti di un messaggio è un processo che si chiama anche autenticazione. L'autenticazione è quindi complementare alla cifratura. L'autenticazione è anche utile per attestare che un messaggio non sia stato modificato. In questo paragrafo illustriamo l'autenticazione come filtro dei possibili mittenti dei messaggi. Si tratta di una nozione simile, ma non identica, all'autenticazione degli utenti trattata nel Paragrafo 16.5.

Un algoritmo di autenticazione che utilizza chiavi simmetriche comprende i seguenti componenti.

- Un insieme K di chiavi.
- Un insieme M di messaggi.
- Un insieme A di autenticatori.
- Una funzione $S : K \rightarrow (M \rightarrow A)$. Cioè, per ogni $k \in K$, S_k è una funzione che genera autenticatori a partire da messaggi. Sia S sia S_k devono essere funzioni calcolabili in modo efficiente per ogni k .
- Una funzione $V : K \rightarrow (M \times A \rightarrow \{\text{true}, \text{false}\})$. Cioè, per ogni $k \in K$, V_k è una funzione che verifica gli autenticatori sui messaggi. Sia V che V_k devono essere funzioni calcolabili in modo efficiente per ogni k .

La proprietà fondamentale che un algoritmo di autenticazione deve possedere è la seguente: dato un messaggio m , un calcolatore può generare un autenticatore $a \in A$ tale che $V_k(m, a) = \text{true}$ soltanto se esso possiede k . Quindi, un calcolatore che possiede k può generare autenticatori su messaggi in modo che tutti gli altri calcolatori che possiedono k li possano verificare. Tuttavia, un calcolatore che non ha k non può generare autenticatori su messaggi che possono essere verificati tramite V_k . Poiché gli autenticatori sono generalmente visibili (per esempio, s'inviano nella rete con i messaggi stessi), k non si deve poter derivare a partire dagli autenticatori. In pratica, se $V_k(m, a) = \text{true}$, allora sappiamo che m non è stato modificato e che il mittente del messaggio possiede k . Se condividiamo k con una sola entità, allora sappiamo che il messaggio arriva da k .

Così come esistono due tipi di algoritmi di cifratura, vi sono due varietà principali di algoritmi di autenticazione. Il primo passo per capire tali algoritmi è la comprensione delle funzioni hash. La funzione hash $H(m)$ crea un piccolo blocco di dati di dimensioni prefissate, noto come valore hash o digest del messaggio, da un messaggio m . La maniera in cui procedono le funzioni hash è di dividere il messaggio in segmenti, ed elaborare i segmenti in modo da produrre un valore hash di n bit. La funzione hash H deve essere a prova di collisione: cioè, dato m , il calcolo di un $m' \neq m$ tale che $H(m) = H(m')$ deve risultare impraticabile. Da ciò segue che $H(m) = H(m')$ comporta che $m = m'$, ossia, che il messaggio non è stato modificato. Alcune delle funzioni hash più comuni sono md5, oggi considerata insicura, che produce valori hash da 128 bit, e sha-1, che fornisce valori da 160 bit. Le funzioni hash sono utili per rilevare modifiche dei messaggi, ma non bastano ad autenticarli: $H(m)$ può certo essere spedito insieme al messaggio, ma se H è una funzione nota, chiunque può modificare il messaggio m in m' , computare $H(m')$, e produrre un messaggio alterato non rilevabile. È per questo che occorre autenticare $H(m)$.

Il primo tipo di algoritmo d'autenticazione sfrutta la cifratura simmetrica. Nei codici per l'autenticazione dei messaggi (*message-authentication code*, mac), una somma di controllo crittografica è generata dal messaggio per mezzo della chiave segreta. Un codice mac fornisce un modo per autenticare in modo sicuro piccoli valori. Se lo usiamo per autenticare $H(m)$, per un H a prova di collisione, allora si ottiene un modo per autenticare in modo sicuro messaggi lunghi utilizzando la funzione hash.

Il secondo tipo di algoritmo di autenticazione è un algoritmo basato sulla firma digitale e gli autenticatori prodotti in questo caso sono chiamati *firme digitali*. Le firme digitali sono molto utili, perché permettono a *chiunque* di verificare l'autenticità del messaggio. In un algoritmo basato sulla firma digitale è praticamente impossibile dal punto di vista computazionale derivare k_s da k_v . Per questa ragione k_v è la chiave pubblica e k_s è la chiave privata.

Si consideri per esempio l'algoritmo rsa per la firma digitale. È simile all'algoritmo crittografico rsa, ma l'uso delle chiavi è invertito. La firma digitale di un messaggio si ottiene calcolando $S_{k_s}(m) = H(m)^{k_s} \bmod N$. La chiave k_s è di nuovo una coppia (d, N) , dove N è il prodotto di due primi grandi p e q scelti a caso. L'algoritmo di verifica è allora $V_{k_v}(m, a) = ? (a^{k_v} \bmod N = H(m))$, dove k_v soddisfa $k_v k_s \bmod (p - 1)(q - 1) = 1$.

Noteate che crittografia e autenticazione possono essere usate insieme o separatamente. A volte ci serve autenticazione ma non riservatezza. Per esempio, un'azienda potrebbe fornire una patch software e "firmare" quella patch per dimostrare che arriva da quell'azienda e che non è stata modificata.

L'autenticazione è parte integrante di molti aspetti della sicurezza. La firma digitale, per esempio, è uno strumento per realizzare il non ripudio (*nonrepudiation*), ossia per dimostrare che un soggetto abbia compiuto una certa azione. Un esempio tipico si ha quando il soggetto compila dei moduli elettronici invece di firmare contratti cartacei: il non ripudio impedisce che il soggetto neghi di aver compilato i moduli.

16.4.1.4 Distribuzione delle chiavi

Senza dubbio una buona parte della battaglia tra crittografi (coloro che inventano i codici) e i crittoanalisti (coloro che tentano di decifrarli) è incentrata sulla questione delle chiavi. Con gli algoritmi simmetrici, entrambi i soggetti coinvolti nella comunicazione devono possedere la chiave, ma essa non deve essere nota ad alcuno. La consegna delle chiavi simmetriche comporta una sfida impegnativa. Talvolta si ricorre a una comunicazione non in linea (*out-of-band*), tramite documenti cartacei o una conversazione. Questi metodi, tuttavia, non si prestano a essere generalizzati. Anche la questione di come gestire le chiavi va presa in considerazione. Si supponga che un utente desideri comunicare in privato con N altri utenti; questo utente dovrebbe avere N chiavi e, per maggior sicurezza, si vedrebbe costretto a cambiarle spesso.

L'analisi di tali situazioni è proprio ciò che ha dato impulso alla creazione degli algoritmi a chiave asimmetrica. Ogni utente non solo può così scambiarsi le chiavi in pubblico, ma ha bisogno di possedere una sola chiave privata, indipendentemente dal numero di persone coinvolte nella comunicazione. Rimane il problema della gestione di tante chiavi pubbliche quanti sono i destinatari da contattare; ma, poiché non è necessario mantenere segrete le chiavi pubbliche, si può adoperare la memorizzazione ordinaria per ciascun mazzo di chiavi (*key ring*).

Purtroppo, anche la distribuzione delle chiavi pubbliche richiede determinate accortezze. Si osservi l'attacco di interposizione illustrato nella Figura 16.9. In questo caso, la persona che intende ricevere un messaggio codificato distribuisce la sua chiave pubblica, ma l'intruso, a sua volta, trasmette la propria chiave pubblica illegittima (che fa coppia con la sua chiave privata). La persona che

vuole inviare il messaggio in codice non ha il minimo sospetto sulla provenienza della chiave, dunque la utilizza per criptare il messaggio; l'intruso, a questo punto, può decrittografarlo tranquillamente.

Il problema esposto concerne l'autenticazione: è necessaria una prova su chi (o che cosa) detenga una chiave pubblica. Una soluzione applicabile prevede l'uso dei certificati digitali. Un certificato digitale è una chiave pubblica firmata digitalmente da un soggetto fidato. Questi riceve un attestato d'identità da una terza parte e certifica che la chiave pubblica appartiene alla parte. Ma da quali garanzie è sorretta la credibilità del soggetto certificatore? Le autorità di certificazione includono le proprie chiavi pubbliche nei browser web (e in altre applicazioni che utilizzano i certificati) prima della loro distribuzione. Le medesime autorità, quindi, possono attestare che altre autorità sono affidabili (firmando digitalmente le chiavi pubbliche di queste ultime); a loro volta, le autorità certificate proseguono in questo processo, creando una rete di fiducia. I certificati possono essere distribuiti nel formato digitale standard X.509, che il calcolatore è in grado di analizzare. Questo schema è usato per la comunicazione sicura sul Web, come si vedrà nel Paragrafo 16.4.3.

16.4.2 Implementazione della crittografia

I protocolli di rete sono di solito organizzati in strati, come una cipolla, in cui ciascuno strato agisce come un client rispetto allo strato sottostante. Cioè, quando un protocollo genera un messaggio per il protocollo di pari livello nel calcolatore di destinazione, consegna il suo messaggio al protocollo sottostante nello stack dei protocolli di rete in modo che questo lo consegna al rispettivo protocollo in quella macchina. Per esempio, in una rete ip, il tcp (un protocollo dello *strato di trasporto*) agisce come un client dell'ip (un protocollo dello *strato di rete*): i pacchetti del tcp sono passati all'ip, lo strato sottostante, affinché siano consegnati all'entità ip all'altro estremo della connessione. Ip incapsula il pacchetto tcp in un pacchetto ip che, in modo analogo, li passa allo *strato di collegamento dei dati* sottostante affinché siano trasmessi attraverso la rete al corrispondente livello ip nel calcolatore di destinazione. Questo livello ip a sua volta consegnerà i pacchetti tcp a livello tcp su quella macchina. Il modello osi, menzionato in precedenza e descritto in dettaglio nel Paragrafo 19.3.2, comprende sette livelli di questo tipo.

La crittografia si può includere quasi in ogni livello del modello osi. Il protocollo tls (Paragrafo 16.4.3) per esempio fornisce sicurezza a livello di trasporto. La sicurezza a livello di rete, definita nello standard ipsec, specifica formati dei pacchetti ip che permettono l'inserimento di autenticatori e la cifratura del contenuto dei pacchetti. Ipsec utilizza la cifratura simmetrica e sfrutta il protocollo Internet Key Exchange (ike) per lo scambio di chiavi. Ike si basa sulla crittografia a chiave pubblica. L'ipsec si sta diffondendo come base per la realizzazione di reti private virtuali (*virtual private network*, vpn), nelle quali tutto il traffico fra due punti terminali ipsec è codificato in modo da rendere privata una rete che sarebbe altrimenti pubblicamente accessibile. Sono stati anche sviluppati numerosi protocolli utilizzabili dalle applicazioni (per esempio pgp, per la cifratura di e-mail), ma poi le stesse applicazioni devono realizzare il codice per implementare la sicurezza.

Ci si può chiedere quale sia la miglior collocazione della protezione critografica all'interno dello stack dei protocolli. Non vi è una risposta univoca a questa domanda. Da una parte, se si colloca la protezione negli strati bassi dello stack, può beneficiarne un maggior numero di protocolli. Per esempio, poiché i pacchetti ip incapsulano i pacchetti tcp, la cifratura dei pacchetti ip (per esempio con l'ipsec) nasconde anche il contenuto del pacchetto tcp incapsulato. Analogamente, gli autenticatori su pacchetti ip rilevano le modifiche delle informazioni nelle intestazioni tcp contenute.

D'altra parte, la protezione collocata negli strati bassi dello stack dei protocolli potrebbe non essere sufficiente ai protocolli dei livelli più alti. Per esempio, un server applicativo che accetta connessioni cifrate con ipsec potrebbe autenticare i calcolatori client dai quali riceve le richieste. Tuttavia, per autenticare un utente in uno specifico calcolatore, si deve poter usare un protocollo del livello applicativo (che include, per esempio, l'inserimento della password da parte dell'utente). Si consideri inoltre il problema della posta elettronica: prima che i messaggi spediti tramite il protocollo standard smtp siano recapitati al destinatario, sono memorizzati e inoltrati da macchine intermedie, spesso più volte. Ognuno di questi passaggi può coinvolgere reti sicure o non sicure. Affinché la posta elettronica sia resa sicura, quindi, i messaggi devono essere crittografati in modo da renderli sicuri indipendentemente dal mezzo che li trasporta.

Sfortunatamente, come molti strumenti, la crittografia può essere utilizzata non solo con "buoni propositi", ma anche con "cattive intenzioni". Gli attacchi ransomware descritti in precedenza, per esempio, si basano sulla crittografia. Come accennato, gli hacker cifrano le informazioni sul sistema di destinazione e le rendono inaccessibili al proprietario, per poi costringerlo a pagare un riscatto per ottenere la chiave necessaria per decodificare i dati. La prevenzione di tali attacchi passa attraverso una migliore sicurezza di rete e di sistema e un buon piano di backup che permette di ripristinare il contenuto dei file senza la chiave.

16.4.3 Un esempio: TLS

Il tls (*Transport Layer Security*) è un protocollo di crittografia che consente a due computer di comunicare in modo sicuro, assicurando che mittente e destinatario siano gli unici ad aver accesso ai messaggi scambiati. Tls è probabilmente il protocollo di crittografia più comunemente utilizzato oggi su Internet, poiché è il protocollo standard utilizzato dai browser Web per comunicare in modo sicuro con i server web. Per completezza, va osservato che tls è un'evoluzione di ssl (Secure Sockets Layer), progettato da Netscape. Il tls è descritto in dettaglio in <https://tools.ietf.org/html/rfc5246>.

Il tls è un protocollo complesso con molte opzioni e in questo testo si presenta una sola delle sue varianti, e solo in una forma astratta e molto semplificata, in modo tale da focalizzarci principalmente sull'uso delle sue primitive crittografiche. Ciò che ci apprestiamo a studiare è un'interazione complessa fra client e server, in cui la crittografia asimmetrica serve a produrre una chiave della sessione utilizzabile per crittografare simmetricamente la comunicazione fra i due attori della sessione – il tutto evitando gli attacchi replay e di interposizione. Per migliorare l'efficacia della crittografia, la chiave della sessione è eliminata al termine della sessione cui si applica: ulteriori comunicazioni fra client e server richiedono la generazione di una nuova chiave.

Il protocollo tls è avviato da un client allo scopo di comunicare in modo sicuro con un server. Prima dell'avvio del protocollo, s'ipotizza che il server *s* abbia ottenuto un certificato, denotato con *cert_s*, da un'autorità di certificazione *ca*. Questo certificato è una struttura dati che contiene i seguenti elementi:

- vari attributi (*attrs*) del server, come il *nome unico che lo contraddistingue* e il suo *nome comune (dns)*;
- l'identità di un algoritmo asimmetrico di cifratura *E()* per il server;

- la chiave pubblica k_e di questo server;
- un intervallo di validità (interval) durante il quale il certificato si può considerare valido;
- una firma digitale a sulle informazioni di cui sopra effettuata da parte della ca , cioè, $a = s_{kca}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle)$.

Prima dell'attivazione del protocollo, si presume che il client abbia ottenuto l'algoritmo pubblico di verifica V_{kca} per la specifica ca . Nel caso del Web, il browser impiegato dall'utente viene consegnato dal relativo produttore insieme con gli algoritmi di verifica e le chiavi pubbliche di alcune autorità di certificazione. L'utente può aggiungere o cancellare a proprio piacimento queste informazioni.

Quando un client si connette a un server, gli invia un valore casuale n_c di 28 byte. Il server s risponde inviando al client un proprio valore casuale n_s oltre al suo certificato cert_s . Il client verifica che $V_{kca}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle, a) = \text{true}$ e che il momento attuale sia compreso nell'intervallo di validità interval del certificato. Se entrambe le condizioni sono verificate, il server ha attestato la propria identità. Il client può quindi generare un premaster secret pms di 46 byte, e trasmettere $\text{cpms} = E_{ks}(\text{pms})$ al server. Esso ottiene in chiaro $\text{pms} = D_{kd}(\text{cpms})$. Entrambe le parti, adesso, sono in possesso di n_c, n_s e pms , e possono computare il master secret condiviso di 48 byte tramite $\text{ms} = H(n_c, n_s, \text{pms})$. Solo il client e il server possono calcolare ms , giacché solo loro conoscono pms . La dipendenza di ms da n_c e n_s , inoltre, garantisce che esso sia un nuovo valore, non usato per una precedente comunicazione. Il client e il server, a questo punto, computano entrambi le seguenti chiavi a partire da ms :

- una chiave di cifratura simmetrica k_{cs}^{crypt} per cifrare i messaggi dal client al server;
- una chiave di cifratura simmetrica k_{sc}^{crypt} per cifrare i messaggi dal server al client;
- una chiave di generazione di mac k_{cs}^{mac} per generare autenticatori sui messaggi dal client al server;
- una chiave di generazione di mac k_{sc}^{mac} per generare autenticatori sui messaggi dal server al client.
- Per inviare un messaggio m al server, il client invia
- $c = E_{kscrypt}(\langle m, S_{kscmac}(m) \rangle)$.

Al ricevimento di c , il server ricostruisce

$$\bullet \langle m, a \rangle = D_{kscrypt}(c)$$

e accetta m se $V_{kscmac}(m, a) = \text{true}$. Analogamente, per inviare un messaggio m al client, il server invia

$$\bullet c = E_{kscrypt}(\langle m, S_{kscmac}(m) \rangle).$$

e il client ricostruisce

$$\bullet \langle m, a \rangle = D_{kscrypt}(c)$$

e accetta m se $V_{kscmac}(m, a) = \text{true}$.

Questo protocollo permette al server di limitare i riceventi dei suoi messaggi al client che ha generato pms , e i mittenti dei messaggi che esso accetta a quello stesso client. Analogamente, il client può limitare i destinatari dei messaggi che invia e il mittente dei messaggi che accetta alla parte che conosce k_d (cioè, la parte capace di decifrare cpms). In molte applicazioni, come le transazioni che si svolgono nel Web, il client deve verificare l'identità della parte che conosce k_d . Questo è uno degli scopi del certificato cert_s ; in particolare, il campo attrs contiene informazioni che il client può usare per determinare l'identità (per esempio, il nome del dominio) del server con il quale sta comunicando. Per applicazioni nelle quali anche il server deve avere informazioni sul client, il tls ha un'opzione tramite la quale un client può inviare un certificato al server.

Il protocollo tls trova vasta applicazione anche al di fuori di Internet: le reti tls vpn per esempio, sono oggi un concorrente di ipsec vpn. Mentre quest'ultimo è competitivo per la cifratura del traffico punto a punto, per esempio tra due divisioni di una società, le reti tls vpn sono più flessibili ma meno efficienti. Per questo motivo, potrebbero essere scelte per la comunicazione fra un singolo impiegato e la sede della sua azienda.

16.5 Autenticazione degli utenti

Fin qui il tema dell'autenticazione ha riguardato messaggi e sessioni di lavoro, ma non gli utenti. Se un sistema non può autenticare un utente, che valore ha l'autenticazione del messaggio proveniente da quell'utente? Per i sistemi operativi un fondamentale problema di sicurezza riguarda quindi l'autenticazione dell'utente. Il sistema di protezione dipende dalla capacità d'identificare i programmi e i processi correntemente in esecuzione, che a sua volta è basata sulla capacità di identificare ogni utente del sistema. Normalmente un utente identifica se stesso, quindi si tratta di stabilire se l'identità di un utente è autentica. Generalmente l'autenticazione è basata su uno o più dei seguenti tre elementi: oggetti posseduti dall'utente (una chiave o una scheda); conoscenze dell'utente (un identificatore utente e una password); un attributo fisico dell'utente (impronta digitale, impronta della retina o firma).

16.5.1 Password

Il metodo più diffuso per identificare un utente è quello che prevede l'uso di password. Quando un utente s'identifica attraverso il proprio user id o account name, riceve la richiesta d'immissione di una password. Se la password immessa dall'utente corrisponde a quella memorizzata nel sistema, il sistema presume che l'utente sia legittimo.

Spesso le password si usano per proteggere oggetti del calcolatore quando non esistono schemi di protezione più completi. Le password si possono considerare un caso particolare di chiavi o di abilitazioni. Per esempio, si potrebbe associare una password a ogni risorsa, come un file; in questo modo ogni volta che si chiede l'uso della risorsa, si deve fornire la relativa password; se questa è giusta, si permette l'accesso. A diritti d'accesso diversi si possono associare password diverse. Si possono per esempio impiegare password diverse per ciascuna delle seguenti operazioni su file: lettura, aggiunta di dati e aggiornamento.

In pratica quasi tutti i sistemi concedono pieni diritti agli utenti sulla base di una sola password. Più password, in teoria, renderebbero il sistema più sicuro, ma anche meno comodo da usare, motivo per cui sistemi del genere sono rari. È un esempio di un principio generale: se la sicurezza rende qualcosa scomodo da usare, si preferisce spesso rinunciare alle misure di sicurezza, o agirarle in qualche modo.

16.5.2 Vulnerabilità delle password

Le password sono assai comuni poiché sono facili da capire e da usare. Sfortunatamente si possono indovinare, rendere accidentalmente visibili, sottrarre o trasferire illegalmente da un utente autorizzato a uno privo d'autorizzazione.

Ci sono due comuni metodi per indovinare una password. Uno si fonda sulla conoscenza dell'utente (o d'informazioni che lo coinvolgono) da parte di intrusi (sia umani sia programmi): troppo spesso le persone usano per le password informazioni ovvie (come il nome del loro gatto o del loro consorte). L'altro modo è l'uso della *forza bruta*: enumerare tutte le possibili combinazioni di lettere, numeri e segni d'interpunkzione finché si trova la password cercata. Le password brevi sono quindi le più vulnerabili; una password di quattro cifre decimali permette solo 10.000 combinazioni. In media occorrono solo 5000 tentativi per indovinare quella giusta. Un programma che può fare un tentativo ogni millisecondo impiegherebbe solo cinque secondi a indovinare una password di quattro cifre. L'enumerazione non è però adatta all'individuazione delle password in sistemi che consentono password lunghe, che distinguono le lettere maiuscole dalle minuscole, e che permettono l'uso dei numeri e dei caratteri di punteggiatura. Naturalmente gli utenti devono approfittare delle maggiori possibilità di scelta e non devono usare, per esempio, soltanto le lettere minuscole.

Oltre a essere indovinate, le password possono anche essere individuate da una sorveglianza visiva o elettronica: un intruso può sbirciare sopra la spalla (shoulder surfing) di un utente mentre questo inizia una sessione di lavoro e carpire facilmente la sua password osservando la tastiera. Alternativamente, chiunque abbia accesso alla rete in cui risiede un calcolatore può inserire un monitor di rete che gli consenta di osservare tutti i dati trasferiti nella rete (sniffing), compresi gli identificatori degli utenti e le password. La cifratura dei flussi di dati contenenti le password risolve questo problema. Tuttavia, persino un sistema di questo tipo può essere vittima di furti. Per esempio, se per conservare le password si usa un file dedicato allo scopo, questo potrebbe essere copiato su altri sistemi per analizzarlo. Si pensi anche a un cavallo di Troia installato sul sistema in grado di rilevare ogni singolo carattere battuto sulla tastiera prima che questo sia inviato alla applicazione.

L'involontaria visibilità della password diventa un problema particolarmente grave se questa viene annotata dove può essere letta o persa. Alcuni sistemi obbligano gli utenti a scegliere password lunghe o difficili da ricordare oppure a cambiare password con una certa frequenza; questo metodo può spingere qualche utente ad annotare per iscritto la propria password o a riutilizzarla, determinando una sicurezza assai minore di quella di sistemi che consentono password semplici.

L'ultimo metodo di compromissione delle password, il trasferimento illecito, deriva dalla natura umana. In molti centri di calcolo vige una regola che vieta la condivisione degli account; tale regola talvolta si stabilisce per ragioni amministrative, ma spesso è impiegata per migliorare la sicurezza. Per esempio, se uno user id è condiviso da più utenti e si è verificata con esso una violazione della sicurezza, è impossibile sapere chi l'ha usato al momento della violazione, o anche se era uno degli utenti autorizzati; con uno user id diverso per ciascun utente, ogni utente può essere direttamente reso responsabile del suo uso. Inoltre l'utente potrebbe notare qualcosa di insolito relativamente al proprio account, e scoprire la violazione. Talvolta gli utenti violano le regole di condivisione per aiutare loro amici o per aggirare la contabilizzazione; tale comportamento può causare l'accesso al sistema di utenti non autorizzati, e magari pericolosi.

Le password possono essere generate dal sistema o scelte dall'utente. Quelle generate dal sistema possono essere difficili da ricordare e quindi gli utenti potrebbero trascriverle. Quelle scelte dagli utenti sono invece più facili da indovinare, un utente potrebbe per esempio usare il proprio nome o il nome della sua automobile preferita. Gli amministratori possono controllare periodicamente le password degli utenti e informarli se sono troppo corte o troppo facili da indovinare. Prima di accettarla, alcuni sistemi verificano l'efficacia della password, cioè controllano che non sia facilmente intuibile. Alcuni sistemi prevedono l'*invecchiamento* delle password, costringendo gli utenti a modificarle a intervalli regolari, per esempio ogni tre mesi. Questo metodo non è del tutto sicuro,

poiché gli utenti possono alternare sempre le stesse due password. Questo problema si risolve, come accade in alcuni sistemi, registrando la storia delle password utilizzate da ciascun utente allo scopo di impedirne il riutilizzo.

Lo schema delle password può avere parecchie varianti. Per esempio, la password può essere cambiata spesso, anche a ogni sessione di lavoro. Alla fine di *ogni* sessione di lavoro, il sistema o l'utente sceglie una nuova password che si dovrà usare per la sessione di lavoro successiva. In questo caso, anche se una password viene usata abusivamente, può essere usata una sola volta. Quando l'utente legittimo, all'apertura della sessione di lavoro successiva, prova a usare la password, trova che non più valida e scopre la violazione della sicurezza. A questo punto si può reagire con azioni che rimedino alla violazione della sicurezza.

16.5.3 Password cifrate

Comune a tutti questi metodi è la difficoltà di mantenere segrete le password all'interno del calcolatore. Come può il sistema memorizzare le password in modo sicuro, ma permetterne l'uso quando un utente presenta la propria password? Il sistema unix utilizza funzioni hash per evitare di mantenere segreta la propria lista di password. Visto che la lista è trattata con funzioni hash e non crittografata, il sistema non può decifrarne i valori per determinare le password originali.

Le funzioni hash sono facili da calcolare, ma è difficile (se non impossibile) calcolare la loro inversa. In altre parole, dato un valore x è facile calcolare il valore della funzione hash $f(x)$, ma dato $f(x)$ è "impossibile" ottenere x . Funzioni di questo tipo sono usate per cifrare tutte le password, che vengono memorizzate solo in forma codificata. Quando un utente immette una password, questa viene cifrata e confrontata con quella già cifrata e memorizzata. Anche se la password cifrata viene vista, non potendo essere decifrata, è impossibile stabilire la password originale. Quindi non è necessario tenere segreto il file che contiene le password.

La pecca di questo metodo consiste nel fatto che il sistema operativo non ha più il controllo delle password. Anche se cifrate, chiunque disponga di una copia del file che le contiene può servirsi di efficienti procedure di hashing contro tale file, per esempio effettuando l'hash di ciascuna parola di un dizionario e confrontando il risultato con le password cifrate contenute nel file; se l'utente ha scelto come password una parola contenuta nel dizionario, la password viene scoperta. Se s'impiegano calcolatori sufficientemente veloci, o anche cluster di calcolatori lenti, un simile confronto può richiedere solo poche ore. Inoltre, poiché i sistemi fanno uso di algoritmi di hashing ben noti, un pirata informatico potrebbe conservare un insieme di password già violate.

Per questo motivo i sistemi includono nell'algoritmo di cifratura anche un numero casuale, detto *salt*, che viene aggiunto alla password per far sì che, nel caso in cui due password coincidano, le loro versioni cifrate siano diverse. Il valore salt rende inoltre inefficace la cifratura di un dizionario, perché ogni termine del dizionario dovrebbe essere combinato con ogni valore salt per poter essere confrontato con le password memorizzate. Inoltre le nuove versioni di unix registrano le password cifrate in un file che può essere letto solo dall'amministratore del sistema (superuser). I programmi che confrontano l'hash delle password inserite con i valori registrati eseguono `setuid` sull'utente `root` in modo che possano leggere questo file, ma gli altri utenti non possano farlo.

PASSWORD ROBUSTE E FACILI DA RICORDARE

È estremamente importante utilizzare password robuste (difficili da indovinare e difficili da carpire osservando la digitazione) su sistemi critici come i conti bancari. È inoltre importante non utilizzare la stessa password su molti sistemi diversi, poiché un sistema meno importante, facilmente attaccabile, potrebbe rivelare la password che si utilizza su sistemi più importanti. Una buona tecnica è quella di generare la propria password usando la prima lettera di ogni parola di una frase facilmente memorizzabile, usando sia caratteri maiuscoli che minuscoli e inserendo un numero o un segno di punteggiatura per aumentare la sicurezza. Ad esempio, la frase "My girlfriend's name is Katherine" potrebbe suggerire la password "Mgn.isK!": una password così formata è difficile da decifrare, ma facile da ricordare per l'utente. Un sistema più sicuro può consentire più caratteri nelle sue password, includendo il carattere dello spazio, in modo che un utente possa creare una **passphrase** (una password formata da un'intera frase) facile da ricordare, ma difficile da violare.

16.5.4 Password monouso

Uno dei modi in cui un sistema può evitare i tentativi di sottrazione delle password è l'uso di un insieme di password accoppiate. All'inizio di una sessione di lavoro, il sistema sceglie a caso una coppia di password dall'insieme delle password accoppiate e presenta all'utente un elemento della coppia selezionata; l'utente deve fornire l'altro elemento. In questo tipo di sistema l'utente viene sfidato (*challenge*) a fornire la risposta (*response*) corretta.

Questo metodo si può generalizzare impiegando un algoritmo come password.

Le password algoritmiche non sono soggette a riutilizzo. Un utente può cioè immettere una password e nessuna entità che intercetta tale password potrà riutilizzarla. In questo schema, l'utente e il sistema condividono una password simmetrica pw che non è mai trasmessa in modi che rischino di comprometterne la riservatezza, ma viene piuttosto utilizzata come argomento per una funzione insieme a un valore ch (*challenge*) presentato dal sistema. Il valore restituito dalla funzione $H(pw, ch)$ viene comunicato al calcolatore per l'autenticazione. Poiché il calcolatore è a conoscenza sia di pw sia di ch , può anch'esso calcolare il valore della funzione. Se i due risultati coincidono, l'identità dell'utente è autenticata. All'accesso successivo il sistema genera un altro valore ch e si ripete il procedimento. Questa volta l'autenticatore calcolato sarà diverso. Le password monouso sono uno fra i pochissimi metodi capaci d'impedire l'errata autenticazione di un utente dovuta all'esposizione della password.

Prodotti commerciali che realizzano le password monouso impiegano specifiche calcolatrici elettroniche dotate di un display e talvolta di un tastierino numerico; molte di loro hanno la forma delle carte di credito, di portachiavi, o delle periferiche usb. Il software in esecuzione sul computer o sullo smartphone fornisce all'utente la funzione $H(pw, ch)$; pw può essere immesso dall'utente o generato dalla calcolatrice in sincronizzazione con il computer. A volte pw è solo un numero di identificazione personale (pin). L'output di uno qualsiasi di questi sistemi consiste nella password monouso. Un generatore di password monouso che richiede l'input da parte dell'utente comporta un'autenticazione a due fattori (*two-factor authentication*). In questo caso sono richiesti due diversi tipi di componenti, per esempio un generatore di password monouso che genera la risposta corretta solo se il pin è valido. L'autenticazione a due fattori offre una protezione decisamente migliore rispetto all'autenticazione a fattore singolo, poiché richiede sia "qualcosa che si ha", sia "qualcosa che si sa".

Una variante di questi sistemi usa un libro dei codici (*code book*), o taccuino monouso (*one-time pad*), cioè una lista di password usa e getta. In questo caso ogni password della lista si usa, nell'ordine, una sola volta, poi si cancella dalla lista. Il diffuso sistema S/Key impiega come fonte delle password monouso uno specifico programma di calcolo o un libro di codici compilato in base a questi calcoli. Naturalmente, l'utente deve proteggere il proprio libro dei codici. È utile, a tal fine, che il libro non identifichi il sistema su cui i codici sono utilizzati.

16.5.5 Tecniche biometriche

Un'altra variante d'uso delle password per l'autenticazione coinvolge l'applicazione di tecniche biometriche. Per garantire l'accesso sicuro ad ambienti fisici protetti, come centri d'elaborazione dati, spesso si usano strumenti per la rilevazione dell'impronta del palmo o dell'intera mano. Questi lettori confrontano i parametri memorizzati con quelli che rileva la tavoletta per la lettura dei parametri della mano. Tali parametri possono comprendere una mappa della temperatura, la lunghezza delle dita, la loro larghezza e il disegno delle linee. Questi dispositivi sono però ancora troppo ingombranti e costosi per essere usati per la normale autenticazione da parte di un calcolatore.

I lettori d'impronte digitali sono diventati accurati e sufficientemente economici e dovrebbero diffondersi maggiormente in futuro. Questi dispositivi leggono il disegno formato dalle increspature della pelle sulle dita e lo convertono in una sequenza di numeri. Di solito memorizzano un insieme di sequenze, per adattarsi alla posizione del dito sulla tavoletta di lettura e ad altri fattori. Uno specifico programma può a questo punto eseguire la scansione del dito e confrontare i dati ottenuti con quelli memorizzati, per vedere se corrispondono. Chiaramente, si possono mantenere profili di molti utenti e il dispositivo di scansione consente di distinguergli. Si può ottenere uno schema di autenticazione a due fattori molto accurato richiedendo sia un nome utente e la relativa password sia la scansione dell'impronta digitale. Se per il trasferimento si cifrano queste informazioni, il sistema può essere molto resistente alle tecniche d'imitazione delle identità e di replay.

L'autenticazione multifattoriale dà risultati ancora migliori. Si pensi alle garanzie offerte da un sistema d'autenticazione che preveda il collegamento di un dispositivo usb al sistema cui si intende accedere, l'inserimento di un pin, e la lettura delle impronte digitali. A parte il fatto che l'utente dovrà collegare il dispositivo usb e poggiarvi sopra il dito, si tratta di un metodo d'autenticazione non meno comodo delle usuali password. Si ricordi, però, che un buon metodo di autenticazione non può di per sé garantire l'identità dell'utente: le sessioni autenticate possono sempre essere dirottate da intrusi, se non opportunamente crittate.

16.6 Realizzazione delle misure di sicurezza

Alla miriade di minacce che mettono a rischio le reti e i sistemi si possono contrapporre numerose strategie di sicurezza. Le strategie difensive spaziano dal miglioramento della consapevolezza degli utenti, alla tecnologia, fino allo scrivere programmi privi di bachi. Molti esperti nel campo della sicurezza sostengono la teoria della difesa in profondità, che afferma la necessità di moltiplicare gli strati di protezione per ottenere una difesa soddisfacente; tale teoria, ovviamente, si applica a qualsiasi contesto di sicurezza: si pensi alla differenza tra un'abitazione senza porta blindata, con una porta blindata, o con porta blindata e allarme. In questo paragrafo esamineremo i principali metodi, strumenti e tecniche utilizzabili al fine di migliorare la resistenza alle minacce. Si noti che alcune tecniche per migliorare la sicurezza fanno più propriamente parte della protezione e verranno quindi trattate nel Capitolo 17.

16.6.1 Politiche di sicurezza

Il primo passo per consolidare la sicurezza dei calcolatori in ogni suo aspetto è adottare una politica della sicurezza (*security policy*). Tra le politiche di sicurezza vi può essere notevole differenza ma, solitamente, essi definiscono esattamente ciò che deve essere protetto. Una politica, per esempio, potrebbe dichiarare che tutte le applicazioni accessibili dall'esterno prevedano, prima del loro utilizzo, una revisione del codice, oppure vietare la condivisione delle password da parte degli utenti o ancora, se si tratta di una società, stabilire l'obbligo di eseguire la scansione delle porte, ogni sei mesi, per tutte le interconnessioni con l'esterno. Senza la definizione di una politica, sarebbe impossibile per utenti e amministratori distinguere tra ciò che è ammissibile, ciò che è necessario e ciò che non è permesso. Le politiche delineano un percorso operativo verso la sicurezza – indispensabile, per esempio, per definire le azioni da intraprendere per i siti che vogliono conseguire un livello di protezione più alto.

Una volta che le norme di sicurezza stabilite dalla politica siano operative, i soggetti a cui sono rivolte dovrebbero conoscerle bene e farne la propria guida. Il complesso di queste norme dovrebbe costituire un documento vivo (*living document*) da aggiornare e rivedere periodicamente, per garantirne pertinenza e validità nel corso del tempo.

16.6.2 Verifica delle vulnerabilità

Come si può accettare se una politica di sicurezza sia stata correttamente attuata? Il modo migliore consiste nell'eseguire una verifica delle vulnerabilità. Il raggio d'azione di queste verifiche può spaziare dall'ingegneria sociale, alla valutazione del rischio, alla scansione delle porte. La valutazione del rischio (*risk assessment*), per esempio, è il tentativo di stimare il valore di un certo oggetto (un programma, un settore dirigenziale, un sistema o un servizio), e la probabilità che attacchi alla sicurezza ne mettano a repentaglio il valore. Solo sulla base di tali stime si può quantificare il valore economico del mettere in sicurezza l'oggetto.

Lo strumento principale delle verifiche di vulnerabilità è il test di penetrazione (*penetration test*) che scandaglia l'oggetto in esame alla ricerca di vulnerabilità note. Poiché questo libro concerne i sistemi operativi e i programmi che vi sono eseguiti, la nostra trattazione si concentrerà su questi aspetti.

Le scansioni delle vulnerabilità si effettuano di preferenza quando l'elaboratore è scarsamente utilizzato, per ridurne al minimo l'impatto. Se opportuno, vengono sperimentate tramite simulazioni su sistemi di test, e non su sistemi reali, dato che possono provocare effetti indesiderati sul funzionamento del sistema o dei dispositivi di rete.

L'azione di scandaglio all'interno di un singolo sistema può individuare vari aspetti:

- password brevi o facili da indovinare;
- programmi privilegiati non autorizzati, come i programmi di *setuid*;
- programmi non autorizzati nelle directory di sistema;
- processi dall'esecuzione inaspettatamente lunga;
- improprie protezioni delle directory sia degli utenti sia di sistema;
- improprie protezioni dei file di dati del sistema, come il file delle password, i driver dei dispositivi, o anche dello stesso kernel del sistema operativo;
- elementi pericolosi nel percorso di ricerca dei programmi (per esempio, cavalli di Troia – Paragrafo 16.2.1) quali la directory attuale e qualsiasi altra directory facile come `/tmp`;
- modifiche ai programmi di sistema, individuate con somme di controllo;
- demoni di rete inattesi o nascosti.

Ogni problema individuato dalla scansione di sicurezza può essere risolto automaticamente oppure notificato al gestore del sistema.

I calcolatori collegati in rete sono più soggetti ad attacchi contro la sicurezza di quanto lo siano i sistemi indipendenti. In questo caso gli attacchi arrivano da un insieme sconosciuto e vasto di punti d'accesso invece che da un insieme noto di punti d'accesso, come i terminali direttamente connessi. Anche se in misura minore, anche i sistemi collegati tramite modem alle linee telefoniche sono più esposti dei sistemi indipendenti.

Il governo degli Usa, per esempio, considera i sistemi sicuri tanto quanto è sicuro il loro collegamento più esteso. A un sistema di massima segretezza è possibile accedere solo dall'interno di un edificio considerato di massima segretezza. Il sistema perde il proprio grado di massima segretezza se dall'esterno di tale ambiente può avvenire un qualsiasi tipo di comunicazione. Alcuni servizi governativi prendono misure di sicurezza estreme; quando un terminale non è in uso i connettori utilizzati per collegarlo con un calcolatore sicuro vengono chiusi in una cassaforte nell'ufficio. Ecco un esempio di autenticazione multifattoriale: se un individuo vuole entrare nell'edificio, e nell'ufficio, deve possedere appositi documenti di riconoscimento, conoscere la combinazione con cui aprire la serratura, e disporre dei dati di autenticazione che gli consentono l'accesso tramite il terminale.

Sfortunatamente per gli amministratori di sistemi e per i professionisti della sicurezza dei calcolatori, è spesso impossibile confinare una macchina in una stanza e disabilitare ogni accesso remoto. La rete Internet, per esempio, connette milioni di calcolatori e sta diventando una risorsa indispensabile per il lavoro di molte società e persone. Se si considera Internet come una comunità, come

accade in ogni comunità formata di milioni di membri, moltissimi sono onesti e alcuni sono disonesti. Gli utenti disonesti di Internet dispongono di molti strumenti che consentono loro di tentare l'accesso ai calcolatori collegati.

La scansione delle vulnerabilità può essere applicata alle reti per risolvere alcuni problemi connessi alla sicurezza in rete. Ogni scansione ricerca porte che rispondano alle richieste. Se sono attivi servizi che non dovrebbero essere disponibili, il loro accesso può essere bloccato, oppure i servizi possono essere disabilitati. Le scansioni quindi analizzano i dettagli delle applicazioni che sono in ascolto su una data porta e cercano eventuali vulnerabilità note; l'esame di queste vulnerabilità rivela se il sistema è configurato erroneamente o se è privo delle necessarie patch di sicurezza.

Infine è opportuno sottolineare come i dispositivi di scansione delle porte possano essere utilizzati da pirati informatici alla ricerca di vulnerabilità nei sistemi da attaccare, invece che da addetti alla sicurezza. (Fortunatamente, è possibile tener sotto controllo l'attività di scansione delle porte attraverso il rilevamento delle anomalie, come vedremo tra breve.) La sfida costante alla sicurezza nasce da un'ambivalenza: i medesimi strumenti, infatti, possono servire a fin di bene o per fare danni. Infatti alcuni hanno sostenuto il concetto di sicurezza tramite segretezza, basato sull'idea di non sviluppare alcuno strumento per il riconoscimento delle falliche di sicurezza, perché potrebbe essere utilizzato per attaccare i sistemi. Altri contestano la validità di tale soluzione obiettando, per esempio, che i pirati informatici sono in grado di scrivere le proprie applicazioni da soli. Sembra ragionevole ritenere la sicurezza tramite segretezza solo uno degli strati della sicurezza, a patto che non resti l'unico. Per esempio, un'azienda può rendere nota al pubblico l'intera struttura della propria rete; se, invece, decidesse di tener segrete queste informazioni, gli intrusi avrebbero maggiori difficoltà nella scelta degli obiettivi da attaccare, e nello stabilire quali azioni possano essere identificate. Persino in questo caso, tuttavia, un'azienda darebbe prova di incauto ottimismo se considerasse scontata la segretezza a lungo termine di tali dati.

16.6.3 Prevenzione delle intrusioni

La sicurezza dei sistemi e delle risorse è strettamente legata alla rilevazione e prevenzione delle intrusioni. La prevenzione delle intrusioni, come suggerisce il nome, è quell'attività volta a scoprire tentativi di intrusione o intrusioni già avvenute nei sistemi elaborativi e ad attivare azioni appropriate in risposta alle intrusioni stesse. Per la prevenzione delle intrusioni s'impiegano molte tecniche che si differenziano secondo vari fattori, fra cui le seguenti.

- La fase in cui avviene il rilevamento dell'intrusione: in tempo reale, ossia mentre si sta verificando, o solo successivamente.
- Le informazioni esaminate per scoprire l'attività intrusiva. Queste potrebbero comprendere comandi impariti tramite shell, chiamate di sistema da parte dei processi, oltre a intestazioni e contenuto dei pacchetti di rete. Alcune forme d'intrusione si possono rilevare solo attraverso una correlazione delle informazioni acquisite da più di una di queste sorgenti.
- La varietà delle opzioni di risposta. Alcune semplici forme di risposta consistono nell'informare l'amministratore del sistema della potenziale intrusione oppure nel bloccare in qualche modo la potenziale attività intrusiva, per esempio arrestando un processo impegnato in tale attività. Con una forma di risposta più raffinata, un sistema potrebbe sviluppare, in modo trasparente, l'attività dell'intruso, portandolo verso un honeypot, una risorsa fittizia esposta agli attacchi. La risorsa appare reale agli intrusi, e permette di acquisire informazioni su di loro tenendone sotto controllo l'attività.

Questi gradi di libertà hanno portato a un'ampia gamma di soluzioni che vanno sotto il nome di sistemi di prevenzione delle intrusioni (ips). I sistemi ips agiscono come firewall automodificantisi, che lasciano passare il traffico fino a quando non viene rilevata una intrusione (a questo punto il traffico viene bloccato).

Ma che cosa si intende per intrusione? È difficile definirne con esattezza il significato, quindi i sistemi ips attuali di solito seguono uno tra due metodi meno ambiziosi. Secondo il primo di questi, chiamato rilevamento basato sulle tracce (*signature-based detection*), si esaminano i dati d'ingresso al sistema o il traffico della rete alla ricerca di particolari schemi o sequenze di azioni, chiamati tracce, che sono notoriamente indicazioni di attacchi. Un semplice esempio di rilevamento basato su tracce è la ricerca di pacchetti di rete che contengano la sequenza `/etc/passwd` indirizzati a un sistema unix. Un altro esempio ancora è offerto da un programma antivirus, che analizza i file binari o pacchetti di rete alla ricerca di virus conosciuti.

Il secondo metodo è in genere conosciuto come rilevamento di anomalie (*anomaly detection*), cerca di identificare, con varie tecniche, i comportamenti anomali in un sistema. Ovviamente, non tutte le attività di sistema anomale indicano un'intrusione, ma l'ipotesi che si fa è che le intrusioni generino spesso un comportamento anomalo. Un esempio di rilevamento di anomalie è il controllo delle chiamate di sistema usate da un demone, allo scopo di identificare se l'uso di queste chiamate differisce significativamente da quello usuale, indicando in questo caso la possibilità che il demone sia stato manipolato da un attacco basato su buffer overflow. Un altro esempio è il controllo dei comandi impariti all'interprete per individuare comandi anomali da parte di un certo utente, oppure l'individuazione di un accesso che si svolge in un orario anomalo per un certo utente, eventi che potrebbero indicare che un aggressore è riuscito a violare l'account di quell'utente.

Il rilevamento basato su tracce e il rilevamento di anomalie si possono considerare come due facce della stessa medaglia: il rilevamento basato su tracce cerca di caratterizzare comportamenti pericolosi e li identifica quando questi si presentano nel sistema, mentre il rilevamento di anomalie cerca di caratterizzare i comportamenti normali (o non pericoloso) e identifica qualsiasi comportamento che differisce da questi.

Questi differenti metodi tuttavia portano a ips con caratteristiche molto diverse tra loro. In particolare, il rilevamento di anomalie può riuscire a scoprire metodi d'intrusione che in precedenza erano sconosciuti (i cosiddetti attacchi giorno zero, *zero-day attacks*). Il rilevamento basato su tracce identifica invece solo gli attacchi conosciuti e che si possono codificare in uno specifico schema di azioni riconoscibili. Per questo motivo, gli attacchi che non erano stati considerati al momento della generazione delle tracce non potranno essere identificati. Il problema è ben noto alle aziende che commercializzano programmi antivirus, costrette a rilasciare frequentemente gli aggiornamenti delle *signature* a mano a mano che nuovi virus vengono creati e identificati con procedure manuali.

Il rilevamento di anomalie non è necessariamente superiore a quello basato su tracce. Infatti analizzare e caratterizzare accuratamente quello che è il comportamento "normale" di un sistema è un problema considerevole per i sistemi che si basano sul rilevamento di anomalie. Se l'intrusione è già avvenuta quando il sistema viene analizzato, l'attività intrusiva potrebbe essere inclusa nel comportamento "normale". Anche se il sistema viene analizzato correttamente, senza l'influenza di comportamenti intrusivi, l'analisi deve fornire un'immagine sufficientemente completa del comportamento normale, altrimenti il numero di falsi positivi (falsi allarmi) o peggio falsi negativi (intrusioni non individuate) sarebbe eccessivo.

Per illustrare l'impatto che può avere una frequenza di falsi allarmi anche solo moderatamente alta, si consideri un'installazione composta da un centinaio di stazioni di lavoro con sistema operativo unix dalle quali si registrano dati relativi a eventi legati alla sicurezza, allo scopo di rilevare le intrusioni. Un'installazione di piccole dimensioni come questa può facilmente generare un milione di record di verifica (*audit*) al giorno. Tuttavia, solo una o due di queste potrebbero essere tali da meritare un'analisi più approfondita da parte dell'amministratore. Se si suppone, in modo ottimistico, che ogni attacco si rifletta in dieci annotazioni di verifica, si può calcolare con una certa approssimazione la frequenza di annotazioni di verifica che riflettono le vere attività intrusive per mezzo della formula:

$$\frac{2 \frac{\text{intrusioni}}{\text{giorno}} \cdot 10 \frac{\text{annotazioni}}{\text{intrusione}}}{10^6 \frac{\text{annotazioni}}{\text{giorno}}} = 0,00002$$

Interpretando il valore che si ottiene come una “probabilità di occorrenza di record d'intrusione”, la si denota con $P(I)$; cioè, l'evento I rappresenta l'occorrenza di un record che riflette un reale comportamento intrusivo. Poiché $P(I) = 0,00002$, si trova che $P(\neg I) = 1 - P(I) = 0,99998$. Si supponga ora che A denoti l'attivazione di un allarme da parte dell'ids. Un ids accurato dovrebbe massimizzare sia $P(I|A)$ sia $P(\neg I|\neg A)$, cioè, sia la probabilità che un allarme indichi un'intrusione, sia la probabilità che l'assenza di un allarme indichi che non vi sono intrusioni. Considerando per il momento $P(I|A)$, si può calcolare tale valore usando la formula di Bayes:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} = \frac{0,00002 \cdot P(A|I)}{0,00002 \cdot P(A|I) + 0,99998 \cdot P(A|\neg I)}$$

Si consideri ora l'impatto della frequenza di falsi allarmi $P(A|\neg I)$ su $P(I|A)$. Anche nel caso di una frequenza di veri allarmi molto buona, con $P(A|I) = 0,8$, da una frequenza di falsi allarmi apparentemente buona, $P(A|\neg I) = 0,0001$, si ottiene $P(I|A) \approx 0,14$. Questo significa che meno di uno su sette allarmi indica una vera intrusione. Nei sistemi in cui un amministratore della sicurezza investiga su ogni allarme, questa frequenza di falsi allarmi porterebbe a un enorme spreco di tempo e convincerebbe molto presto l'amministratore a ignorare tutti gli allarmi.

Quest'esempio mostra un principio generale per i sistemi ips: per essere utilizzabili proficuamente, devono offrire una frequenza di falsi allarmi estremamente bassa. Per i sistemi di rilevamento di anomalie, il raggiungimento di una frequenza di falsi allarmi sufficientemente bassa è un problema particolarmente serio, proprio per le difficoltà che si hanno nell'analizzare e caratterizzare adeguatamente il comportamento normale dei sistemi. Tuttavia, la ricerca nel campo del rilevamento di anomalie registra continui miglioramenti. Il software per la rilevazione delle intrusioni si evolve in modo da integrare tracce di attacchi, algoritmi per le anomalie, e altri algoritmi ancora, così da conseguire una più accurata frequenza di rilevazione delle anomalie.

16.6.4 Protezione dai virus

I virus, come sappiamo, possono arrecare ingenti danni ai sistemi, e sono dunque una questione importante per la sicurezza. Per proteggersi si ricorre di frequente all'opera di programmi antivirus. Alcuni di questi programmi hanno efficacia solo verso particolari virus, che sono già noti: essi operano cercando, in tutti i programmi di un sistema, la specifica sequenza di istruzioni che compone il virus. Quando individuano la sequenza incriminata, eliminano le istruzioni per disinfezionare il programma. I programmi antivirus possono avere elenchi con migliaia di definizioni dei virus ricercati.

Sia i virus sia i programmi antivirus diventano sempre più sofisticati. Alcuni virus si modificano mentre infettano i programmi, allo scopo di evitare la ricerca per sequenze note, che è il metodo base dei programmi antivirus. Questi, a loro volta, si evolvono cercando famiglie di sequenze di istruzioni, piuttosto che sequenze singole; alcuni programmi antivirus implementano vari algoritmi di ricerca differenti, e possono anche decomprimere i virus compressi prima di cercarne la firma. Alcuni di loro hanno anche una funzionalità per rilevare le anomalie dei processi: un processo che apre un file eseguibile per la scrittura, per esempio, è sospetto, a meno che sia un compilatore. Un'altra tecnica diffusa prevede l'esecuzione dei programmi in una scatola di sabbia (*sandbox*, Paragrafo 17.11.3), che è un sottoambiente emulato o controllato, non comunicante con il resto del sistema. Il programma antivirus osserva l'operato del codice nella scatola di sabbia, prima di permettere che sia eseguito senza controllo. Alcuni antivirus, inoltre, anziché limitarsi all'analisi dei file nel file system, innalzano un intero scudo a protezione del sistema. Questi programmi setacciano le partizioni di avviamento, la memoria, la posta elettronica in entrata e in uscita, i file mentre vengono scaricati, i file ospitati da dischi e memorie rimovibili, e così via.

La migliore protezione contro i virus informatici è la prevenzione (o *safe computing*). Acquistare programmi sigillati dai rivenditori autorizzati, ed evitare le copie abusive provenienti da fonti pubbliche o da scambi di dischi, rappresentano il comportamento più sicuro per tenersi alla larga dalle infezioni. D'altra parte, persino i programmi legittimi non sono immuni dalle infezioni virali: ci sono stati casi di impiegati scontenti che, per causare danni alla società in cui lavoravano, hanno infettato le copie originali dei programmi messi in commercio dalla società stessa. Per quanto riguarda i virus delle macro, quando si scambiano documenti Microsoft Word, una misura di prevenzione consiste nell'utilizzo del formato di file rich text format (rtf). A differenza del formato originale Word, l'rtf non è predisposto per l'inclusione delle macro.

Un'ulteriore cautela è evitare l'apertura di qualunque allegato di posta elettronica, se il messaggio proviene da un indirizzo sconosciuto. È assodato, purtroppo, che mentre si pone riparo alle insidie trasmesse per posta elettronica, se ne scoprono costantemente di nuove. Nel 2000, per esempio, il virus *love bug* si diffondeva ampiamente sotto la falsa apparenza di un messaggio d'amore inviato da un amico del ricevente. Una volta aperto, lo script allegato in Visual Basic si propagava autotrasmettendosi ai primi indirizzi presenti in rubrica. Fortunatamente, al di là dell'intasamento dei sistemi di posta elettronica e delle cartelle della posta in arrivo degli utenti, era un virus piuttosto innocuo. Dimostrava, tuttavia, come si potesse vanificare la misura difensiva consistente nell'aprire solo gli allegati di cui si conosceva il mittente. Un metodo di difesa più efficace consiste nell'evitare l'apertura di tutti gli allegati che contengono codice eseguibile. Alcune aziende impongono tale metodo come criterio di sicurezza generale, rimuovendo tutti gli allegati dai messaggi in arrivo.

Un'altra precauzione, ancorché non eviti le infezioni, ne consente una tempestiva individuazione. Si comincia riformattando completamente i dischi del sistema, i settori d'avviamento, spesso soggetti ad attacchi da parte dei virus. Si caricano solamente programmi sicuri e si calcola una firma per ogni programma tramite il calcolo dei valori di una funzione hash. È necessario conservare la lista con i nomi dei file e le firme associate, in modo da impedire l'accesso non autorizzato. A intervalli periodici, oppure ogni volta che viene eseguito un programma, il sistema operativo ricalcerà le firme e le confronterà con quelle dell'elenco originale; ogni differenza funge da allarme per possibili infezioni.

Questa tecnica può essere combinata con altre. Si può effettuare, per esempio, un esame antivirus molto dispendioso in termini di risorse computazionali, quale quello della scatola di sabbia, creando una firma da associare ai programmi che superano la prova. Se, al nuovo avvio del programma, le firme corrispondono a quelle dell'avvio precedente, si può fare a meno di sottoporre il programma all'antivirus.

16.6.5 Verifica, accounting e log

Verifica (*auditing*), accounting e log possono diminuire le prestazioni del sistema, ma sono utili in diverse aree; una di queste è la sicurezza. L'uso di log (*logging*) può essere generale o specifico. Tutte le chiamate di sistema possono essere registrate per monitorare il comportamento dei programmi e scoprirne le anomalie. Più normalmente, sono registrati in un log gli eventi di natura sospetta. I casi di autenticazione e di autorizzazione fallita possono rivelare molto sui tentativi di violazione.

L'accounting può aggiungersi alla dotazione di strumenti degli amministratori della sicurezza; può essere utile a rivelare cambiamenti nelle prestazioni, che a loro volta potrebbero celare problemi di sicurezza. Uno tra i primi episodi di violazione ai calcolatori unix fu scoperto da Cliff Stoll, che notò un'anomalia mentre esaminava i log di accounting.

16.6.6 Firewall a protezione di sistemi e reti

Torniamo ora al problema di come i calcolatori fidati si possano collegare in modo sicuro a una rete non fidata. Una soluzione è data dall'uso di un firewall (*barriera di sicurezza*), che separa i sistemi fidati da quelli non fidati; si tratta di un calcolatore, di un router o di un dispositivo specializzato situato tra questi due gruppi di sistemi. Un firewall di rete limita l'accesso di rete tra i due domini di sicurezza e controlla e registra tutte le connessioni. Può anche bloccare le connessioni a seconda degli indirizzi di sorgente o di destinazione, delle porte di sorgente o di destinazione, o della direzione delle connessioni. I server web, per esempio, impiegano il protocollo http per comunicare con i browser; in tal caso il firewall potrebbe consentire l'accesso al server, da parte dei sistemi esterni, solo col protocollo http ma, per esempio, impedire l'accesso tramite il protocollo finger, usato dal primo worm, l'*Internet worm* di Morris, per inserirsi nei calcolatori.

Un firewall può così separare una rete in più domini. Uno schema comune considera la rete Internet come dominio non fidato; prevede una rete parzialmente fidata, la cosiddetta zona militarizzata (*demilitarized zone, dmz*), come secondo dominio; e un terzo dominio che comprende i calcolatori aziendali (Figura 16.10). Le connessioni sono consentite da Internet ai calcolatori della dmz e dai calcolatori aziendali alla rete Internet, ma non da Internet né dalla dmz ai calcolatori aziendali. Opzionalmente, può essere consentita una comunicazione controllata fra uno o più calcolatori aziendali e la dmz. Per esempio un web server nella dmz può effettuare una query su un database nella rete aziendale. In questo modo ogni accesso è controllato, e qualsiasi sistema della dmz, anche se compromesso, non può in ogni caso accedere ai calcolatori aziendali.

Ovviamente, lo stesso sistema che costituisce il firewall deve essere sicuro e resistente agli attacchi, altrimenti le sue capacità di fornire connessioni sicure possono essere compromesse. I firewall non riescono a prevenire gli attacchi che utilizzano un tunnel, ossia si trasmettono all'interno di protocolli e connessioni che gli stessi firewall permettono. Un attacco basato sul trabocco del buffer diretto a un server web non viene bloccato da un firewall perché le connessioni http sono permesse, ed è proprio il contenuto della connessione http che ospita l'attacco. Allo stesso modo gli attacchi per rifiuto del servizio possono colpire i firewall proprio come ogni altra macchina. Un altro punto vulnerabile dei firewall è la contraffazione delle identità (*spoofing*), tramite la quale un sistema non autorizzato finge di esserlo, riuscendo a soddisfare alcuni criteri di autorizzazione. Per esempio, se una regola del firewall permette una connessione da uno specifico sistema e lo identifica con il suo indirizzo ip, un altro sistema potrebbe inviare pacchetti servendosi proprio di quell'indirizzo e quindi ottenere il permesso d'accesso.

Oltre ai comuni firewall di rete, ve ne sono altri tipi più recenti, ognuno con i suoi vantaggi e svantaggi. Con firewall personale si intende un livello di software incluso nel sistema operativo, o aggiunto come applicazione. Esso non limita la comunicazione tra domini della sicurezza, ma tra il sistema e un dato host. Un utente, per esempio, può aggiungere un firewall personale al proprio pc in modo da negare a un cavallo di Troia l'accesso alla rete cui il pc è connesso. Un firewall proxy per le applicazioni comprende il protocollo seguito da certe applicazioni sulla rete. smtp, per esempio, è usato per il trasferimento di posta elettronica; un suo proxy accetta connessioni esattamente come se fosse un server, per poi connettersi al vero server smtp destinatario. Può quindi tenere sotto controllo il traffico in transito, tentando di rilevare e disabilitando comandi illegali, scoprendo i tentativi di sfruttamento dei bachi, e così via. Alcuni firewall di questo tipo sono progettati espressamente per un unico protocollo: un firewall xml, per esempio, analizza solo i contenuti xml, bloccando il codice xml non permesso o malformato. I firewall delle chiamate di sistema risiedono tra le applicazioni e il kernel, e monitorano l'esecuzione delle chiamate. In Solaris 10, per esempio, l'opzione "privilegio minimo" implementa una lista di più di cinquanta chiamate che i processi possono o non possono invocare. I processi che non hanno bisogno di generare figli, per esempio, possono vedersi vietata la possibilità di eseguire chiamate di sistema con quello scopo.

16.6.7 Altre soluzioni

Nella battaglia in corso tra progettisti di cpu e sviluppatori di sistemi operativi, da una parte, e hacker dall'altra, è stata utilizzata una particolare tecnica per difendersi da attacchi code injection. Per dar inizio a un attacco di questo tipo, gli hacker devono essere in grado di dedurre l'indirizzo esatto in memoria del loro obiettivo. In condizioni normali ciò non sarebbe molto difficile, dal momento che il layout della memoria tende a essere prevedibile. Una tecnica del sistema operativo denominata aslr (*Address Space Layout Randomization*) tenta di risolvere questo problema randomizzando gli spazi degli indirizzi, ovvero collocando in posizioni imprevedibili spazi di indirizzamento come, per esempio, le posizioni iniziali dello stack e dell'heap. L'assegnamento casuale degli

indirizzi, sebbene non infallibile, rende gli attacchi molto più difficili da mettere in atto. aslr è una funzionalità standard in molti sistemi operativi, tra cui Windows, Linux e macos.

Nei sistemi operativi mobili come ios e Android, un approccio spesso adottato consiste nel posizionare i dati utente e i file di sistema in due partizioni separate. La partizione di sistema è montata in sola lettura, mentre la partizione dati lo è in lettura-scrittura. Questo approccio presenta numerosi vantaggi, non ultimo il maggior livello di sicurezza: i file della partizione di sistema non possono essere manomessi facilmente e ciò rafforza l'integrità del sistema. Android fa un ulteriore passo avanti con l'utilizzo del meccanismo dm-verity di Linux per crittografare la partizione di sistema e rilevare eventuali modifiche.

16.6.8 Riepilogo delle difese di sicurezza

Applicando gli appropriati livelli di difesa possiamo mantenere i sistemi al sicuro da tutti, eccetto dagli attaccanti più persistenti. Riassumendo, tra i possibili livelli di difesa troviamo i seguenti.

- Educare gli utenti alla sicurezza: non collegare dispositivi di origine sconosciuta al computer, non condividere password, utilizzare password complesse, evitare di farsi trarre in inganno da strategie di ingegneria sociale, aver chiaro che una e-mail non è necessariamente una comunicazione privata, e così via.
- Informare gli utenti su come prevenire gli attacchi di phishing: non fare clic sugli allegati o sui link presenti in e-mail provenienti da mittenti sconosciuti (o anche noti); verificare (per esempio, tramite una telefonata) che una richiesta sia legittima.
- Utilizzare comunicazioni sicure quando possibile.
- Proteggere fisicamente l'hardware del computer.
- Configurare il sistema operativo per minimizzare la superficie di attacco; disattivare tutti i servizi non utilizzati.
- Configurare i demoni di sistema, i privilegi delle applicazioni e i servizi per renderli più sicuri possibile.
- Utilizzare hardware e software moderni, poiché è probabile che abbiano funzionalità di sicurezza più recenti.
- Tenere aggiornati sistemi e applicazioni, e installare le patch più recenti.
- Eseguire solo applicazioni provenienti da fonti attendibili (per esempio, software firmato digitalmente).
- Abilitare il logging e il controllo; verificare periodicamente i log o automatizzare gli avvisi.
- Installare e utilizzare software antivirus su sistemi soggetti a virus e mantenere aggiornato il software.
- Usare password e passphrase robuste e non trascriverle dove potrebbero essere trovate.
- Utilizzare il rilevamento delle intrusioni, il firewall e altri sistemi di protezione della rete, a seconda dei casi.
- Per infrastrutture importanti, effettuare valutazioni periodiche della vulnerabilità e utilizzare metodi di test per verificare la sicurezza e la risposta in caso di incidenti.
- Crittografare i dispositivi di memorizzazione di massa e prendere in considerazione l'ulteriore cifratura dei singoli file più importanti.
- Per sistemi e infrastrutture importanti, adottare una politica di sicurezza e aggiornarla periodicamente.

16.7 Un esempio: Windows 10

Microsoft Windows 10 è un sistema operativo a uso generale progettato per disporre di una varietà di funzioni e metodi di sicurezza. In questo paragrafo si esaminano gli strumenti impiegati da questo sistema operativo per garantire le funzionalità di sicurezza; per maggiori informazioni su Windows 10 si veda il Capitolo 21 (reperibile sulla piattaforma MyLab).

Il modello di sicurezza si basa sulla nozione di utente accreditato del sistema (*user account*). Il sistema operativo Windows 10 permette la creazione di un numero arbitrario di user account, che si possono raggruppare in un qualunque modo. L'accesso agli oggetti del sistema si può poi consentire o negare come si vuole. Il sistema identifica gli utenti per mezzo di un identificatore di sicurezza *unico*. Quando un utente accede al sistema, si crea un contrassegno d'accesso di sicurezza (*security access token*) che include l'identificatore di sicurezza dell'utente, gli identificatori di sicurezza per tutti i gruppi dei quali l'utente è membro, e una lista di tutti i privilegi speciali di cui l'utente gode. Alcuni esempi di privilegi speciali sono la possibilità di fare copie di backup di file e directory, di spegnere il calcolatore, di accedere al sistema in modo interattivo e di regolare l'orologio del sistema. Ogni processo che il sistema esegue per conto di un utente riceve una copia del security token. Ogniqualvolta l'utente – direttamente o per mezzo di un processo – tenta di accedere a oggetti del sistema, l'accesso è negato o concesso secondo gli identificatori di sicurezza contenuti nel token. L'autenticazione è di solito effettuata tramite nome utente e password, anche se la struttura modulare del sistema Windows 10 permette lo sviluppo di metodi di autenticazione specifici. Per esempio, si potrebbe usare un analizzatore elettronico della retina per verificare la reale identità dell'utente.

Il sistema Windows 10 usa l'idea del soggetto per far sì che i programmi eseguiti per conto di un utente non ottengano permessi d'accesso al sistema più ampi di quelli dell'utente stesso. Un soggetto (*subject*) si usa per identificare e gestire i permessi relativi a ogni programma eseguito dall'utente. È composto dall'access token e dal programma che agisce per conto dell'utente. Poiché il sistema Windows 10 si basa su un modello client-server, per controllare gli accessi si usano due classi di soggetti: soggetti semplici e soggetti server. Un esempio di soggetto semplice è il tipico programma applicativo che un utente esegue dopo aver ottenuto l'accesso al sistema. A un soggetto semplice si assegna un contesto di sicurezza definito secondo l'access token dell'utente. Un soggetto server è un processo realizzato come un server protetto che usa il contesto di sicurezza del client quando agisce per suo conto.

Come si è detto nel Paragrafo 16.6.6, l'auditing è un utile strumento per migliorare la sicurezza. Il sistema Windows 10 è dotato di un sistema di verifica integrato che permette il controllo di molte comuni minacce per la sicurezza del sistema. Alcuni esempi di casi in cui la verifica è utile per localizzare minacce alla sicurezza sono la registrazione degli accessi non riusciti per individuare i tentativi d'accesso con password casuali e quella degli accessi riusciti per scoprire attività all'interno del sistema in orari insoliti; inoltre, al fine di prevenire il diffondersi di un virus, è utile tener traccia dei tentativi di scrittura – riusciti e falliti – su file eseguibili; infine, la registrazione dei tentativi d'accesso a un file permette d'individuare gli accessi ai file riservati.

Windows Vista aveva aggiunto un controllo di integrità obbligatorio, che funziona assegnando un'etichetta di integrità a ogni oggetto e soggetto da proteggere. Per accedere a un oggetto, un dato soggetto deve avere l'accesso richiesto nell'elenco di controllo di accesso discrezionale e la sua etichetta di integrità deve essere uguale o superiore a quella dell'oggetto protetto (per l'operazione in questione). Le etichette di integrità in Windows Vista sono (in ordine crescente): non attendibile (untrusted), bassa (low), media (medium), alta (high) e di sistema (system). Inoltre, tre bit della maschera di accesso sono utilizzati per le etichette di integrità: NoReadUp, NoWriteUp e NoExecuteUp. NoWriteUp viene applicata automaticamente, di modo che un soggetto di integrità inferiore non possa eseguire un'operazione di scrittura su un oggetto di integrità superiore. Tuttavia, se non esplicitamente vietato dal descrittore di protezione, il soggetto può eseguire operazioni di lettura ed esecuzione.

Agli oggetti protetti privi di un'esplicita etichetta di integrità, viene assegnata l'etichetta "media", predefinita. L'etichetta per un dato soggetto viene assegnata durante l'accesso. Per esempio, un utente che non è amministratore riceverà un'etichetta di integrità media. Oltre alle etichette di integrità, Windows Vista utilizza il meccanismo uac (*User Account Control*), che rappresenta un account amministratore (diverso da quello predefinito) con due token separati: uno, per un utilizzo normale, vede disabilitato il gruppo Administrators predefinito e ha un'etichetta di integrità media; l'altro, per un utilizzo di livello più alto, vede il gruppo Administrators abilitato e ha un'etichetta di integrità alta.

Gli attributi di sicurezza di un oggetto del sistema Windows 10 sono descritti da un descrittore di sicurezza, contenente l'identificatore di sicurezza del proprietario dell'oggetto (che può cambiare i permessi d'accesso, un identificatore di sicurezza di gruppo usato solo dal sottosistema posix; una lista di controllo discrezionale degli accessi che stabilisce quali utenti o gruppi abbiano possibilità d'accesso (o abbiano esplicitamente negata questa possibilità) e una lista di sistemi di controllo degli accessi che controlla quali messaggi di verifica saranno generati. Opzionalmente la lista di sistema di controllo degli accessi può impostare l'integrità di un oggetto e indicare le operazioni da vietare ai soggetti con integrità più bassa: lettura, scrittura (sempre vietata) o esecuzione. Per esempio, il descrittore di sicurezza del file `foo.bar` potrebbe essere di proprietà di gwen e avere la seguente lista di controllo discrezionale degli accessi:

- proprietario gwen – consentito ogni accesso;
- gruppo cs – consentiti accessi per lettura e scrittura;
- utente maddie – nessun accesso consentito.

Inoltre, il descrittore potrebbe includere una lista di sistema di controllo degli accessi che richieda di verificare le scritture di ogni utente, oltre a un'etichetta di integrità media per vietare scrittura, lettura ed esecuzione a soggetti di integrità più bassa.

Una lista di controllo degli accessi contiene elementi composti dell'identificatore di sicurezza dell'individuo e della maschera d'accesso che definisce tutte le azioni permesse sull'oggetto, che può assumere il valore *accesso consentito* o *accesso negato* per ogni azione. I file di Windows 10 possono avere i seguenti tipi d'accesso: `ReadData`, `WriteData`, `AppendData`, `Execute`, `ReadExtendedAttribute`, `WriteExtendedAttribute`, `ReadAttributes` e `WriteAttributes`. È chiaro che ciò permette un preciso grado di controllo delle modalità d'accesso agli oggetti.

Il sistema Windows 10 classifica gli oggetti come contenitori e non contenitori. Gli oggetti contenitori, per esempio le directory, possono contenere, in senso logico, altri oggetti. Di norma, quando si crea un oggetto all'interno di un oggetto contenitore, il nuovo oggetto eredita i permessi dell'oggetto genitore; così pure, se l'utente copia un file da una directory a un'altra, il file eredita i permessi

della directory di destinazione. Gli oggetti non contenitori non ereditano alcun altro permesso. Tuttavia, se si cambiano i permessi di una directory, i nuovi permessi non si applicano automaticamente alle directory e ai file in essa già contenuti; l'utente, se lo desidera, può richiedere esplicitamente che siano applicati i nuovi permessi.

L'amministratore del sistema può anche inibire l'uso di una delle stampanti del sistema per un intero giorno o per parte di esso, e può avvalersi del Monitor delle Prestazioni per individuare problemi emergenti. In generale, un punto di forza del sistema Windows 10 è la capacità di mettere a disposizione strumenti che aiutano a fornire un ambiente informatico sicuro. Molti di questi strumenti però di solito non sono abilitati per default, il che probabilmente costituisce una delle spiegazioni alla miriade di violazioni di sicurezza che si verificano ai danni dei sistemi Windows 10. Un'altra causa è il gran numero di servizi installati da questo sistema all'avvio, e il numero di applicazioni generalmente installate. In un reale ambiente multutente, l'amministratore del sistema dovrebbe formulare e implementare una politica della sicurezza, sia con gli strumenti propri di Windows 10 sia tramite altri mezzi.

Una funzionalità che distingue la sicurezza in Windows 10 dalle versioni precedenti è la verifica della firma digitale del software (*code signing*). Alcune versioni di Windows 10 rendono la firma obbligatoria – le applicazioni che non sono firmate in modo appropriato dai loro autori non verranno eseguite – mentre altre versioni la rendono facoltativa o lasciano all'amministratore il compito di decidere che cosa fare con le applicazioni non firmate.

16.8 Sommario

- La protezione è un problema interno. La sicurezza invece deve prendere in considerazione sia il sistema elaborativo sia l'ambiente – persone, edifici, affari, oggetti di valore, minacce – all'interno del quale si usa il sistema.
- I dati memorizzati in un sistema elaborativo devono essere protetti da accessi non autorizzati, distruzioni o alterazioni dolose, e dall'introduzione accidentale d'incoerenze. È più semplice proteggere un sistema dalla perdita accidentale di coerenza dei dati che da accessi dolosi ai dati. Un'assoluta protezione dalle azioni dolose sulle informazioni memorizzate in un sistema elaborativo non è possibile, ma il costo di tali azioni può essere reso sufficientemente alto da scoraggiare quasi tutti, se non tutti, i tentativi d'accesso non autorizzati alle informazioni contenute nel sistema.
- Vari tipi di attacco possono essere sferrati contro specifici programmi e calcolatori, o anche agire in maniera indiscriminata. Le tecniche di alterazione dello stack e di buffer overflow procurano agli intrusi maggiori privilegi di accesso al sistema. I virus e i worm sono in grado di autopropagarsi e riescono talvolta a infettare migliaia di elaboratori. Gli attacchi basati sul rifiuto del servizio impediscono il corretto utilizzo dei sistemi colpiti.
- La cifratura delimita l'insieme di coloro i quali ricevono informazioni, mentre l'autenticazione circoscrive il dominio di chi le trasmette. La cifratura è il metodo utilizzato per garantire la riservatezza delle informazioni, all'atto di memorizzarle o trasferirle. La cifratura simmetrica richiede una chiave condivisa, mentre la cifratura asimmetrica è effettuata con una chiave pubblica e una chiave privata. L'uso combinato dell'autenticazione e delle funzioni hash permette di verificare che i dati non abbiano subito modifiche.
- I metodi di autenticazione degli utenti sono volti a identificare gli utenti legittimi di un sistema. Ai tradizionali metodi di protezione basati su nome-utente e password, se ne possono affiancare altri. Le password monouso, per esempio, cambiano da sessione a sessione per evitare attacchi replay. L'autenticazione a due fattori richiede due elementi di autenticazione, per esempio un dispositivo hardware insieme a un pin di attivazione. L'autenticazione multifattoriale impiega tre o più elementi. I metodi citati riducono fortemente le possibilità di falsificare l'autenticazione.
- I metodi per prevenire o rilevare le violazioni alla sicurezza comprendono una politica di sicurezza aggiornata, i sistemi di rilevamento delle intrusioni, i programmi antivirus, l'auditing e il log delle attività, il monitoraggio delle chiamate di sistema, la firma digitale del codice, le sandbox e i firewall.

Esercizi

16.1 Gli attacchi per buffer overflow sono evitabili adottando una migliore metodologia di programmazione o con l'ausilio di hardware particolare. Considerate queste soluzioni.

16.2 Una password può divenire nota ad altri utenti in diversi modi. Dite se esiste un metodo semplice per individuare un evento di questo tipo e motivate la risposta.

16.3 A quale scopo si usa un valore *salt* insieme alla password fornita dall'utente? Dove andrebbe memorizzato questo valore, e come andrebbe usato?

16.4 La lista di tutte le password è conservata all'interno del sistema operativo. Quindi, se un utente riesce a leggerla, la protezione delle password non è più garantita. Proponete uno schema per impedire il verificarsi del problema. (Suggerimento: si usino diverse rappresentazioni interne ed esterne.)

16.5 Un'estensione sperimentale del sistema operativo unix permette a un utente di associare un programma "guardiano" (*watchdog*) a un file in modo che tale programma sia attivato ogni volta che un programma richieda l'accesso al file. Il guardiano allora permette o nega l'accesso al file. Analizzate due pro e due contro di questo metodo ai fini della sicurezza.

16.6 Discutete un modo in cui gli amministratori di sistemi connessi a Internet potrebbero progettare il proprio sistema in modo da limitare o eliminare il danno causato dai worm. Quali sono gli inconvenienti dell'effettuazione dei cambiamenti che proponete?

16.7 Elencate sei elementi riguardanti la sicurezza di un sistema elaborativo per una banca. Per ogni elemento nell'elenco specificate se riguarda la sicurezza fisica, umana, o del sistema operativo.

16.8 Esponete due vantaggi offerti dalla cifratura dei dati memorizzati in un sistema elaborativo.

16.9 Quali programmi di uso comune sono vulnerabili al pericolo di attacchi di interposizione? Esaminate le soluzioni adatte a evitare questa forma di attacco.

16.10 Ponete a confronto i modelli di cifratura simmetrica e asimmetrica, considerando in quali circostanze sia opportuno utilizzare l'uno o l'altro per un sistema distribuito.

16.11 Per quale ragione $D_{kd,N}(E_{ke,N}(m))$ non garantisce l'autenticazione del mittente? A quali scopi può servire un'operazione di cifratura simile?

16.12 Spiegate come si può utilizzare l'algoritmo di cifratura asimmetrica per raggiungere i seguenti obiettivi.

- Autenticazione: il destinatario sa che può essere stato solo il mittente a generare il messaggio.
- Segretezza: soltanto il destinatario può decrittografare il messaggio.
- Autenticazione e segretezza: soltanto il destinatario può decrittografare il messaggio, e il destinatario sa che solo il mittente può aver generato il messaggio.

16.13 Si consideri un sistema che genera 10 milioni di record di monitoraggio al giorno. Si supponga, inoltre, che questo sistema riceva in media 10 attacchi al giorno e che ciascuno di questi attacchi dia luogo a 20 record di monitoraggio. Se il sistema di rilevamento delle intrusioni prevede una frequenza di veri allarmi pari a 0,6, e una frequenza di falsi allarmi pari a 0,0005, qual è la percentuale di allarmi nel sistema che corrisponde alle intrusioni reali?

16.14 Nei sistemi operativi mobili come ios e Android un approccio spesso adottato consiste nel posizionare i dati utente e i file di sistema in due partizioni separate. Oltre alla sicurezza, quali sono i vantaggi offerti da questa separazione?

CAPITOLO 17

Protezione

Nel Capitolo 16 abbiamo trattato la sicurezza, che consiste nel proteggere le risorse del computer da accessi non autorizzati, da modifiche intenzionali dannose o distruttive e dall'introduzione accidentale di incoerenze. In questo capitolo affronteremo la protezione, cioè il controllo dell'accesso di processi e utenti alle risorse definite da un sistema informatico.

I processi di un sistema operativo devono essere protetti dalle attività altrui. Molteplici meccanismi provvedono al raggiungimento di tale scopo, in modo che file, segmenti di memoria, cpu e altre risorse possano essere adoperati solo dai processi che hanno ottenuto l'autorizzazione dal sistema operativo. La protezione riguarda il meccanismo per il controllo dell'accesso alle risorse definite da un sistema elaborativo da parte di programmi, processi o utenti. Questo meccanismo deve fornire un metodo per specificare i controlli da imporre e alcuni mezzi per garantirli.

17.1 Scopi della protezione

I calcolatori sono diventati più complessi e le loro applicazioni sono cresciute a dismisura; di conseguenza è aumentata anche la necessità di proteggere la loro integrità. La protezione era originariamente concepita come un elemento aggiuntivo dei sistemi operativi con multiprogrammazione tale che utenti non fidati potessero condividere tranquillamente uno spazio di nomi logici comuni, come una directory di file, oppure uno spazio di nomi fisici comuni, come la memoria. I moderni concetti di protezione si sono evoluti in modo da aumentare l'affidabilità di qualsiasi sistema complesso che usi risorse condivise e sia connesso a piattaforme di comunicazione non sicure come la rete Internet.

È necessario disporre di sistemi di protezione per diversi motivi. Tra i più ovvi c'è la necessità di prevenire la violazione intenzionale e dannosa di un vincolo d'accesso da parte di un utente. Tuttavia, ha un'importanza più generale la necessità di assicurare che ogni componente del programma attivo in un sistema impieghi le risorse del sistema in modo coerente con i criteri (*policies*) stabiliti; per un sistema affidabile, questo requisito è assolutamente necessario.

La protezione può migliorare l'affidabilità rilevando errori latenti nelle interfacce tra sottosistemi componenti. Un'individuazione precoce di errori d'interfaccia riesce spesso a impedire la contaminazione di un sottosistema intatto da parte di un sottosistema malfunzionante. Una risorsa non protetta non può difendersi contro l'uso, o l'abuso, da parte di un utente non autorizzato o incompetente. Un sistema orientato alla protezione offre i mezzi per distinguere tra uso autorizzato e non autorizzato.

Il ruolo della protezione è quello di offrire un meccanismo d'impostazione di criteri che controllino l'uso delle risorse. Questi criteri si possono stabilire in vari modi: alcuni sono stati fissati nella fase di progettazione del sistema, altri sono determinati dalla gestione di un sistema; altri ancora sono definiti dai singoli utenti per proteggere i loro file e programmi. Un sistema di protezione deve avere una flessibilità tale da consentire d'imporre vari tipi di criteri.

I criteri d'uso di una risorsa possono variare secondo l'applicazione e possono cambiare nel tempo. Per queste ragioni la protezione non è più una questione riguardante solamente i progettisti di un sistema operativo; anche i programmatore di applicazioni hanno bisogno di meccanismi per proteggere dagli abusi le risorse create e gestite dai sottosistemi applicativi. In questo capitolo si descrivono i meccanismi di protezione che il sistema operativo dovrebbe fornire, e che anche i progettisti di applicazioni possono impiegarli nella progettazione del proprio sistema di protezione.

I criteri vanno distinti dai *meccanismi*. I meccanismi determinano *come* qualcosa si debba eseguire; i criteri decidono *che cosa* si debba fare. La distinzione tra criteri e meccanismi è importante ai fini della flessibilità. I criteri sono soggetti a cambiamenti in dipendenza dall'ambiente e dal tempo. Nel peggior dei casi qualsiasi cambiamento di criterio richiederebbe un cambiamento anche nel meccanismo sottostante. Utilizzando meccanismi generali si possono evitare tali situazioni.

17.2 Princìpi di protezione

Spesso un principio guida può essere utilizzato attraverso un intero progetto, quale per esempio la progettazione di un sistema operativo. L'osservanza di questo principio semplifica le decisioni relative al progetto, garantendo che il sistema sia coerente e facile da capire. Un principio guida, che nel tempo ha confermato la sua importanza per la protezione, è il principio del minimo privilegio. Come discusso nel Capitolo 16, esso sancisce che programmi, utenti, e finanche i sistemi, debbano ricevere solo i privilegi strettamente necessari per l'esecuzione dei rispettivi compiti.

Uno dei principi di unix sancisce che un utente non debba essere eseguito come root (in unix solo l'utente root può eseguire comandi privilegiati). La maggior parte degli utenti rispetta questo principio in maniera innata, temendo di eseguire un'operazione accidentale di cancellazione senza disporre in seguito di una corrispondente operazione di ripristino. Poiché la root è virtualmente onnipotente, il potenziale di errore umano quando un utente agisce come root è notevole e le conseguenze di un errore sono di vasta portata.

Consideriamo ora che un evento dannoso possa derivare da un attacco, invece che da un errore umano, per esempio da un virus lanciato da un clic accidentale su un allegato. Un altro esempio è un buffer overflow o un altro attacco di tipo code injection portato a termine con successo contro un processo con privilegi di root (o, in Windows, con privilegi di amministratore). In entrambi i casi l'evento potrebbe rivelarsi catastrofico per il sistema.

Osserviamo che il principio del minimo privilegio darebbe al sistema la possibilità di mitigare l'attacco: se il codice malevolo non può ottenere i privilegi di root, c'è la possibilità che autorizzazioni adeguatamente definite possano bloccare tutte le operazioni dannose, o almeno alcune di queste. In questo senso, le autorizzazioni possono agire da sistema immunitario a livello di sistema operativo.

Il principio del minimo privilegio assume molte forme, che esamineremo in dettaglio più avanti nel capitolo. Un altro principio importante, spesso visto come un derivato del principio del minimo privilegio, è la compartmentazione, ovvero il processo di protezione di ogni singolo componente del sistema attraverso l'uso di autorizzazioni specifiche e restrizioni di accesso. Quando un componente viene alterato, entra in funzione un'altra linea di difesa per impedire all'aggressore di compromettere ulteriormente il sistema. La compartmentazione è implementata in molte forme, dalle zone demilitarizzate (dmz) a livello di rete, alla virtualizzazione.

L'uso attento delle restrizioni di accesso può aiutare a rendere un sistema più sicuro e può essere utile nella creazione di *tracce di verifica* (audit trail), che tengono traccia delle divergenze rispetto agli accessi consentiti. Un audit trail è un record nei log di sistema che, se monitorato attentamente, può rivelare i primi segnali di un attacco o (se resta integro nonostante l'attacco) fornire indizi sui vettori di attacco che sono stati utilizzati, oltre a stimare accuratamente il danno causato.

La cosa forse più importante è che nessun singolo principio è una panacea per le vulnerabilità di sicurezza. Deve essere usata una *difesa in profondità*: devono cioè essere applicati diversi strati di protezione l'uno sull'altro (si pensi per esempio a un castello con una guarnigione, un muro e un fossato per proteggerlo). Allo stesso tempo, ovviamente, gli aggressori usano diversi mezzi per superare la difesa in profondità, con il risultato di una corsa agli armamenti sempre crescente.

17.3 Anelli di protezione

Come abbiamo visto, il componente principale dei moderni sistemi operativi è il kernel, che gestisce l'accesso alle risorse di sistema e all'hardware. Il kernel, per definizione, è un componente affidabile e privilegiato e, pertanto, deve essere eseguito con un livello di privilegi più elevato rispetto ai processi utente.

Per eseguire questa separazione dei privilegi è richiesto il supporto hardware. Tutto l'hardware moderno supporta la nozione di livelli di esecuzione separati, anche se le implementazioni variano leggermente. Un modello di separazione dei privilegi utilizzato di frequente è quello degli anelli di protezione. In questo modello, ispirato a Bell-LaPadula (<https://www.acsac.org/2005/papers/Bell.pdf>), l'esecuzione è definita come un insieme di anelli concentrici in cui un anello i fornisce un sottoinsieme delle funzionalità di un anello j , per ogni $j < i$. L'anello più interno, l'anello 0, fornisce quindi l'intera classe di privilegi. Il modello ad anelli è mostrato nella Figura 17.1.

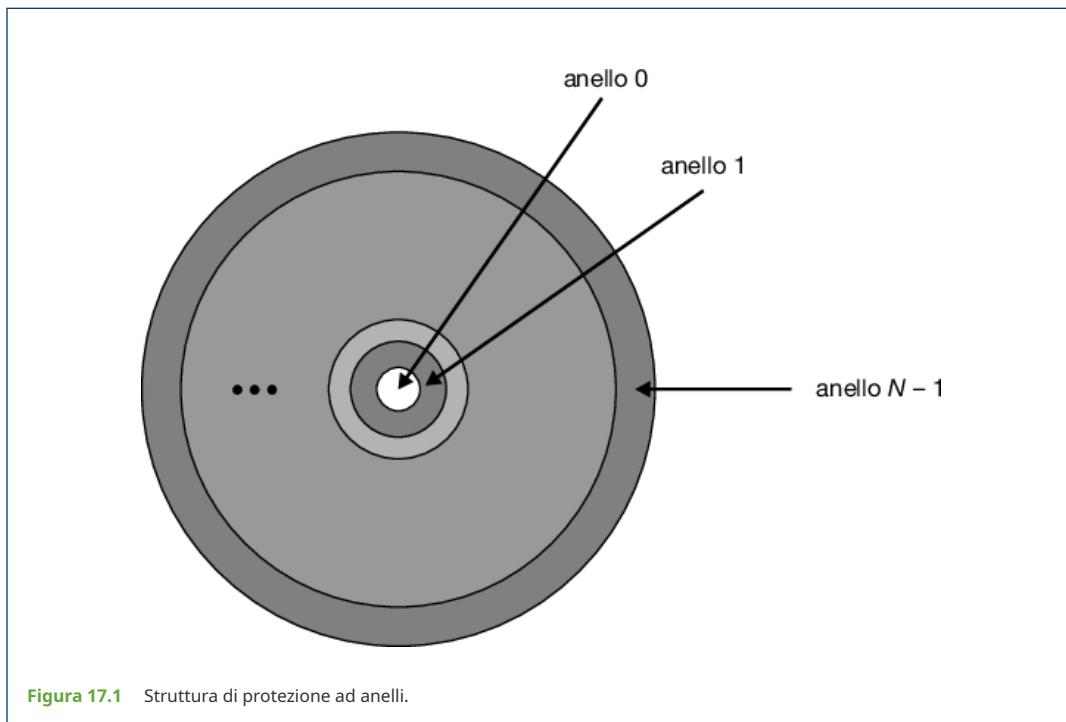


Figura 17.1 Struttura di protezione ad anelli.

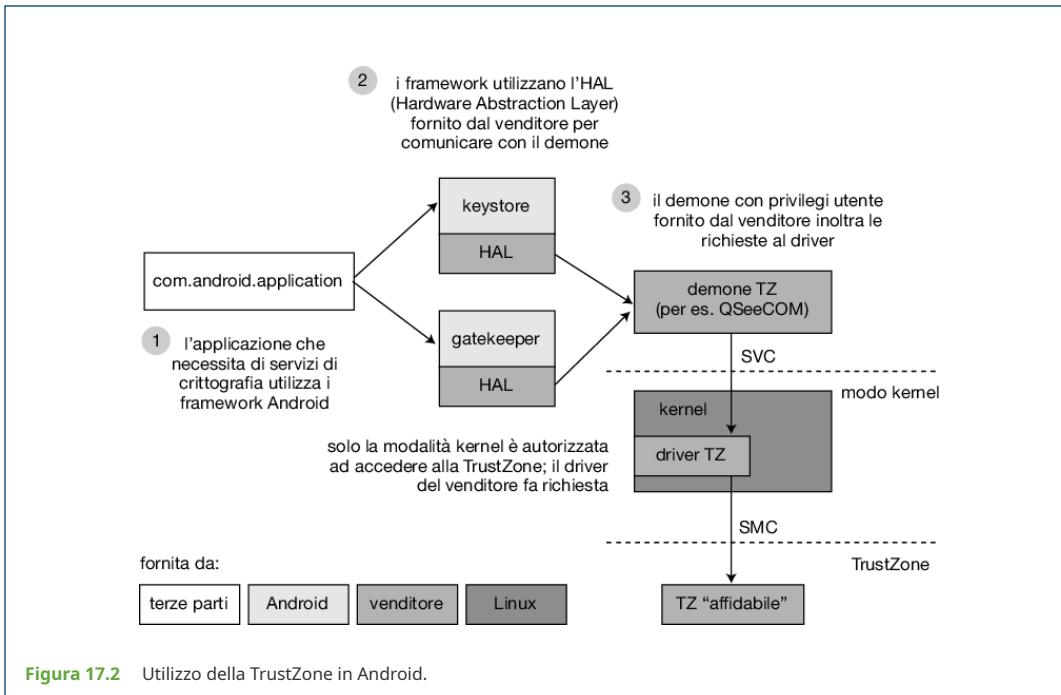
L'avvio di un sistema avviene al massimo livello di privilegi. Il codice di quel livello esegue l'inizializzazione necessaria prima di passare a un livello meno privilegiato. Per tornare a un livello di privilegi più elevato, il codice chiama di solito un'istruzione speciale, a volte denominata gate, che fornisce un passaggio tra gli anelli. Ne è un esempio l'istruzione `syscall` (in Intel), che sposta l'esecuzione dalla modalità utente alla modalità kernel. Come abbiamo visto, l'esecuzione di una chiamata di sistema trasferisce sempre l'esecuzione a un indirizzo predefinito, consentendo al chiamante di specificare solo gli argomenti (incluso il numero della chiamata di sistema) e non indirizzi arbitrari del kernel. In questo modo, l'integrità dell'anello più privilegiato resta generalmente assicurata.

Un altro meccanismo che causa lo spostamento in un anello più privilegiato è il verificarsi di una trap del processore o di un'interruzione. Quando l'una o l'altra si verifica, l'esecuzione viene immediatamente trasferita all'anello con privilegi più elevati. Ancora una volta, tuttavia, l'esecuzione nell'anello con privilegi maggiori è predefinita e limitata a un percorso di codice ben controllato.

Le architetture Intel seguono questo modello, inserendo il codice della modalità utente nell'anello 3 e il codice della modalità kernel nell'anello 0. La distinzione viene effettuata da due bit nel registro speciale `eflags`. L'accesso a questo registro non è consentito nell'anello 3, impedendo così a un processo dannoso di aumentare i suoi privilegi. Con l'avvento della virtualizzazione, Intel ha definito un anello aggiuntivo (-1) riservato agli hypervisor o ai gestori di macchine virtuali, che creano ed eseguono macchine virtuali. Gli hypervisor hanno più privilegi dei kernel dei sistemi operativi ospitati.

L'architettura del processore arm consente inizialmente solo le modalità `usr` e `svc`, rispettivamente per le modalità utente e kernel (supervisore). Nei processori armv7 è stata introdotta la `TrustZone` (tz), che ha fornito un anello aggiuntivo. Questo ambiente di esecuzione privilegiato ha anche accesso esclusivo alle funzionalità di crittografia supportate dall'hardware, come l'`nfC` Secure Element, e una chiave di crittografia su chip, che rende più sicura la gestione delle password e delle informazioni sensibili. Persino lo

stesso kernel non ha accesso alla chiave su chip e può solo richiedere tramite l'istruzione specializzata smc, Secure Monitor Call, utilizzabile solo dalla modalità kernel, i servizi di crittografia e decrittografia all'ambiente TrustZone. Come per le chiamate di sistema, il kernel non ha la possibilità di passare l'esecuzione direttamente su specifici indirizzi nella TrustZone, ma può solo passare argomenti tramite registri. Android utilizza la TrustZone in maniera estesa a partire dalla sua versione 5.0, come mostrato nella Figura 17.2.



Se si utilizza correttamente un *trusted execution environment* ("ambiente di esecuzione attendibile"), nel caso in cui il kernel sia compromesso, un utente malintenzionato non può recuperare con facilità la chiave dalla memoria del kernel. Lo spostamento dei servizi di crittografia in un ambiente separato e attendibile rende anche meno probabili gli attacchi a forza bruta (come descritto nel Capitolo 16, questi attacchi consistono nel provare tutte le possibili combinazioni di caratteri di password validi fino a quando non viene trovata la password). Le varie chiavi utilizzate dal sistema, dalla password dell'utente a quella di sistema, sono memorizzate nella chiave su chip, che è accessibile solo nel contesto attendibile. Quando viene immessa una parola chiave, diciamo una password, questa viene verificata tramite una richiesta all'ambiente TrustZone. Se una chiave non è nota e deve essere indovinata, il controllore di TrustZone può imporre limitazioni, per esempio limitando il numero di tentativi.

Nell'architettura armv8 a 64 bit, arm ha esteso il suo modello per supportare quattro livelli, denominati "livelli di eccezione" e numerati da el0 a el3. La modalità utente viene eseguita in el0 e la modalità kernel in el1, mentre el2 è riservato agli hypervisor e el3 (il più privilegiato) è riservato al secure monitor (il livello TrustZone). Qualsiasi livello di eccezione consente di eseguire sistemi operativi distinti affiancati, come mostrato nella Figura 17.3.

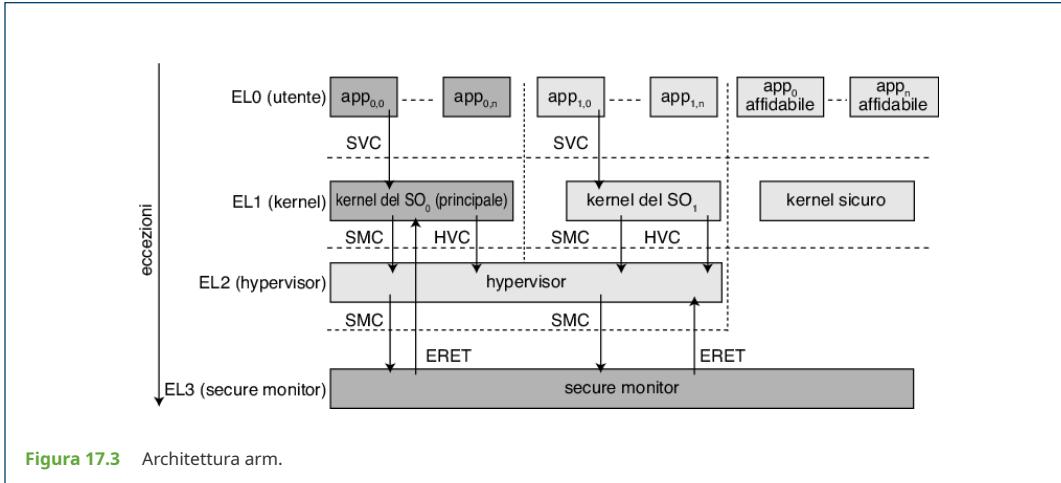


Figura 17.3 Architettura arm.

Si noti che il secure monitor viene eseguito a un livello superiore rispetto ai kernel generici, il che lo rende il luogo ideale per il codice in grado di verificare l'integrità dei kernel. Questa funzionalità è inclusa in *Real Time Kernel Protection* (rkp) di Samsung per Android e in *Apple WatchTower* (noto anche come kpp, *Kernel Patch Protection*) per ios.

17.4 Domini di protezione

Gli anelli di protezione separano le funzioni in domini e le ordinano gerarchicamente. Una generalizzazione degli anelli utilizza domini privi di gerarchia. Un sistema elaborativo è un insieme di processi e oggetti. Con il nome di *oggetti* si designano sia gli oggetti fisici, come cpu, segmenti di memoria, stampanti, dischi e unità a nastri, sia gli oggetti logici, come file, programmi e semafori. Ogni oggetto ha un nome unico che lo distingue da tutti gli altri oggetti del sistema; è inoltre possibile accedere a ciascuno di loro solo tramite operazioni ben definite e significative. Gli oggetti sono fondamentalmente tipi di dati astratti.

Le operazioni possibili dipendono dall'oggetto: per esempio, una cpu può compiere solo elaborazioni; i segmenti di memoria si possono leggere e scrivere, mentre da un lettore di cd-rom o dvd-rom si può soltanto leggere; le unità a nastri si possono leggere, scrivere e riavvolgere; i file di dati si possono creare, aprire, leggere, scrivere, chiudere e cancellare; i file di programmi si possono leggere, scrivere, eseguire e cancellare.

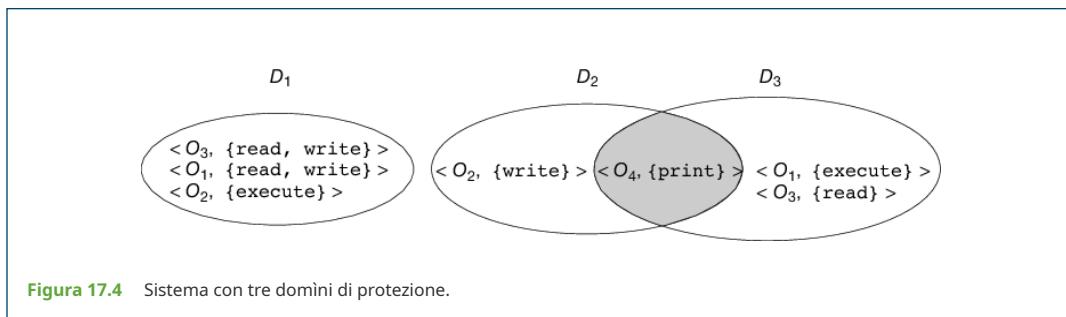
A un processo si deve permettere l'accesso alle sole risorse su cui ha l'autorizzazione. Inoltre, a un dato istante, un processo deve poter accedere solamente alle risorse di cui ha bisogno per eseguire il proprio compito. Questo secondo requisito, noto con il nome di principio della necessità di sapere (*need-to-know-principle*), è utile per limitare i danni che possono essere causati al sistema da un processo difettoso. Se, per esempio, il processo *P* invoca la procedura *A()*, alla procedura si deve permettere l'accesso solo alle proprie variabili e ai parametri che le vengono passate; non deve poter accedere a tutte le variabili del processo *P*. Analogamente, si consideri il caso in cui il processo *P* richieda la compilazione di un particolare file. Al compilatore non si deve permettere l'accesso a qualsiasi file, ma solo al ben definito sottoinsieme di file associato al file da compilare. Viceversa, il compilatore può avere dei file privati, che impiega per scopi di accounting o di ottimizzazione, ai quali il processo *P* non deve aver accesso.

Confrontando il principio della necessità di sapere con quello del minimo privilegio, può essere facile pensare al primo come alla politica adottata e al secondo come al meccanismo per ottenere questa politica. Per esempio, nel caso dei permessi dei file, la necessità di sapere potrebbe stabilire che un utente abbia accesso a un file in lettura, ma non in scrittura e in esecuzione, mentre il principio del minimo privilegio richiederebbe che il sistema operativo fornisca un meccanismo per consentire la lettura, ma non l'accesso in scrittura o in esecuzione.

17.4.1 Struttura dei domini di protezione

Per facilitare lo schema appena descritto, un processo opera all'interno di un dominio di protezione, che specifica le risorse accessibili dal processo. Ogni dominio definisce un insieme di oggetti e i tipi di operazioni che si possono compiere su ciascun oggetto. La possibilità di eseguire un'operazione su un oggetto è detta diritto d'accesso. Un dominio è dunque un insieme di diritti d'accesso, ciascuno dei quali è composto di una coppia ordinata *<nome oggetto, insieme di diritti>*. Per esempio, se il dominio *D* ha il diritto d'accesso *<fileF, {read, write}>*, un processo in esecuzione nel dominio *D* può leggere e scrivere il file *F*, ma non può eseguire altre operazioni su quello stesso oggetto.

I domini possono condividere diritti d'accesso. Nella Figura 17.4, per esempio, sono illustrati tre domini: *D₁*, *D₂* e *D₃*. Il diritto d'accesso *<O₄, {print}>* è condiviso da *D₂* e *D₃*, implicando che un processo in esecuzione su uno dei due domini può stampare l'oggetto *O₄*. Occorre notare che un processo, per leggere e scrivere l'oggetto *O₁*, deve essere in esecuzione nel dominio *D₁*. D'altra parte, soltanto i processi che si trovano nel dominio *D₃* possono eseguire l'oggetto *O₁*.



L'associazione tra un processo e un dominio può essere statica, se l'insieme delle risorse disponibili per un processo è fisso per tutta la durata del processo, o dinamica. Com'è prevedibile, la determinazione dei domini di protezione dinamici è più complicata della determinazione dei domini di protezione statici.

Se l'associazione tra processi e domini è fissa, per aderire al principio del privilegio minimo è necessario disporre di un meccanismo che permetta di modificare il contenuto di un dominio. Ciò deriva dal fatto che l'esecuzione di un processo si può dividere in più fasi e, per esempio, il processo può richiedere l'accesso in lettura in una fase e l'accesso in scrittura in un'altra. Se un dominio è statico, occorre definire un dominio che contenga sia l'accesso per la lettura sia quello per la scrittura. Tuttavia questa disposizione fornisce più diritti di quanti siano necessari in ciascuna delle due fasi, poiché è disponibile il diritto d'accesso per la lettura nella fase in cui è necessario il solo accesso per la scrittura e viceversa; quindi il principio del privilegio minimo è violato. È necessario permettere che il contenuto di un dominio sia modificato, in modo che il dominio rifletta sempre i minimi diritti d'accesso necessari.

Se l'associazione è dinamica, deve essere disponibile un meccanismo per permettere a un processo di passare da un dominio all'altro (*domain switching*). Si può anche voler modificare al contenuto di un dominio. Se non è possibile modificare il contenuto di un dominio, si può ottenere lo stesso effetto creando un nuovo dominio con il contenuto modificato e passando al nuovo dominio quando sia necessario modificare il contenuto del dominio originario.

Si noti che un dominio si può realizzare in diversi modi.

- Ogni *utente* può essere un dominio. In questo caso l'insieme d'oggetti cui si può accedere dipende dall'identità dell'utente. Il cambio di dominio avviene quando cambia l'utente, generalmente quando un utente chiude una sessione di lavoro e un altro la comincia.
- Ogni *processo* può essere un dominio. In questo caso l'insieme d'oggetti cui si può accedere dipende dall'identità del processo. Il cambio di dominio avviene quando un processo invia un messaggio a un altro processo e quindi attende una risposta.
- Ogni *procedura* può essere un dominio. In questo caso l'insieme d'oggetti cui si può accedere corrisponde alle variabili locali definite all'interno della procedura. Il cambio di dominio avviene quando s'invoca una procedura.

Il cambio di dominio è approfondito nel Paragrafo 17.5.

Si consideri l'ordinaria duplice modalità di funzionamento (di sistema e utente) dei sistemi operativi. Quando si esegue un processo in modalità di sistema, esso può impiegare istruzioni privilegiate e quindi acquisire il controllo completo del calcolatore. D'altra parte, se è eseguito in modalità utente, il processo può impiegare solo istruzioni non privilegiate; di conseguenza, può essere eseguito solo all'interno del proprio spazio di memoria predefinito. Questi due modi proteggono il sistema operativo, che è eseguito nel dominio di sistema, dai processi utenti, che si eseguono nel dominio dell'utente. In un sistema operativo con multiprogrammazione due domini di protezione sono insufficienti, poiché gli utenti richiedono anche la protezione reciproca. Serve quindi uno schema più elaborato: lo illustreremo esaminando due influenti sistemi operativi, unix e Android, e osservando come realizzano questi concetti.

17.4.2 Un esempio: unix

In unix, come osservato in precedenza, l'utente root può eseguire comandi privilegiati, mentre gli altri utenti non possono farlo. Questa limitazione può ostacolare le ordinarie operazioni degli utenti. Si consideri, per esempio, un utente che voglia cambiare la sua password. Inevitabilmente, ciò richiede l'accesso al database delle password (solitamente `/etc/shadow`), a cui è possibile accedere solo da root. Un problema simile si incontra quando si imposta un lavoro programmato (utilizzando il comando `at`): l'operazione richiede l'accesso a directory privilegiate che sono fuori dalla portata di un utente normale.

La soluzione a questo problema è il bit setuid. In unix, a ogni file sono associati un identificativo del proprietario e un bit di dominio, noto come bit setuid, che può essere abilitato oppure no. Quando il bit viene abilitato su un file eseguibile (per mezzo di `chmod +s`), chiunque esegua il file assume temporaneamente l'identità del proprietario. Ciò significa che, se un utente riesce a creare un file con l'id utente "root" e il bit setuid abilitato, chiunque ottenga l'accesso per eseguire il file diventa utente "root" per l'intera durata del processo.

Se state pensando che questa soluzione sia allarmante, ne avete buone ragioni. A causa del loro potenziale, i binari eseguibili con setuid devono necessariamente possedere due importanti proprietà: essere "sterili" (interessare esclusivamente i file necessari, sotto vincoli specifici) ed essere "ermetici" (per esempio, a prova di manomissione e impossibili da corrompere). I programmi setuid devono essere scritti scrupolosamente per offrire queste garanzie. Tornando all'esempio di modifica delle password, il comando `passwd` è setuid-root e in effetti modificherà il database delle password, ma solo dopo la verifica della validità della password dell'utente e solo limitandosi a modificare esclusivamente la password di quell'utente.

Sfortunatamente, l'esperienza ha ripetutamente dimostrato che solo pochi binari setuid (se ve n'è alcuno) soddisfano entrambi i criteri con successo. Più volte i binari setuid sono stati corrotti, alcuni per mezzo di race condition e altri mediante code injection, consentendo un accesso di root immediato agli aggressori. Attraverso queste tecniche, gli aggressori hanno spesso successo nell'aumentare i propri privilegi. I metodi per farlo sono discussi nel Capitolo 16. La limitazione del danno provocato dagli errori nei programmi setuid è discussa nel Paragrafo 17.8.

17.4.3 Un esempio: ID delle applicazioni Android

In Android vengono forniti id utente distinti in base alle applicazioni. Quando un'applicazione viene installata, il demone `installd` le assegna un id utente (uid) e un id gruppo (gid) distinti, insieme a una directory dati privata (`/data/data/<nome_app>`) la cui proprietà è concessa solo a questa combinazione uid/gid. In questo modo le applicazioni sul dispositivo godono dello stesso livello di protezione fornito a utenti distinti nei sistemi unix. Si tratta di un modo rapido e semplice per fornire isolamento, sicurezza e privacy. Il meccanismo viene esteso modificando il kernel per consentire determinate operazioni (come i socket di rete) solo ai membri di un particolare gid (per esempio, `aid_inet`, 3003). Un ulteriore miglioramento in Android consiste nel definire determinati uid come "isolati", limitando la loro possibilità di avviare richieste rpc esclusivamente a un numero minimo di servizi.

17.5 Matrice d'accesso

Il modello generale di protezione qui descritto si può considerare in modo astratto come una matrice, chiamata matrice d'accesso. Le righe della matrice rappresentano i domini, e le colonne gli oggetti. Ciascun elemento della matrice consiste di un insieme di diritti d'accesso. Poiché le colonne definiscono esplicitamente gli oggetti, si possono omettere i nomi degli oggetti dai diritti d'accesso. L'elemento $\text{access}(i, j)$ definisce l'insieme di operazioni che un processo in esecuzione nel dominio D_i può richiamare sull'oggetto O_j .

Per spiegare questi concetti si consideri la matrice d'accesso riportata nella Figura 17.5, in cui sono presenti quattro domini e quattro oggetti: tre file (F_1, F_2, F_3) e una stampante. Quando viene eseguito nel dominio D_1 , un processo può leggere i file F_1 ed F_3 . Un processo in esecuzione nel dominio D_4 ha gli stessi privilegi di un processo in esecuzione nel dominio D_1 , ma in più può scrivere anche sui file F_1 e F_3 . Solo un processo in esecuzione nel dominio D_2 può accedere alla stampante.

Lo schema della matrice d'accesso offre un meccanismo che permette di specificare diversi criteri. Il meccanismo consiste nella realizzazione della matrice d'accesso e nella garanzia che le proprietà semantiche sottolineate siano sempre valide. Più specificamente, occorre assicurare che un processo in esecuzione nel dominio D_i possa accedere solo agli oggetti specificati nella riga i e solo nel modo indicato dagli elementi della matrice d'accesso.

Con la matrice d'accesso si possono attuare i criteri riguardanti la protezione. Tali criteri implicano la scelta dei diritti da inserire nell' (i, j) -esimo elemento. Occorre anche stabilire il dominio in cui avviene l'esecuzione di ogni processo. Quest'ultimo criterio è generalmente stabilito dal sistema operativo.

Normalmente sono gli utenti a decidere il contenuto degli elementi della matrice d'accesso. Quando un utente crea un nuovo oggetto O_j , si aggiunge la colonna O_j alla matrice d'accesso con gli elementi di inizializzazione stabiliti dal creatore. L'utente può decidere di inserire alcuni diritti in determinati elementi della colonna j e altri diritti in altri elementi, secondo le necessità.

La matrice d'accesso fornisce un meccanismo adeguato alla definizione e realizzazione di uno stretto controllo sia per l'associazione statica sia per l'associazione dinamica tra processi e domini. Quando un processo passa da un dominio all'altro, si esegue un'operazione (`switch`) su un oggetto (il dominio). Il passaggio da un dominio all'altro si può controllare inserendo i domini tra gli oggetti della matrice d'accesso. Analogamente, quando si modifica il contenuto della matrice d'accesso, si esegue un'operazione su un oggetto: la matrice d'accesso. Anche in questo caso si possono controllare le modifiche considerando la stessa matrice d'accesso come un oggetto. In effetti, poiché si può modificare singolarmente, ogni elemento della matrice d'accesso si deve considerare come un oggetto da proteggere. A questo punto si devono considerare solo le operazioni possibili su questi nuovi oggetti, domini e matrice d'accesso, e occorre decidere come debbano essere eseguite dai processi.

I processi devono poter passare da un dominio all'altro. Il passaggio dal dominio D_i al dominio D_j è permesso se e solo se l'operazione $\text{switch} \in \text{access}(i, j)$. Quindi, com'è illustrato nella Figura 17.6, un processo in esecuzione nel dominio D_2 può passare al dominio D_3 oppure al dominio D_4 . Un processo del dominio D_4 può passare al dominio D_1 , e uno del dominio D_1 può passare al dominio D_2 .

oggetto dominio	F_1	F_2	F_3	stampante	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figura 17.6 Matrice d'accesso della Figura 17.5 con domini come oggetti.

dominio \ oggetto	F_1	F_2	F_3	stampante
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figura 17.5 Matrice d'accesso.

Permettere la modifica controllata del contenuto degli elementi della matrice d'accesso richiede altre tre operazioni: `copy`, `owner` e `control`. Esamineremo queste operazioni nel seguito.

La possibilità di copiare un diritto d'accesso da un dominio (o riga) a un altro della matrice d'accesso è indicata da un asterisco (*) apposto sul diritto d'accesso. Il diritto `copy` permette di copiare il diritto d'accesso solo all'interno della colonna (cioè per l'oggetto) per la quale il diritto stesso è definito. Per esempio, nella Figura 17.7 (a), un processo in esecuzione nel dominio D_2 può copiare l'operazione `read` in un elemento qualsiasi associato al file F_2 . Quindi, la matrice d'accesso della Figura 17.7(a) si può modificare nella matrice d'accesso illustrata nella Figura 17.7(b). Questo schema ha due ulteriori varianti.

dominio \ oggetto	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

dominio \ oggetto	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(a)

(b)

Figura 17.7 Matrice d'accesso con diritti `copy`.

- Un diritto viene copiato da $access(i, j)$ ad $access(k, j)$ e viene successivamente rimosso da $access(i, j)$; quest'azione è un *trasferimento* di un diritto piuttosto che una copiatura.
- La propagazione del diritto `copy` può essere limitata. Ciò significa che quando si copia il diritto R^* da $access(i, j)$ ad $access(k, j)$, si crea solo il diritto R e non R^* . Un processo in esecuzione nel dominio D_k non può copiare ulteriormente il diritto R .

Un sistema può scegliere uno tra questi tre diritti `copy`, oppure può fornirli tutti e tre identificandoli come diritti separati: `copy`, `transfer` e `limited copy`.

Occorre anche un meccanismo che permetta di aggiungere nuovi diritti e rimuoverne altri; queste operazioni sono controllate dal diritto `owner`. Se $access(i, j)$ contiene il diritto `owner`, un processo in esecuzione nel dominio D_i può aggiungere e rimuovere qualsiasi diritto di qualsiasi elemento della colonna j . Per esempio, nella Figura 17.8(a) il dominio D_1 è il proprietario di F_1 e quindi può aggiungere e cancellare qualsiasi diritto valido nella colonna di F_1 . Analogamente, il dominio D_2 è proprietario di F_2 e F_3 , quindi può aggiungere e rimuovere qualsiasi diritto valido che si trovi all'interno di queste due colonne. Così, la matrice d'accesso della Figura 17.8(a) si può modificare nella matrice d'accesso illustrata nella Figura 17.8(b).

dominio \ oggetto	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

dominio \ oggetto	F_1	F_2	F_3
D_1	owner execute		
D_2			owner read* write*
D_3		write	write

(b)

Figura 17.8 Matrice d'accesso con diritti owner.

I diritti `copy` e `owner` permettono a un processo di modificare gli elementi di una colonna. Occorre anche un meccanismo per modificare gli elementi di una riga. Il diritto `control` si può applicare solo a oggetti di dominio. Se $access(i, j)$ contiene il diritto `control`, allora un processo in esecuzione nel dominio D_i può rimuovere qualsiasi diritto d'accesso dalla riga j . Si supponga, prendendo come esempio la Figura 17.6, di inserire il diritto `control` in $access(D_2, D_4)$. Quindi un processo in esecuzione nel dominio D_2 può modificare il dominio D_4 , com'è illustrato nella Figura 17.9.

dominio \ oggetto	F_1	F_2	F_3	stampante	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figura 17.9 Matrice d'accesso della Figura 17.6 modificata.

I diritti `copy` e `owner` forniscono un meccanismo per limitare la propagazione dei diritti d'accesso, però non forniscono strumenti adeguati per impedire la propagazione delle informazioni. Il problema di garantire che nessuna informazione, tenuta inizialmente in un oggetto, possa migrare all'esterno del proprio ambiente d'esecuzione si chiama problema della reclusione (*confinement problem*). Tale problema è in generale insolubile (si vedano le Note bibliografiche in proposito).

Queste operazioni sui domini e sulla matrice d'accesso non sono di per sé importanti, ma spiegano come il modello della matrice d'accesso consenta la realizzazione e il controllo dei requisiti di protezione dinamica. Nuovi oggetti e nuovi domini si possono creare dinamicamente e inserire nel modello della matrice d'accesso. Tuttavia, in questo paragrafo è spiegato soltanto il meccanismo di base; chi progetta e usa il sistema deve scegliere i criteri riguardanti quali domini, e in che modo, abbiano accesso a determinati oggetti.

17.6 Realizzazione della matrice d'accesso

È necessario implementare efficacemente la matrice d'accesso. Tale matrice generalmente è sparsa, ossia la maggior parte dei suoi elementi è vuota. Tuttavia, a causa del modo in cui si usa la funzione di protezione, le tecniche standard di strutturazione dei dati per la rappresentazione delle matrici sparse non sono particolarmente utili per quest'applicazione. Nel seguito descriveremo e confronteremo vari metodi per implementare la matrice d'accesso.

17.6.1 Tabella globale

La realizzazione più semplice della matrice d'accesso è una tabella globale costituita di un insieme di triple ordinate $\langle \text{dominio}, \text{oggetto}, \text{insieme dei diritti} \rangle$. Ogni volta che si esegue un'operazione M su un oggetto O_j del dominio D_i , si cerca una tripla $\langle D_i, O_j, R_k \rangle$ nella tabella globale, tale che $M \in R_k$. Se si trova questa tripla, l'operazione può continuare, altrimenti si verifica una condizione di eccezione (errore).

Questa tipo di realizzazione ha, però, parecchi svantaggi. Generalmente la tabella è grande e non può essere conservata nella memoria centrale, perciò sono richieste ulteriori operazioni di i/o. Per gestire questa tabella spesso si usano tecniche di memoria virtuale; inoltre è difficile anche trarre vantaggio da speciali raggruppamenti di oggetti o domini. Per esempio, se chiunque può leggere un particolare oggetto, esso deve avere un elemento distinto in ogni dominio.

17.6.2 Liste d'accesso per oggetti

Ogni colonna della matrice d'accesso si può realizzare come una lista d'accesso per un oggetto (Paragrafo 13.4.2). Naturalmente gli elementi vuoti si possono scartare. Per ogni oggetto la lista risultante è composta di coppie ordinate $\langle \text{dominio}, \text{insieme dei diritti} \rangle$ che definiscono tutti i domini il cui insieme di diritti d'accesso per quell'oggetto risulta non vuoto.

Questo metodo si può estendere facilmente definendo una lista più un insieme di *default* di diritti d'accesso. Quando nel dominio D_i si tenta un'operazione M su un oggetto O_j , si cerca un elemento $\langle D_i, R_k \rangle$, con $M \in R_k$, nella lista d'accesso relativa all'oggetto O_j . Se si trova quest'elemento, l'operazione è permessa; altrimenti, si controlla l'insieme di default. Se M si trova in questo insieme, l'accesso è permesso, altrimenti l'accesso viene negato e si verifica una condizione di eccezione. Per motivi di efficienza, conviene controllare prima l'insieme di default e poi cercare nella lista d'accesso.

17.6.3 Liste delle abilitazioni per domini

Anziché associare le colonne della matrice d'accesso agli oggetti formando liste d'accesso, è possibile associare ogni riga della matrice al suo dominio. La lista delle abilitazioni (*capability list*) per un dominio è una lista di oggetti insieme con le operazioni ammesse su quegli oggetti. Un oggetto è spesso rappresentato dal suo nome fisico o indirizzo, detto abilitazione (*capability*). Per eseguire l'operazione M sull'oggetto O_j , il processo esegue l'operazione M , specificando l'abilitazione (puntatore) per l'oggetto O_j come parametro. Il semplice *possesso* dell'abilitazione indica che l'accesso è permesso.

La lista delle abilitazioni è associata a un dominio, ma non è mai direttamente accessibile a un processo che si trova in esecuzione in quel dominio: è un oggetto protetto, mantenuto dal sistema operativo e al quale gli utenti possono accedere solo indirettamente. La protezione basata sulle abilitazioni si fonda sul presupposto che non è mai permessa la migrazione delle abilitazioni in qualsiasi spazio di indirizzi direttamente accessibile a un processo utente, dove queste potrebbero essere modificate. Se tutte le abilitazioni sono sicure, è sicuro anche l'oggetto da esse protetto contro accessi non autorizzati.

Le abilitazioni sono state originariamente proposte come un tipo di puntatore sicuro, per soddisfare la necessità di protezione delle risorse dovuta alla diffusione dei calcolatori multiprogrammati. L'idea di un puntatore intrinsecamente protetto (dal punto di vista dell'utente di un sistema) fornisce la base per una protezione che si può estendere fino a livello delle applicazioni.

Per fornire una protezione intrinseca occorre distinguere le abilitazioni dagli altri oggetti, e interpretarle attraverso una macchina astratta su cui si eseguono programmi di livello superiore. Generalmente le abilitazioni si distinguono dagli altri dati in uno dei due modi seguenti.

- Ogni oggetto ha un'etichetta (*tag*) che ne indica il tipo: abilitazione o dati accessibili. Le etichette non devono essere direttamente accessibili ai programmi applicativi. Per impostare tale limite si può ricorrere al supporto dell'hardware o del firmware. Anche se per distinguere tra abilitazioni e altri oggetti è sufficiente un bit, spesso se ne adoperano di più. Questa estensione permette all'hardware di applicare a tutti gli oggetti etichette indicanti i rispettivi tipi. Quindi l'hardware può distinguere interi, numeri in virgola mobile, puntatori, valori booleani, caratteri, istruzioni, abilitazioni e valori non inizializzati grazie alle rispettive etichette.
- Come alternativa, lo spazio d'indirizzi associato a un programma si può dividere in due parti: una contenente i dati e le istruzioni del programma, accessibile al programma; l'altra, contenente la lista delle abilitazioni, accessibile solo al sistema operativo. Lo spazio di memoria segmentato è un utile supporto di questo metodo.

Sono stati sviluppati parecchi sistemi di protezione basati sulle abilitazioni, brevemente descritti nel Paragrafo 17.10. Anche il sistema operativo Mach, descritto nell'Appendice D (reperibile sulla piattaforma MyLab), fa uso di un tipo di protezione basata sulle abilitazioni.

17.6.4 Meccanismo chiave-serratura

Lo schema chiave-serratura (*lock-key scheme*) rappresenta un compromesso tra le liste d'accesso e le liste di abilitazioni. Ogni oggetto ha una lista di sequenze di bit uniche, chiamate serrature (*lock*); analogamente, ogni dominio ha una lista di sequenze di bit

uniche, chiamate chiavi (*key*). Un processo in esecuzione in un dominio può accedere a un oggetto solo se quel dominio ha una chiave che corrisponde a una delle serrature dell'oggetto.

Come le liste delle abilitazioni, anche la lista delle chiavi per un dominio deve essere gestita dal sistema operativo per conto del dominio. Gli utenti non possono esaminare o modificare direttamente la lista delle chiavi o delle serrature.

17.6.5 Confronto

Come è ovvio, la scelta di una tecnica di implementazione della matrice degli accessi comporta vari trade-off. L'uso di una tabella globale è relativamente semplice; tuttavia, tale tabella può essere piuttosto grande e spesso non può avvantaggiarsi dalla esistenza di gruppi speciali di oggetti o domini. Le liste d'accesso corrispondono direttamente alle necessità degli utenti. Quando un utente crea un oggetto, può specificare quali domini abbiano accesso a quell'oggetto e quali operazioni siano permesse. Tuttavia, poiché le informazioni sui diritti d'accesso per un particolare dominio non sono localizzate, è difficile stabilire l'insieme dei diritti d'accesso per ogni dominio. Inoltre ogni accesso all'oggetto deve essere controllato, il che richiede una ricerca nella lista d'accesso. In un grande sistema con lunghe liste d'accesso, questa ricerca può causare un notevole spreco di tempo.

Le liste delle abilitazioni non corrispondono direttamente alle necessità degli utenti, ma sono utili per localizzare le informazioni per un dato processo. Un processo che tenti un accesso deve presentare la relativa abilitazione, quindi il sistema di protezione deve verificare soltanto che l'abilitazione sia valida. Tuttavia la revoca delle abilitazioni può essere inefficiente; questo problema è trattato nel Paragrafo 17.7.

Il meccanismo chiave-serratura rappresenta un compromesso tra questi due schemi. Il meccanismo può essere efficace e flessibile, a seconda della lunghezza delle chiavi, che possono essere passate liberamente da dominio a dominio. Inoltre i privilegi d'accesso si possono revocare in modo semplice ed efficace modificando alcune chiavi associate all'oggetto; anche questo problema è trattato nel Paragrafo 17.7.

La maggior parte dei sistemi adopera una combinazione di liste d'accesso e liste di abilitazioni. Quando un processo tenta per la prima volta di accedere a un oggetto, si fa una ricerca nella lista d'accesso. Se l'accesso viene negato, si verifica una condizione di eccezione, altrimenti si crea un'abilitazione che si associa al processo. I riferimenti successivi si servono dell'abilitazione per dimostrare rapidamente che l'accesso è consentito. Dopo l'ultimo accesso, l'abilitazione viene distrutta. Questa strategia è usata nel sistema multics e nel sistema cal.

Come esempio di come opera questa strategia, si consideri un file system in cui a ogni file è associata una lista d'accesso. Quando un processo apre un file, si fa una ricerca nella directory per trovare il file, si controlla il permesso d'accesso e si assegnano i buffer per l'i/o. Tutte queste informazioni si registrano in un nuovo elemento della tabella dei file associata al processo. L'operazione riporta un indice in questa tabella per il file appena aperto, tramite il quale si compiono tutte le operazioni sul file. Quindi l'elemento della tabella dei file punta al file e ai suoi buffer. Quando il file viene chiuso, si cancella l'elemento della tabella dei file. Poiché la tabella dei file è mantenuta dal sistema operativo, gli utenti non possono alterarla accidentalmente, quindi gli utenti possono accedere ai soli file che sono stati aperti. Poiché l'accesso viene controllato al momento dell'apertura del file, la protezione è assicurata. Nel sistema operativo unix si adopera questa strategia.

Il diritto d'accesso si deve ancora controllare per ogni accesso e l'elemento della tabella dei file contiene l'abilitazione per le sole operazioni ammesse. Se si apre un file per la lettura, nell'elemento della tabella dei file viene inserita un'abilitazione d'accesso alla lettura. Se si tenta di scrivere in quel file, il sistema rileva questa violazione della protezione confrontando l'operazione richiesta con l'abilitazione presente nell'elemento della tabella dei file.

17.7 Revoca dei diritti d'accesso

In un sistema di protezione dinamica talvolta può essere necessario revocare i diritti d'accesso a oggetti condivisi da diversi utenti. A proposito della revoca si possono presentare diverse questioni.

- Immediata o ritardata. Occorre stabilire se la revoca ha effetto immediatamente o con ritardo. Se la revoca è ritardata, occorre stabilire quando avverrà.
- Selettiva o generale. Quando si revoca un diritto d'accesso a un oggetto, occorre stabilire se la revoca vale per *tutti* gli utenti che hanno un diritto d'accesso a quell'oggetto, oppure se si può specificare un gruppo di utenti a cui si debbano revocare i diritti d'accesso.
- Parziale o totale. Occorre stabilire se si può revocare un sottoinsieme di diritti associati a un oggetto, oppure se si devono revocare tutti i diritti d'accesso a quest'oggetto.
- Temporanea o permanente. Occorre stabilire se l'accesso si può revocare in modo permanente, cioè il diritto d'accesso non sarà più disponibile, oppure se può essere nuovamente ottenuto.

Disponendo di un sistema basato su liste d'accesso, effettuare una revoca è facile. Si cercano i diritti d'accesso da revocare nella lista d'accesso, e li si cancellano dalla lista. La revoca è immediata e può essere generale o selettiva, totale o parziale, permanente o temporanea.

La revoca delle abilitazioni, invece, è molto più difficile. Poiché le abilitazioni sono distribuite su tutto il sistema, occorre prima trovarle e poi revocarle. Tra gli schemi che realizzano la revoca delle abilitazioni ci sono i seguenti.

- Riacquisizione. Le abilitazioni vengono cancellate periodicamente da ogni dominio. Se un processo intende usare un'abilitazione, può scoprire che quell'abilitazione è stata cancellata. Il processo allora può tentare di riacquisirla. Se l'accesso è stato revocato, il processo non è più in grado di riacquisire l'abilitazione.
- Puntatori all'indietro. Insieme a ciascun oggetto si conserva una lista di puntatori a tutte le abilitazioni a esso associate. Quando si richiede una revoca, si possono seguire questi puntatori, modificando le abilitazioni secondo le necessità. Questo schema era adottato nel sistema multics, ed è abbastanza generale, anche se la sua realizzazione è onerosa.
- Riferimento indiretto. Le abilitazioni non puntano direttamente agli oggetti. Ogni abilitazione punta all'elemento di una tabella globale che a sua volta punta all'oggetto. La revoca si realizza cercando nella tabella globale l'elemento desiderato e cancellandolo. Quando si tenta l'accesso, si riscontra che l'abilitazione punta a un elemento illegale della tabella. Gli elementi della tabella si possono riutilizzare senza difficoltà per altre abilitazioni, poiché sia l'abilitazione sia l'elemento della tabella contengono il nome unico dell'oggetto. L'oggetto dell'abilitazione e dell'elemento nella tabella devono corrispondere. Questo schema, adottato nel sistema cal, non permette la revoca selettiva.
- Chiavi. Una chiave è una sequenza unica di bit associabile a ogni abilitazione. Tale chiave viene definita al momento della creazione dell'abilitazione e non può essere modificata né esaminata dal processo proprietario dell'abilitazione stessa. Esiste una chiave principale (*master key*) associata a ogni oggetto, che si può definire o sostituire con l'operazione *set-key*. Quando si crea un'abilitazione, a questa si associa il valore corrente della chiave principale. Quando l'abilitazione viene esercitata, si confronta la sua chiave con la chiave principale. Se le due chiavi coincidono, l'operazione può continuare, altrimenti si verifica una condizione di eccezione. La revoca sostituisce la chiave principale con un valore nuovo tramite l'operazione *set-key*, che invalida tutte le abilitazioni precedenti per quest'oggetto. Questo schema non permette la revoca selettiva, poiché a ogni oggetto è associata solo una chiave principale. Se a ogni oggetto si associa una lista di chiavi, si può realizzare la revoca selettiva. Infine, tutte le chiavi si possono raggruppare in una tabella globale di chiavi. Un'abilitazione è valida solo se la sua chiave coincide con una delle chiavi della tabella globale. La revoca si realizza rimuovendo dalla tabella la chiave coincidente. In questo schema una chiave si può associare a più oggetti, e più chiavi si possono associare a ciascun oggetto, offrendo la massima flessibilità. Negli schemi basati sulle chiavi, le operazioni di definizione, inserimento in liste e cancellazione dalle liste delle chiavi stesse non devono essere a disposizione di tutti gli utenti. In particolare, è ragionevole permettere soltanto al proprietario di un oggetto di impostare le chiavi per quell'oggetto. Questa scelta, in ogni modo, riguarda un criterio (*policy*), che il sistema di protezione può realizzare ma non deve definire.

17.8 Controllo dell'accesso basato sui ruoli

Nel Paragrafo 13.4.2 abbiamo descritto l'uso dei controlli dell'accesso ai file in un file system. A tutti i file e le directory è assegnato un proprietario, un gruppo o, in alcuni casi, una lista di utenti, e a ciascuna di tali entità sono assegnate informazioni di controllo dell'accesso. Una funzionalità simile può essere realizzata per altri aspetti del sistema; Solaris 10 e successive versioni ne sono un buon esempio.

L'idea è quella di anticipare la protezione offerta dal sistema operativo applicando il principio del minor privilegio tramite il controllo dell'accesso basato sui ruoli (*role-based access control, rbac*). Questa funzionalità si basa sui privilegi. Si dice privilegio il diritto di eseguire una chiamata di sistema o di sfruttare un'opzione di tale chiamata (come l'apertura di un file con accesso in scrittura). I privilegi possono essere assegnati ai processi, limitandoli al minimo indispensabile per svolgere il compito cui sono preposti. Inoltre, privilegi e programmi possono essere assegnati sulla base di ruoli. Un utente può avere un ruolo assegnato o può assumere un ruolo tramite l'uso di una password. In questo modo, un utente può assumere un ruolo che gli attribuisce un certo privilegio; ciò permette all'utente di eseguire un programma per portare a termine un compito specifico, come rappresentato nella Figura 17.10. Questa organizzazione attenua i rischi per la sicurezza del sistema, in particolare quelli legati ai superuser e ai programmi e setuid.

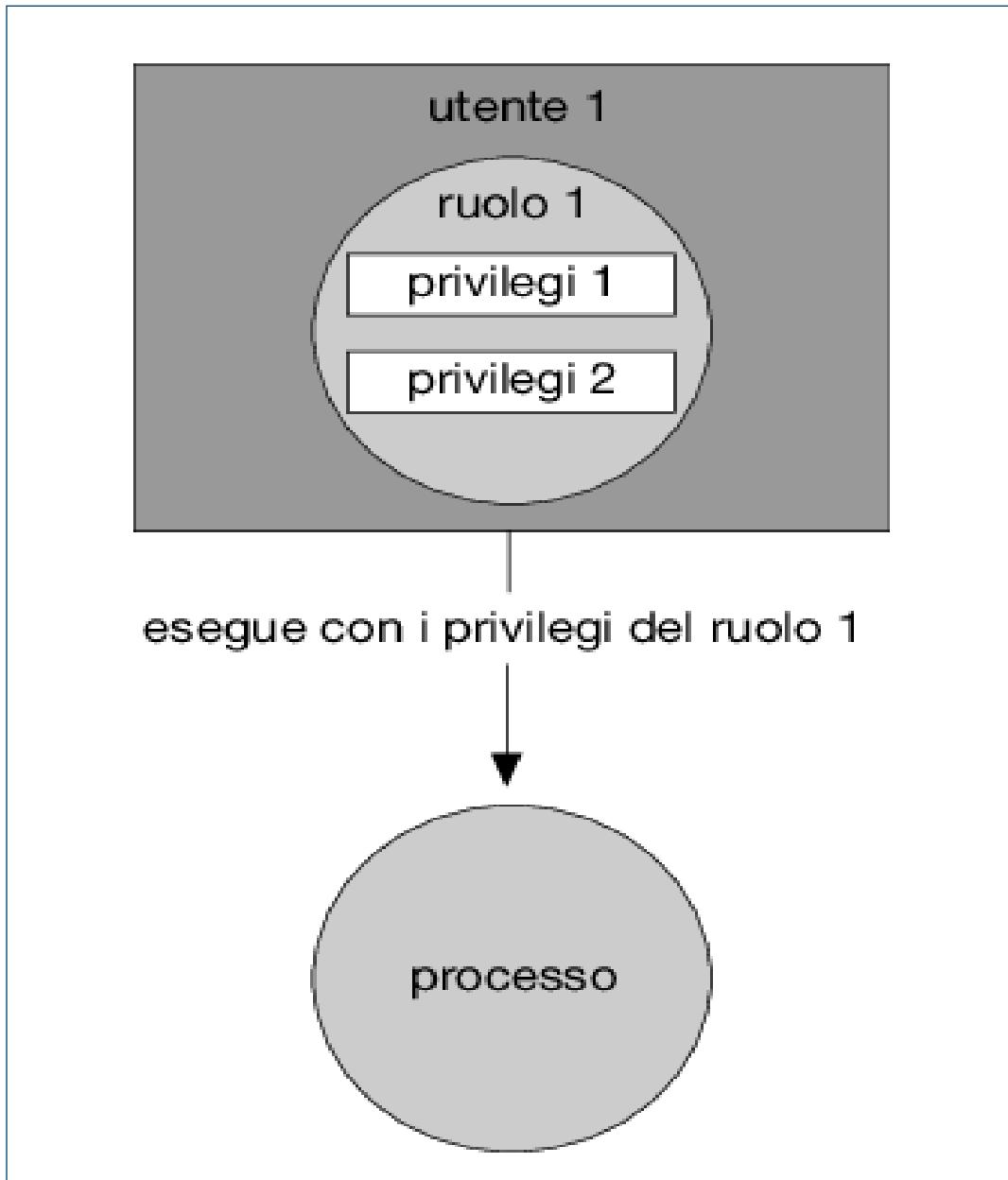


Figura 17.10 Controllo dell'accesso basato sui ruoli in Solaris 10.

Si noti l'analogia di questa tecnica con la matrice d'accesso, trattata nel Paragrafo 17.5. Questa relazione sarà esplorata più compiutamente negli esercizi che concludono il capitolo.

17.9 Controllo obbligatorio dell'accesso (MAC)

I sistemi operativi hanno tradizionalmente utilizzato il controllo discrezionale di accesso (dac) come mezzo per limitare l'accesso ai file e agli altri oggetti del sistema. Con il dac, l'accesso è controllato in base alle identità dei singoli utenti o gruppi. In un sistema basato su unix, dac assume la forma di permessi di file (impostabili con `chmod`, `chown` e `chgrp`), mentre in Windows (e in alcune varianti di unix) è consentita una granularità più fine mediante le liste di controllo degli accessi (acl).

I dac, tuttavia, si sono rivelati insufficienti nel corso degli anni. Una debolezza fondamentale risiede nella loro natura discrezionale, che consente al proprietario di una risorsa di impostare o modificare le sue autorizzazioni. Un altro punto debole è l'accesso illimitato consentito all'utente amministratore o root. Come abbiamo visto, questo meccanismo può lasciare il sistema vulnerabile ad attacchi sia accidentali sia dolosi e non fornisce alcuna difesa quando gli hacker ottengono i privilegi di root.

È nata quindi la necessità di una protezione più forte, introdotta sotto forma di controllo obbligatorio degli accessi (mac). Il mac viene applicato come una politica di sistema che nemmeno l'utente root può modificare (a meno che la politica adottata non consenta esplicitamente modifiche o il sistema venga riavviato, di solito in una configurazione alternativa). Le restrizioni imposte dalle regole della politica mac sono più forti delle abilitazioni dell'utente root e possono essere utilizzate per rendere le risorse inaccessibili a chiunque tranne ai proprietari designati.

Tutti i moderni sistemi operativi forniscono mac e dac, anche se le implementazioni differiscono. Solaris è stato tra i primi a introdurre il mac, che faceva parte di Trusted Solaris (2.5). Freebsd ha reso mac parte della sua implementazione Trustedbsd (Freebsd 5.0). L'implementazione di Freebsd è stata adottata da Apple in macos 10.5 ed è servita come un substrato su cui è stata implementata la maggior parte delle funzionalità di sicurezza di mac e ios. L'implementazione del mac di Linux fa parte del progetto selinux, ideato dalla nsa, ed è stato integrato nella maggior parte delle distribuzioni. Microsoft Windows ha seguito la tendenza introducendo Mandatory Integrity Control in Windows Vista.

Il cuore del mac è il concetto di etichette. Un'etichetta è un identificatore (di solito una stringa) assegnato a un oggetto (file, dispositivi e altro). Le etichette possono anche essere applicate ai soggetti (agli attori, per esempio i processi) quando richiedono di eseguire operazioni sugli oggetti. Quando il sistema operativo deve soddisfare tali richieste, esegue prima i controlli definiti in una politica che stabilisce se un determinato soggetto titolare dell'etichetta è autorizzato a eseguire l'operazione sull'oggetto etichettato.

Si consideri, per esempio, questo semplice insieme di etichette, ordinato in base al livello di privilegio: "non classificato", "segreto" e "top secret". Un utente con autorizzazione "segreto" sarà in grado di creare processi etichettati come "segreti", che avranno quindi accesso a file "non classificati" e "segreti", ma non a file "top secret". L'utente e i suoi processi non saranno nemmeno a conoscenza dell'esistenza di file "top secret", dal momento che il sistema operativo li filtrerebbe in tutte le operazioni sui file (per esempio, non verrebbero visualizzati quando si elencano i contenuti delle directory). Analogamente, i processi utente verrebbero protetti in questo modo, così che un processo "non classificato" non sarebbe in grado di vedere un processo "segreto" (o "top secret") o eseguire richieste ipc verso di esso. Le etichette mac costituiscono dunque un'implementazione della matrice di accesso descritta in precedenza.

17.10 Sistemi basati su abilitazioni

Il concetto di protezione basata sulle abilitazioni (capabilities) fu introdotto all'inizio degli anni '70 e due dei primi sistemi furono Hydra e cap. Nessuno dei due è stato molto usato, ma entrambi sono interessanti banchi di prova delle teorie sulla protezione. Per maggiori dettagli su questi sistemi si vedano i Paragrafi A.14.1 e A.14.2 (Appendice A, reperibile sulla piattaforma MyLab). Nel seguito vengono considerati due moderni approcci all'abilitazione.

17.10.1 Abilitazioni di Linux

Linux usa le abilitazioni per affrontare i limiti del modello unix, che abbiamo descritto in precedenza. Lo standard posix ha introdotto le abilitazioni in posix 1003.1e. Sebbene posix.1e sia stato ritirato, Linux ha adottato rapidamente le abilitazioni nella versione 2.2 e ha continuato a svilupparle.

In sostanza, le abilitazioni di Linux "spezzettano" i poteri della root in aree distinte, ciascuna rappresentata da un bit in una maschera di bit, come mostrato nella Figura 17.11. Il controllo accurato sulle operazioni con privilegi può essere ottenuto commutando i bit nella maschera.

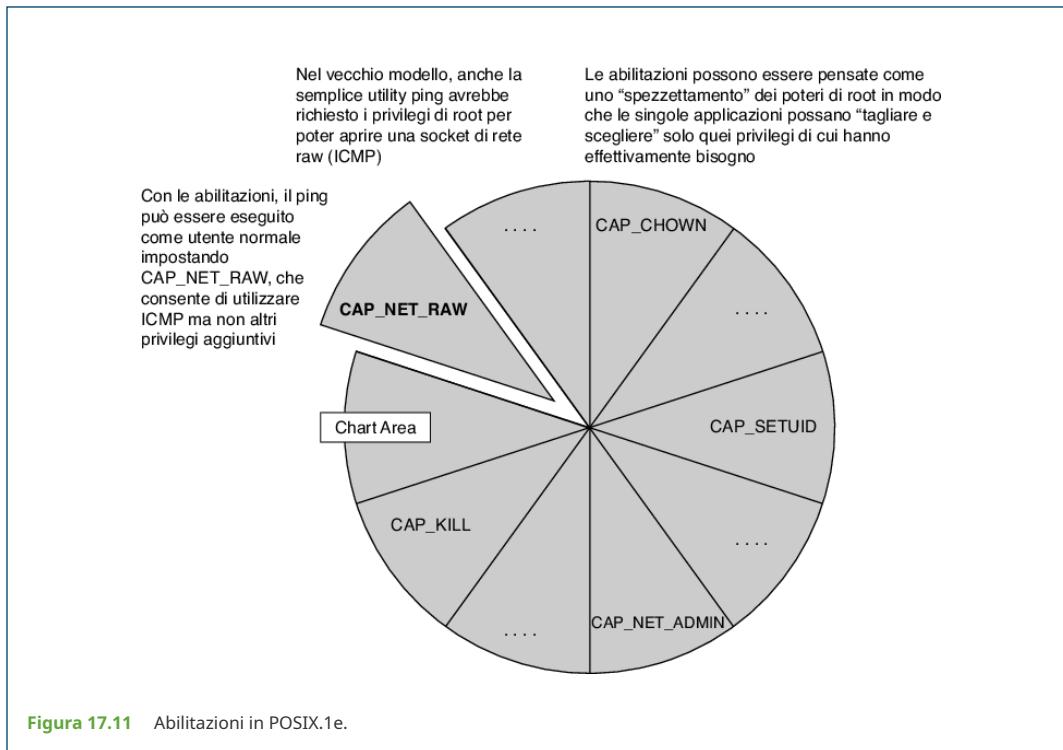


Figura 17.11 Abilitazioni in POSIX.1e.

In pratica vengono utilizzate tre maschere di bit per indicare le abilitazioni *consentite*, *effettive* ed *ereditabili*. Le maschere di bit possono essere applicate sulla base dei processi o dei thread. Inoltre, una volta revocate, le abilitazioni non possono essere riacquisite. Solitamente un processo o un thread inizia con l'insieme completo di abilitazioni consentite e restringe volontariamente questo insieme durante l'esecuzione. Per esempio, dopo aver aperto una porta di rete, un thread può rimuovere tale abilitazione in modo da rendere impossibile l'apertura di ulteriori porte.

Potete facilmente intuire che le abilitazioni sono un'implementazione diretta del principio del minimo privilegio. Come spiegato in precedenza, questo principio di sicurezza stabilisce che a un'applicazione o a un utente debbano essere assegnati esclusivamente i diritti che sono necessari per il suo normale funzionamento.

Anche Android (basato su Linux) utilizza le abilitazioni per consentire ai processi di sistema (in particolare, ai "server di sistema"), di evitare la proprietà della root, abilitando selettivamente solo le operazioni richieste.

Il modello di abilitazioni di Linux è un grande miglioramento rispetto al modello unix tradizionale, ma è ancora poco flessibile. Per prima cosa, l'utilizzo di una bitmap con un bit per ciascuna abilitazione rende impossibile aggiungere abilitazioni in modo dinamico e richiede la ricompilazione del kernel per ogni aggiunta. Inoltre, la funzione si applica solo alle abilitazioni imposte dal kernel.

17.10.2 Autorizzazioni di Darwin

La protezione del sistema di Apple si basa sulle autorizzazioni (*entitlement*). Le autorizzazioni sono permessi dichiarativi e consistono in un elenco xml di proprietà che indica quali autorizzazioni sono dichiarate come necessarie dal programma (si veda la Figura 17.12). Quando il processo tenta un'operazione privilegiata (nella figura, caricare un'estensione del kernel), le sue autorizzazioni vengono verificate e solo se sono presenti le autorizzazioni necessarie l'operazione è consentita.

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>com.apple.private.kernel.get-kext-info
  <true/>
  <key>com.apple.rootless.kext-management
  <true/>
</dict>
</plist>
```

Figura 17.12 Autorizzazioni di Apple Darwin.

Per impedire ai programmi di richiedere arbitrariamente un'autorizzazione, Apple incorpora le autorizzazioni nella firma del codice (come spiegato nel Paragrafo 17.11.4). Una volta caricato, un processo non ha modo di accedere alla sua firma, mentre gli altri processi (e il kernel) possono facilmente interrogare la firma, e in particolare le autorizzazioni. La verifica di un'autorizzazione è quindi una semplice operazione di corrispondenza tra stringhe. In questo modo, solo le app verificabili e autenticate possono richiedere autorizzazioni. Tutti le autorizzazioni di sistema (`com.apple.*`) sono ulteriormente limitate ai binari di Apple.

17.11 Altri metodi per il miglioramento della protezione

Siccome la battaglia per proteggere i sistemi da danni accidentali e dolosi diventa sempre più dura, i progettisti del sistema operativo stanno implementando ulteriori tipologie di meccanismi di protezione, agendo su più livelli. Questo paragrafo esamina alcuni importanti miglioramenti della protezione realmente utilizzati.

17.11.1 Protezione dell'integrità del sistema

Apple ha introdotto in macos 10.11 un nuovo meccanismo di protezione chiamato System Integrity Protection (sip). I sistemi operativi basati su Darwin utilizzano sip per limitare l'accesso ai file e alle risorse di sistema in modo tale che non possano essere manomessi nemmeno dall'utente root. sip utilizza attributi estesi sui file per contrassegnarli come limitati e protegge ulteriormente i file binari del sistema, in modo che non possano essere sottoposti a debug, né esaminati internamente, né tantomeno manomessi. Ancora più importante è il fatto che sono consentite solo le estensioni del kernel firmate; inoltre, sip può essere configurato per consentire solamente eseguibili con codice firmato.

Sotto sip, l'utente root può fare molto meno di quanto prima gli era permesso, sebbene sia ancora l'utente più potente del sistema. L'utente root può comunque gestire i file di altri utenti, nonché installare e rimuovere programmi, ma non può in alcun modo sostituire o modificare i componenti del sistema operativo. sip è implementato come uno schermo globale e ineludibile su tutti i processi, con le sole eccezioni di file eseguibili di sistema (per esempio, `fsck` o `kextload`, come mostrato nella Figura 17.12), che hanno autorizzazioni specifiche per le operazioni necessarie per assolvere le loro funzioni.

17.11.2 Filtraggio delle chiamate di sistema

Ricordiamo dal Capitolo 2 che i sistemi monolitici collocano tutte le funzionalità del kernel in un singolo file che viene eseguito in un unico spazio di indirizzamento. Di solito, i kernel dei sistemi operativi general-purpose sono monolitici e pertanto sono implicitamente considerati affidabili. Il confine dell'affidabilità, quindi, dipende dalla modalità kernel e dalla modalità utente, a livello di sistema. Possiamo ragionevolmente presumere che qualsiasi tentativo di compromettere l'integrità del sistema verrà effettuato dalla modalità utente per mezzo di una chiamata di sistema. Per esempio, un utente malintenzionato può tentare di ottenere l'accesso sfruttando una chiamata di sistema non protetta.

È quindi imperativo implementare una qualche forma di filtraggio delle chiamate di sistema. Per fare ciò possiamo aggiungere codice al kernel per eseguire un'ispezione ai gate della chiamata di sistema, limitando una chiamata a un sottoinsieme di chiamate di sistema ritenute sicure o necessarie per la funzionalità di quel chiamante. È possibile creare profili di chiamata di sistema specifici per singoli processi. Il meccanismo di Linux seccomp-bpf fa proprio questo, sfruttando il linguaggio Berkeley Packet Filter per caricare un profilo personalizzato attraverso la chiamata di sistema proprietaria `prctl` di Linux. Questo filtraggio è volontario, ma può essere imposto efficacemente se richiamato da una libreria di run-time durante l'inizializzazione o dal loader stesso prima che trasferisca il controllo al punto di ingresso del programma.

Una seconda forma di filtraggio delle chiamate di sistema va ancora più in profondità e controlla gli argomenti di ogni chiamata di sistema. Questa forma di protezione è considerata molto più forte, perché anche le chiamate di sistema apparentemente benigne possono ospitare gravi vulnerabilità. È stato il caso della chiamata di sistema fast mutex (`futex`) di Linux: una race condition nella sua implementazione ha portato a una sovrascrittura della memoria del kernel controllata dagli aggressori e alla compromissione dell'intero sistema. I mutex sono una componente fondamentale del multitasking e quindi la chiamata di sistema non può essere completamente filtrata.

Una sfida a cui si va incontro con entrambi gli approcci consiste nel mantenerli il più flessibili possibile, evitando allo stesso tempo la necessità di ricostruire il kernel ogni volta che sono richieste modifiche o nuovi filtri, un evento comune a causa delle diverse esigenze dei diversi processi. La flessibilità è particolarmente importante data la natura imprevedibile delle vulnerabilità: ogni giorno ne vengono scoperte di nuove e queste possono essere immediatamente sfruttate dagli aggressori.

Un approccio per affrontare questa sfida è quello di disaccoppiare l'implementazione del filtro dal kernel stesso. Il kernel deve contenere solo una serie di callout, che possono quindi essere implementati in un driver specializzato (Windows), in un modulo del kernel (Linux) o in un'estensione (Darwin). Poiché la logica di filtraggio è fornita da un componente modulare esterno, essa può essere modificata indipendentemente dal kernel. Questo componente fa comunque uso di un linguaggio di profiling specializzato ed è dotato di un interprete o di un parser incorporato. È pertanto possibile disaccoppiare dal codice il profilo stesso e fornire un profilo modificabile leggibile dall'uomo, oltre a semplificare ulteriormente gli aggiornamenti. Il componente di filtraggio può inoltre chiamare un processo demone attendibile in modalità utente che assista nella logica di validazione.

17.11.3 Sandboxing

Il meccanismo di sandboxing consiste nell'esecuzione di processi in ambienti che limitino le loro funzionalità. In un sistema standard, un processo viene eseguito con le credenziali dell'utente che l'ha avviato e ha accesso a tutte le risorse a cui l'utente può accedere. Se eseguito con privilegi di sistema, come root, il processo può fare letteralmente qualsiasi cosa nel sistema. Quasi sempre, però, un processo non ha bisogno dei privilegi completi di utente o di sistema. Per esempio, un elaboratore di testi deve accettare le connessioni di rete? Un servizio di rete che fornisce l'ora del giorno deve accedere ai file al di fuori di un insieme specifico?

Il termine sandboxing si riferisce alla pratica di applicare rigorose limitazioni a un processo. Piuttosto che dare a quel processo il set completo di chiamate di sistema che i suoi privilegi consentirebbero, imponiamo un insieme irremovibile di restrizioni sul processo nelle fasi iniziali del suo avvio, molto prima dell'esecuzione della sua funzione `main()` e spesso fin dalla sua creazione per mezzo di una chiamata di sistema `fork`. Il processo viene quindi reso incapace di eseguire operazioni al di fuori dell'insieme consentito. In questo modo è possibile impedire che il processo comunichi con qualsiasi altro componente del sistema, portando a una stretta suddivisione del sistema in compartimenti in grado di mitigare i danni anche quando il processo è compromesso.

Esistono numerosi approcci allo sandboxing. Java e .net, per esempio, impongono restrizioni sandbox a livello della macchina virtuale. Altri sistemi applicano meccanismi di sandboxing come parte della loro politica di controllo obbligatorio degli accessi (mac). Un esempio è Android, che si basa su una politica selinux arricchita con etichette specifiche per le proprietà di sistema e gli endpoint dei servizi.

Lo sandboxing può anche essere implementato come una combinazione di più meccanismi. Android ha trovato selinux utile, ma carente, perché non può limitare in modo efficace le singole chiamate di sistema. Le ultime versioni di Android ("Nougat" e "O") utilizzano un meccanismo sottostante a Linux chiamato seccomp-bpf, menzionato in precedenza, per applicare restrizioni alle chiamate di sistema attraverso l'uso di una chiamata di sistema specializzata. La libreria di run-time C di Android ("Bionic") effettua questa chiamata di sistema per imporre restrizioni su tutti i processi Android e sulle applicazioni di terze parti.

Tra i principali fornitori di sistemi operativi, Apple è stata la prima a implementare un meccanismo di sandboxing, apparso in macos 10.5 ("Tiger") con il nome di "Seatbelt" ("Cintura di sicurezza"). La "seatbelt" era opzionale, e poteva essere utilizzata dalle applicazioni, che non erano comunque costrette a farlo. La sandbox di Apple era basata su profili dinamici scritti nel linguaggio Scheme, che fornivano la capacità di controllare non solo quali operazioni dovevano essere consentite o bloccate, ma anche i loro argomenti. Questa funzionalità ha consentito a Apple di creare diversi profili personalizzati per ogni file eseguibile sul sistema, una pratica che continua ancora oggi. La Figura 17.13 mostra un esempio di profilo.

```
(version 1)
(deny default)
(allow file-chroot)
(allow file-read-metadata (literal "/var"))
(allow sysctl-read)
(allow mach-per-user-lookup)
(allow mach-lookup)
(global-name "com.apple.system.logger")
```

Figura 17.13 Un profilo sandbox di un demone MacOS che nega la maggior parte delle operazioni.

Il meccanismo di sandboxing di Apple si è evoluto considerevolmente rispetto ai suoi esordi. Ora è utilizzato nelle varianti ios, dove serve (insieme alla firma del codice) come principale protezione contro il codice di terze parti non attendibile. In ios e a partire da macos 10.8, la sandbox macos è obbligatoria e viene applicata automaticamente a tutte le app scaricate dal Mac-store. Di recente, come già accennato, Apple ha adottato la System Integrity Protection (sip), utilizzata in macos 10.11 e successive versioni. sip è, in effetti, un "profilo della piattaforma" a livello di sistema, che Apple impone su tutti i processi all'avvio del sistema. Solo i processi autorizzati possono eseguire operazioni privilegiate; questi processi hanno il codice firmato da Apple e quindi affidabili.

17.11.4 Firma del codice

Come può un sistema "fidarsi" di un programma o di uno script? In genere, se l'elemento è parte del sistema operativo viene considerato attendibile, ma che cosa succede se subisce modifiche? Se viene modificato da un aggiornamento di sistema è di nuovo considerato attendibile, ma in caso contrario non dovrebbe essere eseguibile o dovrebbe richiedere un'autorizzazione speciale (da parte dell'utente o dell'amministratore) prima dell'esecuzione. Strumenti di terze parti, commerciali o meno, sono più difficili da giudicare. Come possiamo essere sicuri che lo strumento non sia stato modificato nel suo percorso da dove è stato creato verso i nostri sistemi?

Attualmente, la firma del codice è lo strumento di protezione migliore per risolvere questi problemi. La firma del codice è la firma digitale di programmi ed eseguibili che serve per confermare che non sono stati modificati da quando l'autore li ha creati. Per verificare l'integrità e l'autenticità viene utilizzato un hash (Paragrafo 16.4.1.3). La firma del codice è utilizzata sia per le distribuzioni dei sistemi operativi, sia per le patch, sia per gli strumenti di terze parti. Alcuni sistemi operativi, tra cui ios, Windows e macos, rifiutano di eseguire i programmi che non superano il controllo della firma. La firma del codice può migliorare la funzionalità del sistema anche in altri modi. Per esempio, Apple può disabilitare tutti i programmi scritti per una versione di ios ormai obsoleta, interrompendo la firma di tali programmi quando vengono scaricati dall'App Store.

17.12 Protezione basata sul linguaggio

Il grado di protezione fornito negli attuali sistemi elaborativi è di solito ottenuto attraverso il kernel del sistema operativo, che si occupa di controllare e convalidare ogni tentativo d'accesso a una risorsa protetta. Poiché una completa convalida degli accessi è potenzialmente una fonte di notevole sovraccarico, o si dispone del supporto dell'hardware per ridurre il costo di ogni convalida, o si deve accettare un compromesso rispetto agli obiettivi della protezione. Se la flessibilità di realizzazione dei criteri di protezione è limitata dai meccanismi di supporto disponibili, o se gli ambienti di protezione sono resi più grandi di quanto è necessario per assicurare una maggiore efficienza, soddisfare tutti questi obiettivi è difficile.

Gli scopi della protezione sono stati perfezionati con l'aumentare della complessità dei sistemi operativi e soprattutto con il tentativo di fornire interfacce utente di livello superiore. I progettisti dei sistemi di protezione si sono basati in modo determinante su idee nate nell'ambito dei linguaggi di programmazione e, soprattutto, sui concetti di tipo di dati astratti e di oggetto. Attualmente i sistemi di protezione non riguardano solo l'identità di una risorsa a cui si tenta di accedere, ma anche la natura funzionale di quell'accesso. Nei sistemi di protezione più recenti la azione di controllo sulle funzioni da invocare va oltre un insieme di funzioni definite dal sistema, come gli ordinari metodi per l'accesso ai file, per comprendere anche funzioni definibili dagli utenti.

Anche i criteri concernenti l'uso delle risorse variano secondo l'applicazione e, con il passare del tempo, possono essere soggetti a cambiamenti. Per questi motivi la protezione non può più essere considerata come un esclusivo interesse del progettista di un sistema operativo; dovrebbe essere uno strumento disponibile al progettista di applicazioni per proteggere le risorse di un sottosistema d'applicazione da manomissioni o effetti dovuti a errori.

17.12.1 Controllo realizzato dal compilatore

A questo punto entrano in gioco i linguaggi di programmazione. Specificare il controllo degli accessi desiderato per una risorsa condivisa significa fare un'asserzione dichiarativa su tale risorsa. Questo tipo di dichiarazione si può integrare in un linguaggio di programmazione estendendone la funzione di tipizzazione. Quando insieme alla tipizzazione dei dati si dichiara anche la protezione, i progettisti dei sottosistemi possono specificare i propri requisiti di protezione, come anche la necessità di utilizzare altre risorse del sistema. Tali specificazioni si dovrebbero fornire direttamente nella fase di stesura dei programmi, e nello stesso linguaggio in cui si scrivono i programmi. Questo metodo presenta importanti vantaggi:

1. le necessità di protezione si devono semplicemente dichiarare e non programmare come una sequenza di chiamate di procedure di un sistema operativo;
2. i requisiti di protezione si possono stabilire indipendentemente dalle funzioni fornite da uno specifico sistema operativo;
3. i progettisti dei sottosistemi non devono fornire i meccanismi di controllo dei criteri;
4. la notazione dichiarativa è naturale, perché i privilegi d'accesso sono strettamente connessi al concetto linguistico di tipo di dati.

L'implementazione di un linguaggio di programmazione può fornire diverse tecniche per imporre protezioni, ma ciascuna di loro deve dipendere in qualche misura dalle funzioni di supporto offerte dalla macchina sottostante e dal suo sistema operativo. Si supponga, per esempio, che un linguaggio sia impiegato per generare codice da eseguire sul sistema Cambridge cap (Paragrafo A.14.2, Appendice A reperibile sulla piattaforma MyLab). In questo sistema ogni riferimento alla memoria effettuato sull'hardware sottostante avviene indirettamente tramite abilitazioni. Questo limite impedisce a un processo qualsiasi di accedere a una risorsa esterna al proprio ambiente di protezione. Tuttavia un programma può imporre limitazioni arbitrarie su come una risorsa può essere usata durante l'esecuzione di un particolare segmento di codice. Tali limiti si possono realizzare in modo pressoché immediato ricorrendo alle abilitazioni software offerte da cap. L'implementazione di un linguaggio di programmazione potrebbe fornire procedure protette standard per interpretare le abilitazioni software che realizzano i criteri di protezione specificabili nel linguaggio stesso. Questo schema permette ai programmati di specificare i criteri di protezione, senza costringerli a occuparsi dei dettagli riguardanti i relativi meccanismi di supporto.

Anche se un sistema non offre un kernel di protezione potente come quelli di Hydra (Paragrafo A.14.1 reperibile sulla piattaforma MyLab) o cap, esistono meccanismi che permettono la realizzazione delle specifiche di protezione presenti in un linguaggio di programmazione. La differenza principale è dovuta al fatto che la *sicurezza* di questa protezione non è grande quanto quella gestita da un kernel di protezione, poiché il meccanismo si deve basare su un maggior numero di assunzioni sullo stato operativo del sistema. Un compilatore può separare i riferimenti per i quali non è possibile avere violazioni di protezione da quelli per i quali la violazione può avvenire, e trattarli in modi diversi. La sicurezza offerta da questo tipo di protezione si fonda sul presupposto che il codice generato dal compilatore non venga modificato prima o durante la sua esecuzione.

Di seguito sono elencati i vantaggi dei meccanismi basati esclusivamente su un kernel, rispetto a quelli che si hanno con i meccanismi forniti da un compilatore.

- Sicurezza. Il controllo effettuato da un kernel offre un grado di sicurezza del sistema di protezione maggiore di quello offerto dalla generazione, da parte di un compilatore, di codice per il controllo della protezione. In uno schema supportato dal compilatore, la sicurezza è basata sulla correttezza del traduttore, su qualche meccanismo di gestione della memoria che proteggi i segmenti dai quali si esegue il codice compilato e, in ultima analisi, sulla sicurezza dei file dai quali si carica il programma. Alcune di queste considerazioni valgono anche per un kernel di protezione supportato esclusivamente dal software, ma in questo caso in misura minore poiché il kernel può risiedere in segmenti fissati di memoria fisica e può essere caricato solo da un file designato. In un sistema di abilitazioni con etichette, in cui tutto il calcolo degli indirizzi è eseguito dall'hardware o da un microprogramma fisso, si può avere una sicurezza ancora maggiore. La protezione basata sull'hardware è anche relativamente immune dalle violazioni della protezione verificabili a causa di malfunzionamenti sia fisici sia logici.
- Flessibilità. La flessibilità di un kernel di protezione ha alcuni limiti per quel che riguarda la realizzazione di criteri di protezione definiti dagli utenti, anche se può fornire al sistema le funzioni necessarie per l'imposizione dei propri criteri. Con un linguaggio di programmazione i criteri di protezione si possono dichiarare, così come si può fornire il controllo richiesto tramite l'implementazione del linguaggio. Se un linguaggio non offre una sufficiente flessibilità, si può estendere o sostituire, interferendo con il sistema in servizio meno di quanto non accadrebbe modificando il kernel del sistema operativo.

- Efficienza. La maggior efficienza si ha quando la protezione è gestita direttamente dall'hardware (o dal microcodice). Il controllo basato sul linguaggio ha il vantaggio di poter verificare il controllo degli accessi in maniera statica nella fase di compilazione. Inoltre, poiché un compilatore intelligente può adattare il meccanismo di controllo alla specifica necessità, spesso si può evitare l'overhead fisso delle chiamate del kernel.

Riepilogando, la specificazione della protezione in un linguaggio di programmazione permette la descrizione ad alto livello di criteri di allocazione e uso di risorse. L'implementazione di un linguaggio di programmazione può fornire gli strumenti per la realizzazione della protezione quando non è disponibile il controllo automatico gestito dall'hardware; inoltre può interpretare le indicazioni di protezione per generare chiamate a qualsiasi sistema di protezione sia fornito dall'hardware e dal sistema operativo.

Un modo per rendere disponibile la protezione ai programmi applicativi prevede l'uso delle abilitazioni software come oggetti di calcolo. Questo concetto è basato sull'idea che alcuni componenti di programmi possano avere il privilegio di creare o esaminare queste abilitazioni. Un programma che crea un'abilitazione può eseguire un'operazione primitiva (`seal`) che sigilla una struttura dati, rendendo il contenuto di quest'ultima inaccessibile a qualsiasi componente di programma che non possieda i privilegi `seal` o `unseal`. Questi componenti potrebbero copiare la struttura dati, o passarne l'indirizzo ad altri componenti del programma, ma non possono ottenere l'accesso al suo contenuto. Tali abilitazioni software si introducono per portare un meccanismo di protezione all'interno del linguaggio di programmazione. L'unico problema che s'incontra seguendo tale metodo riguarda l'uso delle operazioni `seal` e `unseal`; tale uso richiede infatti un orientamento procedurale alla specifica della protezione. Per rendere disponibile l'ambiente di protezione ai programmatore di applicazioni sembra preferibile una notazione non procedurale o dichiarativa.

È necessario un meccanismo dinamico sicuro di controllo degli accessi, per poter distribuire tra i processi utenti le abilitazioni alle risorse del sistema. Per contribuire all'affidabilità generale di un sistema, il meccanismo per il controllo dell'accesso deve essere utilizzabile in modo sicuro, e per essere utile nella pratica deve essere anche ragionevolmente efficiente. Questo requisito ha portato allo sviluppo di un certo numero di costrutti di linguaggio che consentono ai programmatore di dichiarare vari limiti riguardanti l'uso di specifiche risorse (si vedano le Note bibliografiche per gli appropriati riferimenti).

Questi costrutti forniscono meccanismi per tre funzioni:

1. distribuire, in modo sicuro ed efficiente, le abilitazioni tra i processi clienti: in particolare i meccanismi assicurano che un processo utente si servirà di una risorsa solo se gli è stata concessa una specifica abilitazione;
2. specificare il tipo di operazioni che un processo particolare può compiere su una risorsa assegnata (per esempio, a un lettore di file si deve permettere soltanto di leggere i file, mentre a uno scrittore si possono concedere sia lettura sia scrittura): non dovrebbe essere necessario concedere lo stesso insieme di diritti a ogni processo utente e un processo non dovrebbe avere la possibilità di ampliare il proprio insieme di diritti d'accesso, tranne che se abbia ricevuto l'autorizzazione da parte del meccanismo di controllo degli accessi;
3. specificare l'ordine in cui un processo particolare può compiere le diverse operazioni su una risorsa (per esempio, un file deve essere aperto prima di essere letto): deve essere possibile dare a due processi limitazioni diverse sull'ordine in cui possono compiere le operazioni sulla risorsa assegnata.

L'incorporazione dei concetti di protezione nei linguaggi di programmazione, intesa come strumento pratico per la progettazione dei sistemi, è in una fase iniziale. La protezione probabilmente acquisterà un crescente interesse da parte dei progettisti di nuovi sistemi con architetture distribuite, e requisiti sempre più severi sulla sicurezza dei dati; sarà quindi riconosciuta più diffusamente anche l'importanza d'idonee notazioni di linguaggio in cui esprimere i requisiti di protezione.

17.12.2 Protezione al run-time nel linguaggio Java

Poiché Java è stato pensato per l'esecuzione in ambiente distribuito, la macchina virtuale Java, o jvm (*Java virtual machine*) è dotata di molti meccanismi di protezione. I programmi Java sono composti da classi, ognuna delle quali è un insieme di campi di dati e di funzioni (chiamate metodi) che operano su quei campi. La jvm carica una classe come risposta a una richiesta di creazione di istanze (o oggetti) di quella classe. Una tra le caratteristiche più originali e utili del linguaggio Java è la gestione del caricamento dinamico di classi non fidate da una rete e dell'esecuzione all'interno della stessa jvm di classi che si ritengono mutuamente sospette.

Proprio per queste caratteristiche, il problema della protezione è di vitale importanza. Le classi eseguite dalla stessa jvm possono provenire da diverse fonti e possono avere diversi gradi di affidabilità. Quindi, è insufficiente imporre la protezione a livello del processo della jvm. Intuitivamente, il fatto che una richiesta d'apertura di file sia consentita dipenderà generalmente da quale classe ha fatto la richiesta d'apertura. Il sistema operativo non ha queste informazioni.

Dunque questo tipo di decisioni di protezione sono gestite all'interno della jvm. Quando la jvm carica una classe, la assegna a un dominio di protezione che fornisce i permessi per quella classe. Il dominio di protezione al quale si assegna una classe dipende dall'url da cui la classe è stata caricata e da eventuali firme digitali sul file della classe (le firme digitali sono trattate nel Paragrafo 16.4.1.3). Un file che permette la configurazione dei criteri di protezione determina i permessi che si danno al dominio (e alle sue classi). Per esempio, le classi prelevate da un server fidato si potrebbero mettere in un dominio di protezione che permette a tali classi di accedere ai file nelle directory iniziali degli utenti, mentre le classi prelevate da un server ritenuto non fidato potrebbero non avere alcun permesso d'accesso ai file.

Per una jvm può essere difficile stabilire quale sia la classe responsabile di una richiesta d'accesso a una risorsa protetta. Gli accessi avvengono spesso in modo indiretto, per mezzo di librerie di sistema o altre classi. Si consideri per esempio una classe cui non sia permesso aprire connessioni di rete. Potrebbe chiamare una libreria di sistema per richiedere il caricamento dei contenuti di un url. La jvm deve decidere se aprire a no una connessione di rete per questa richiesta. Ma quale classe si dovrebbe considerare per determinare se si debba concedere o no la connessione, l'applicazione o la libreria di sistema?

L'orientamento seguito nel linguaggio Java è quello di richiedere alla classe di libreria di permettere esplicitamente una connessione di rete. Più in generale, per poter accedere a una risorsa protetta, uno dei metodi nella sequenza delle chiamate che ha portato alla richiesta deve esplicitamente asserire il privilegio di accedere alla risorsa. In questo modo, il metodo si *assume la responsabilità* della richiesta e, presumibilmente, eseguirà anche tutti i controlli necessari ad assicurare la sicurezza della richiesta stessa. Chiaramente, non tutti i metodi sono abilitati ad asserire privilegi; lo possono fare solo se la classe d'appartenenza è in un dominio di protezione che ha esso stesso il permesso di esercitare quel privilegio.

Questo metodo di realizzazione si chiama ispezione dello stack. Ogni thread nella jvm ha uno stack associato per le sue attuali invocazioni di metodi. Quando un chiamante può non essere fidato, un metodo esegue una richiesta d'accesso all'interno di un blocco `doPrivileged()`, per accedere direttamente o indirettamente a una risorsa protetta. `doPrivileged()` è un metodo statico della classe `AccessController` al quale si passa una classe con un metodo `run()` da invocare. Quando l'esecuzione entra nel blocco `doPrivileged()`, si annota l'elemento dello stack per questo metodo per indicare questo fatto, e successivamente si eseguono le istruzioni nel blocco. Quando, in seguito, si richiede l'accesso a una risorsa protetta, o da parte di questo metodo o da parte di un metodo da esso chiamato, s'invoca `checkPermissions()` per richiedere l'ispezione dello stack, allo scopo di determinare se l'accesso richiesto debba essere concesso. L'ispezione consiste nell'esame degli elementi presenti nello stack del thread chiamante, partendo da quello inserito più recentemente e procedendo verso il meno recente. Se prima si trova un elemento che ha l'annotazione `doPrivileged()`, `checkPermissions()` permette immediatamente l'accesso. Se invece si trova prima un elemento per il quale l'accesso non è consentito nell'ambito del dominio di protezione della classe del metodo, `checkPermissions()` genera un'eccezione `AccessControlException`. Se l'ispezione esaurisce lo stack senza trovare né un tipo d'elemento né l'altro, allora la concessione dell'accesso dipende dalla implementazione (alcune implementazioni della jvm permettono l'accesso, altre lo negano).

La procedura d'ispezione dello stack è illustrata nella Figura 17.14. In quest'esempio, il metodo `gui()` di una classe nel dominio di protezione *untrusted applet* compie due operazioni: prima una `get()` e poi una `open()`. La prima corrisponde all'invocazione del metodo `get()` di una classe nel dominio di protezione *url loader*, che ha i permessi per aprire sessioni nei siti nel dominio `lucent.com`, e in particolare per il server `proxy.lucent.com` per individuare gli url. Per questa ragione, l'invocazione di `get()` dall'*applet* non fidata andrà a buon fine: la chiamata a `checkPermissions()` nella libreria di rete troverà l'elemento dello stack relativo al metodo `get()` che ha eseguito la sua `open()` in un blocco `doPrivileged()`. Tuttavia, l'invocazione della `open()` da parte dell'*applet* non fidata risulta invece in un'eccezione, poiché la chiamata a `checkPermissions()` non trova alcuna annotazione `doPrivileged()` prima di raggiungere l'elemento dello stack relativo al metodo `gui()`.

dominio di protezione:	<code>applet</code> non fidata	caricatore di URL	interconnessione
permesso della socket:	nessuno	<code>*.lucent.com:80, connect</code>	qualsiasi
classe:	<pre>gui: ... get(url); open(addr); ... </pre>	<pre>get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> ... </pre>	<pre>open(Addr a): ... checkPermission (a, connect); connect (a); ... </pre>

Figura 17.14 Ispezione dello stack.

Ovviamente, affinché l'ispezione dello stack possa funzionare, un programma non deve poter modificare le annotazioni sul suo stesso elemento dello stack, né compiere altre manipolazioni che interferiscano con l'ispezione dello stack. Questa è una delle differenze più importanti tra il linguaggio Java e molti altri linguaggi (compreso il C++). Un programma scritto in Java non può accedere direttamente alla memoria. Piuttosto, può manipolare solo oggetti per i quali ha un riferimento. I riferimenti non si possono contraffare e le manipolazioni si compiono per mezzo d'interfacce ben definite. La conformità a questi criteri è imposta da un raffinato insieme di controlli della fase di caricamento e della fase d'esecuzione. Ne segue che un oggetto non può manipolare il proprio stack, poiché non può ottenere un riferimento né a essa né ad altri componenti del sistema di protezione.

Più in generale, i controlli del linguaggio Java nella fase di caricamento e nella fase d'esecuzione impongono la sicurezza dei tipi (*type safety*) delle classi. La sicurezza dei tipi garantisce che le classi non possano trattare gli interi come puntatori, scrivere oltre la fine di un array, accedere alla memoria in modi arbitrari. Un programma può accedere a un oggetto soltanto attraverso i metodi definiti per quell'oggetto dalla sua classe. Su ciò si fonda il sistema di protezione del linguaggio Java, poiché permette a una classe di *incapsulare* efficacemente i propri dati e metodi e di proteggerli da altre classi caricate nella stessa jvm. Per esempio, una variabile si può definire `private`, in modo che solo la classe che la contiene possa accedervi, oppure si può definire `protected`, in modo che possano accedervi solo le classi che la contiene, le sue sottoclassi e le classi dello stesso package. La sicurezza dei tipi garantisce l'imposizione di queste limitazioni.

17.13 Sommario

- Le caratteristiche di protezione del sistema sono guidate dal principio della necessità di sapere e attuano meccanismi per far rispettare il principio del minimo privilegio.
- I sistemi elaborativi contengono molti oggetti, che devono essere protetti contro i possibili abusi. Gli oggetti possono essere hardware (come la memoria, il tempo di cpu o i dispositivi di i/o) o software (come i file, i programmi e i semafori).
- Un diritto d'accesso è un permesso per eseguire un'operazione su un oggetto. Un dominio è un insieme di diritti d'accesso. I processi vengono eseguiti in domini e possono usare tutti i diritti d'accesso del dominio per accedere agli oggetti e manipolarli. Durante il suo ciclo di vita un processo può essere vincolato a un dominio di protezione o può essergli consentito di passare da un dominio a un altro.
- Un metodo comune per proteggere gli oggetti consiste nel fornire una serie di anelli di protezione, ciascuno con più privilegi del precedente. arm, per esempio, fornisce quattro livelli di protezione di cui il più privilegiato, TrustZone, è richiamabile solo dalla modalità kernel.
- La matrice d'accesso è un modello generale di protezione; fornisce un meccanismo per la protezione senza imporre uno specifico criterio di protezione al sistema o ai suoi utenti. La separazione tra criteri e meccanismi è un'importante caratteristica di progettazione.
- La matrice d'accesso è sparsa e normalmente si realizza per mezzo di liste d'accesso associate a ciascun oggetto, oppure per mezzo di liste di abilitazioni associate a ciascun dominio. Si può inserire la protezione dinamica nel modello della matrice d'accesso considerando i domini e la stessa matrice d'accesso come oggetti. La revoca dei diritti d'accesso in un modello di protezione dinamico è di solito più facile da realizzare con lo schema delle liste d'accesso che con le liste di abilitazioni.
- I sistemi reali sono molto più limitati rispetto al modello generale. Le distribuzioni unix più datate sono rappresentative, perché forniscono, per ogni file, controlli di accesso discrezionali di protezione di lettura, scrittura ed esecuzione in maniera separata per il proprietario, il gruppo e gli utenti generali. I sistemi più moderni sono più vicini al modello generale o perlomeno forniscono una varietà di funzioni di protezione per proteggere il sistema e i suoi utenti.
- Solaris, dalla versione 10, realizza il principio del privilegio minimo attraverso il controllo dell'accesso basato sul ruolo, una forma di matrice d'accesso. Un'altra estensione di protezione è il controllo di accesso obbligatorio, una forma di imposizione delle politiche di sistema.
- I sistemi basati sulle abilitazioni offrono una protezione più raffinata rispetto ai modelli precedenti, fornendo capacità specifiche ai processi per mezzo dello "spezzettamento" dei poteri di root in aree distinte. Altri metodi per migliorare la protezione includono la protezione dell'integrità del sistema (sip), il filtraggio delle chiamate di sistema, il meccanismo di sandboxing e la firma del codice.
- La protezione basata sul linguaggio offre un controllo delle richieste e dei privilegi più selettivo di quello ottenibile con il sistema operativo. Per esempio, una singola jvm può eseguire molti thread, ognuno in un diverso dominio di protezione. La jvm controlla le richieste di risorse attraverso un raffinato meccanismo di ispezione dello stack e attraverso la sicurezza dei tipi offerta dal linguaggio.

Esercizi di ripasso

17.1 Quali sono le principali differenze tra liste delle abilitazioni e liste d'accesso?

17.2 Un file nel Borroughs B7000/B6000 mcp può essere contrassegnato come dato sensibile. Quando un tale file viene cancellato, l'area in cui era memorizzato viene sovrascritta da bit casuali. Per quali scopi un tale schema può essere utile?

17.3 In un sistema di protezione ad anello il livello 0 ha l'accesso più ampio agli oggetti, e il livello n (con $n > 0$) ha diritti di accesso più bassi. I diritti di accesso di un programma a un dato livello nell'anello sono considerati un insieme di abilitazioni. Che relazione c'è tra le abilitazioni a un oggetto per un dominio a livello j e un dominio a livello i (per $j > i$)?

17.4 Il sistema rc 4000, come altri sistemi, ha definito un albero dei processi tale che tutti i discendenti di un processo possono ottenere risorse (oggetti) e diritti di accesso solo dai loro antenati. Un discendente non potrà quindi mai fare niente di quello che gli antenati non possono fare. La radice dell'albero è il sistema operativo, che ha la capacità di fare ogni cosa. Assumete che l'insieme dei diritti di accesso sia rappresentato tramite una matrice di accesso A . $A(x,y)$ definisce i diritti di accesso del processo x a un oggetto y . Se x è discendente di z , che relazione c'è tra $A(x,y)$ e $A(z,y)$, per un oggetto arbitrario y ?

17.5 Quali problemi di protezione possono nascere se viene utilizzata uno stack condiviso per il passaggio di parametri?

17.6 Considerate un ambiente elaborativo in cui a ogni processo e a ogni oggetto del sistema sia associato un numero univoco. Supponete che un processo con numero n possa accedere a oggetti con numero m solo se $n > m$. Quale tipo di struttura di protezione avremmo?

17.7 Considerare un ambiente elaborativo in cui un processo ottiene il privilegio di accedere a un oggetto soltanto n volte. Suggerite uno schema per implementare una tale politica.

17.8 Se tutti i diritti di accesso di un oggetto vengono cancellati, l'oggetto non è più accessibile. A questo punto, lo stesso oggetto dovrebbe essere cancellato, e lo spazio da lui occupato dovrebbe essere restituito al sistema. Suggerite un'efficiente implementazione di questo schema.

17.9 Quali difficoltà si incontrano nella protezione di un sistema in cui gli utenti hanno il permesso di eseguire le loro operazioni di i/o?

17.10 La lista delle abilitazioni è solitamente mantenuta nello spazio di indirizzi dell'utente. In che modo il sistema assicura che l'utente non possa modificare il contenuto della lista?

Esercizi

17.11 Grazie alla matrice per il controllo dell'accesso si può determinare se un processo sia abilitato a passare, per esempio, dal dominio A al dominio B, e se possa godere dei privilegi d'accesso del dominio B. Valutate se questa soluzione sia da considerarsi equivalente all'inclusione dei privilegi d'accesso del dominio B in quelli del dominio A.

17.12 Considerate un sistema in cui i videogiochi possano essere usati dagli studenti solo tra le 22 e le 6, dai membri della facoltà tra le 17 e le 8 e dal personale del centro di calcolo a tutte le ore. Suggerite uno schema per realizzare efficacemente questo criterio.

17.13 Dite quali caratteristiche dell'hardware di un calcolatore siano necessarie per ottenere un'efficiente manipolazione delle abilitazioni. Dite se queste caratteristiche si possano adoperare per la protezione della memoria.

17.14 Evidenziate i punti di forza e le vulnerabilità di una matrice d'accesso realizzata tramite le liste d'accesso associate agli oggetti.

17.15 Evidenziate i punti di forza e le vulnerabilità di una matrice d'accesso realizzata tramite le abilitazioni associate ai domini.

17.16 Spiegate perché i sistemi basati sull'abilitazione, come Hydra, siano più flessibili nell'applicare i parametri di protezione rispetto alla struttura di protezione ad anelli.

17.17 Dite che cos'è il principio della necessità di sapere, e perché è importante che un sistema di protezione aderisca a questo principio.

17.18 Chiarite quale tra i seguenti sistemi consente ai progettisti di moduli di rendere effettivo il principio della necessità di sapere.

- a. La struttura di protezione ad anelli.
- b. Lo schema di ispezione dello stack della jvm.

17.19 Descrivete in che modo il modello di protezione del linguaggio Java verrebbe meno se si permettesse a un programma scritto in Java di alterare direttamente le annotazioni nel suo elemento dello stack.

17.20 In che cosa si assomigliano la funzionalità della matrice d'accesso e il controllo dell'accesso basato sui ruoli? In che cosa differiscono?

17.21 In che modo il principio del privilegio minimo aiuta a creare sistemi di protezione?

17.22 Come possono persistere, nei sistemi che adottano il principio del privilegio minimo, carenze di protezione tali da causare violazioni alla sicurezza?

CAPITOLO 18

Macchine virtuali

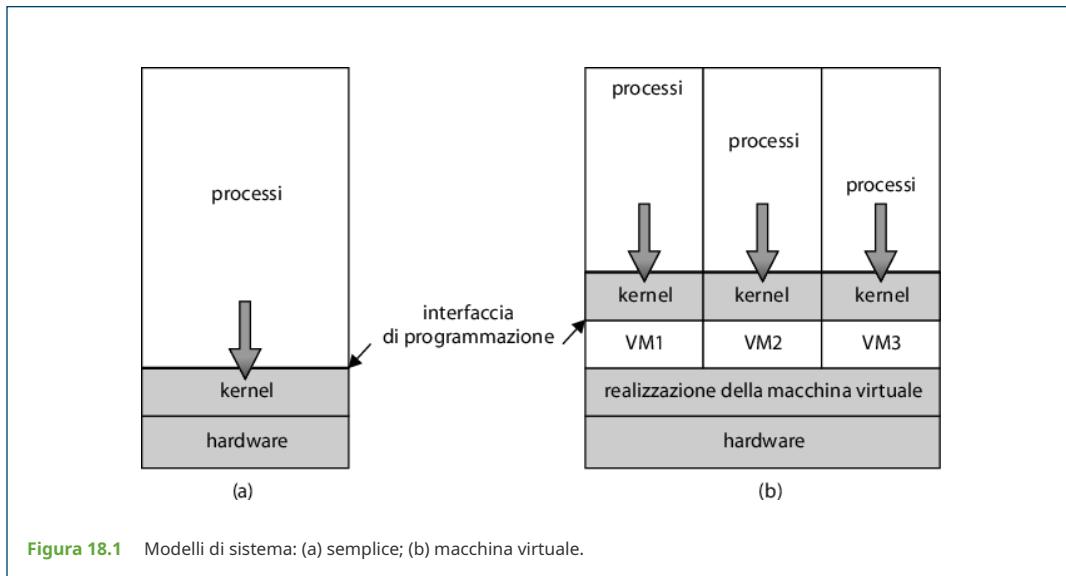
Il termine virtualizzazione ha molti significati e gli aspetti della virtualizzazione permeano il mondo dei calcolatori. Le macchine virtuali sono un esempio di questa tendenza. In genere per mezzo di una macchina virtuale i sistemi operativi e le applicazioni ospiti vengono eseguiti in un ambiente che appare loro come l'hardware nativo, ma che inoltre li protegge, li gestisce e li limita.

Questo capitolo approfondisce gli usi, le funzioni e l'implementazione delle macchine virtuali. Le macchine virtuali possono essere implementate in vari modi e questo capitolo descrive queste varianti. Una possibilità è quella di aggiungere il supporto per la macchina virtuale al kernel. Dato che questo metodo di implementazione è il più pertinente a questo libro lo analizzeremo in maniera completa. Inoltre, dato che le caratteristiche hardware fornite dalla cpu e dai dispositivi di i/o possono supportare l'implementazione della macchina virtuale, vedremo come queste caratteristiche vengono utilizzate dai moduli del kernel appropriati.

18.1 Introduzione

L'idea fondamentale che sta dietro a una macchina virtuale è quella di astrarre l'hardware di un singolo computer (cpu, memoria, dischi, schede di rete e così via) in diversi ambienti di esecuzione differenti, creando così l'illusione che ogni ambiente distinto sia in esecuzione su un proprio computer privato. Questo concetto può sembrare simile al metodo stratificato di implementazione del sistema operativo (si veda il Paragrafo 2.8.2) e per certi versi lo è. Nel caso della virtualizzazione, vi è uno strato che crea un sistema virtuale in cui è possibile eseguire sistemi operativi o applicazioni.

L'implementazione della macchina virtuale coinvolge diverse componenti. Alla base vi è l'host, il sistema hardware sottostante che gestisce le macchine virtuali. Il gestore della macchina virtuale (*virtual machine manager*, vmm), noto anche come hypervisor, crea e gestisce le macchine virtuali fornendo un'interfaccia che è identica all'host (eccetto nel caso della paravirtualizzazione, discussa più avanti). A ogni processo guest (*ospite*) viene fornita una copia virtuale dell'host (Figura 18.1). Di solito il processo guest è un sistema operativo. Una singola macchina fisica può quindi eseguire più sistemi operativi simultaneamente, ciascuno nella propria macchina virtuale.



Vale la pena di prendersi un momento per riflettere su come, con la virtualizzazione, la definizione di "sistema operativo" risulta ulteriormente sfumata. Considerate, per esempio, un software vmm come vmware esx. Questo software di virtualizzazione è installato sull'hardware, viene eseguito all'avvio e fornisce servizi alle applicazioni. I servizi includono quelli tradizionali, come la pianificazione e la gestione della memoria, insieme a nuove tipologie di servizi, come la migrazione delle applicazioni tra sistemi. Inoltre, le applicazioni sono di fatto sistemi operativi guest. Possiamo considerare il vmm vmware esx come un sistema operativo che, a sua volta, esegue altri sistemi operativi? Certamente, perché si comporta come un sistema operativo. Per chiarezza, però, noi chiamiamo vmm il componente che offre ambienti virtuali.

L'implementazione di un vmm è molto varia. Tra le opzioni disponibili vi sono le seguenti.

- Soluzioni basate su hardware che forniscono il supporto per la creazione e la gestione della macchina virtuale tramite firmware. Questi vmm, che si trovano comunemente nei mainframe e nei server di medie/grandi dimensioni, sono generalmente conosciuti come hypervisor di tipo 0. ibm Ipar e Oracle Idoms ne sono esempi.
- Software assimilabili a sistemi operativi costruiti per fornire la virtualizzazione. Tra questi vi sono vmware esx (citato in precedenza), Joyent Smartos e Citrix XenServer. Questi vmm sono noti come hypervisor di tipo 1.
- Sistemi operativi general-purpose che forniscono funzioni standard e funzioni di vmm, tra cui Microsoft Windows Server con HyperV e Linux RedHat con la funzionalità kvm. Poiché tali sistemi hanno un insieme di funzionalità simile a quello degli hypervisor di tipo 1, anch'essi sono classificati come di tipo 1.
- Applicazioni che si eseguono su sistemi operativi standard, ma forniscono funzionalità di vmm per sistemi operativi guest. Queste applicazioni, che includono vmware Workstation e Fusion, Parallels Desktop e Oracle Virtual-Box, sono hypervisor di tipo 2.
- Paravirtualizzazione, una tecnica in cui il sistema operativo guest viene modificato per lavorare in collaborazione con il vmm al fine di ottimizzare le prestazioni.
- Virtualizzazione dell'ambiente di programmazione, in cui i vmm non virtualizzano l'hardware vero e proprio, ma creano piuttosto un sistema virtuale ottimizzato. Questa tecnica è utilizzata da Oracle Java e Microsoft.Net.
- Emulatori che consentono ad applicazioni scritte per un certo ambiente hardware di funzionare su un ambiente hardware molto diverso, per esempio su un diverso tipo di cpu.

- Contenitori di applicazioni, che non forniscono una completa virtualizzazione, ma offrono alcune funzionalità di virtualizzazione separando le applicazioni dal sistema operativo. Oracle Solaris Zones, bsd Jails e ibm aix wpars “contengono” le applicazioni rendendole più sicure e gestibili.

La varietà delle tecniche di virtualizzazione in uso oggi è una testimonianza dell'ampiezza, della profondità e dell'importanza della virtualizzazione nell'informatica moderna. La virtualizzazione è di fondamentale importanza per i data-center, per lo sviluppo di applicazioni efficienti e per il test del software, solo per fare alcuni esempi.

INDIREZIONE

“In informatica qualsiasi problema può essere risolto da un altro livello di indirezione” – David Wheeler “... a parte quello di troppi livelli di indirezione.” – Kevin Henney.

18.2 Storia

Le macchine virtuali hanno fatto la loro comparsa sul mercato nel 1972, con i mainframe ibm. La virtualizzazione era fornita dal sistema operativo ibm vm, che con il tempo si è evoluto ed è disponibile ancora oggi. Molti dei concetti originali di ibm vm si trovano in altri sistemi, ragion per cui vale davvero la pena di analizzare questo sistema operativo.

ibm vm/370 divideva un mainframe in più macchine virtuali, ciascuna con il proprio sistema operativo. Una delle maggiori difficoltà riguardava i sistemi di dischi. Supponiamo che la macchina fisica avesse tre dischi, ma si volessero supportare sette macchine virtuali. Chiaramente non era possibile allocare un disco a ogni macchina virtuale: la soluzione fu quella di fornire dischi virtuali, chiamati minidisk nel sistema operativo vm di ibm. I minidisk sono identici ai dischi rigidi del sistema in tutti gli aspetti eccetto la dimensione. Il sistema implementava ogni minidisk allocando le tracce sui dischi fisici di cui il minidisk aveva bisogno.

Una volta che le macchine virtuali erano state create, gli utenti potevano eseguire uno dei sistemi operativi o pacchetti software che erano disponibili sulla macchina sottostante. Nel caso del sistema ibm vm, un utente eseguiva di norma cms – un sistema operativo interattivo monoutente.

Per molti anni dopo che ibm ha introdotto questa tecnologia la virtualizzazione è rimasta confinata in quell'ambito; la maggior parte dei sistemi non la supportava. Tuttavia, una definizione formale di virtualizzazione ha contribuito a stabilire i requisiti di sistema e gli obiettivi per le funzionalità.

I requisiti di virtualizzazione richiedevano:

- Fedeltà. Un vmm fornisce un ambiente per i programmi essenzialmente identico alla macchina originale.
- Prestazioni. I programmi in esecuzione all'interno di tale ambiente soffrono di una perdita di prestazioni modesta.
- Sicurezza. Il vmm è in completo controllo delle risorse di sistema.

Questi requisiti guidano ancora oggi lo sviluppo della virtualizzazione.

Dalla fine degli anni '90 le cpu Intel 80x86 diventarono piuttosto diffuse, veloci e ricche di funzionalità. Come conseguenza gli sviluppatori iniziarono a dedicare i loro sforzi all'implementazione della virtualizzazione su tale piattaforma. Sia Xen sia vmware hanno creato tecnologie, ancora in uso oggi, per permettere di eseguire sistemi operativi guest su 80x86. Da quel momento la virtualizzazione è cresciuta in modo da coinvolgere tutte le cpu più diffuse, molti strumenti commerciali e open-source e molti sistemi operativi. Per esempio, il progetto open-source VirtualBox (<http://www.virtualbox.org>), fornisce un programma che può essere eseguito su cpu Intel x86 e amd64 e su sistemi operativi host Windows, Linux, macos e Solaris. Tra i possibili sistemi operativi guest vi sono diverse versioni di Windows, Linux, Solaris e bsd, oltre a ms-dos e ibm os/2.

18.3 Vantaggi e caratteristiche

Diversi vantaggi rendono la virtualizzazione attraente. La maggior parte di questi è essenzialmente legata alla capacità di condividere lo stesso hardware, ma eseguire diversi ambienti differenti (cioè sistemi operativi diversi) contemporaneamente.

Un importante vantaggio della virtualizzazione è che il sistema host viene protetto dalle macchine virtuali, proprio come le macchine virtuali sono protette l'una dall'altra. Un virus all'interno di un sistema operativo guest potrebbe danneggiare il sistema operativo, ma è improbabile che possa influenzare l'host o gli altri guest. Poiché ogni macchina virtuale è pressoché completamente isolata da tutte le altre, non esistono quasi problemi di protezione.

Un potenziale svantaggio di questo isolamento è che esso può impedire la condivisione delle risorse. Sono stati seguiti due diversi approcci per fornire la condivisione. In primo luogo, è possibile condividere un volume del file system e quindi condividere i file. In secondo luogo, è possibile definire una rete di macchine virtuali, ciascuna delle quali può inviare informazioni attraverso la rete di comunicazione virtuale. La rete modella reti fisiche di comunicazione, ma è implementata via software. Naturalmente il vmm è libero di consentire a qualunque ospite di utilizzare le risorse fisiche, come una connessione di rete fisica (con condivisione fornita dal vmm), nel qual caso gli ospiti accettati possono comunicare tra loro attraverso la rete fisica.

Una caratteristica comune alla maggior parte delle implementazioni della virtualizzazione è la capacità di congelare, o *sospendere*, una macchina virtuale in esecuzione. Molti sistemi operativi forniscono una tale caratteristica, fondamentale per i processi, ma i vmm fanno un passo in più e permettono di effettuare copie e istantanee del guest. La copia può essere usata per creare una nuova vm o per spostare una vm da una macchina all'altra mantenendo inalterato il suo stato. Il guest può quindi *riprendere* dal punto in cui si trovava, come se fosse ancora sulla sua macchina originale, formando così un clone. L'istanziana memorizza la situazione in un dato istante di tempo in modo che il guest, se necessario, possa essere riportato a quello stato (per esempio quando è stato fatto un cambiamento, ma questo non è più desiderato). Spesso un vmm consente di scattare diverse istantanee. Per esempio, è possibile registrare lo stato di un guest con un'istanziana ogni giorno per un mese, rendendo possibile il ripristino di uno qualsiasi degli stati registrati. Queste funzionalità sono utilizzate con grande vantaggio in ambienti virtuali.

Un sistema virtuale è uno strumento perfetto per ricerca e sviluppo su sistemi operativi. Normalmente la modifica di un sistema operativo è un'operazione complicata, perché i sistemi operativi sono programmi grandi e complessi e un cambiamento in una loro parte può causare la comparsa di misteriosi errori in altre parti. La potenza del sistema operativo rende le modifiche particolarmente pericolose. Poiché il sistema operativo viene eseguito in modalità kernel, una modifica errata a un puntatore potrebbe causare un errore in grado di distruggere l'intero file system. È pertanto necessario testare con attenzione tutte le modifiche al sistema operativo.

Il sistema operativo viene eseguito su una macchina di cui ha il completo controllo, che deve dunque essere arrestata e restare inutilizzata mentre le modifiche vengono apportate e testate. Questo intervallo di tempo viene comunemente chiamato periodo di sviluppo del sistema e sui sistemi condivisi, dal momento che rende il sistema indisponibile agli utenti, è spesso pianificato a tarda notte o nei fine settimana, quando il carico del sistema è basso.

Un sistema virtuale può eliminare in gran parte quest'ultimo problema. I programmatore di sistema sono dotati di una macchina virtuale privata e lo sviluppo avviene sulla macchina virtuale invece che su una macchina fisica. Il normale funzionamento del sistema viene interrotto solo quando una modifica è completata, testata e pronta per essere messa in produzione.

Un altro vantaggio offerto agli sviluppatori dalle macchine virtuali è la possibilità di eseguire più sistemi operativi contemporaneamente sulla propria workstation. Questa workstation virtualizzata consente di trasferire e testare rapidamente i programmi in ambienti diversi. È inoltre possibile eseguire più versioni di uno stesso programma, ciascuna nel suo sistema operativo isolato, su un unico sistema. Allo stesso modo, gli ingegneri del controllo qualità possono testare le applicazioni in diversi ambienti senza il bisogno di acquistare, alimentare e mantenere un computer per ogni ambiente.

Un vantaggio importante nell'utilizzo di macchine virtuali nei data-center di produzione è il consolidamento del sistema, che consiste nel prendere due o più sistemi fisici distinti ed eseguirli, all'interno di macchine virtuali, su un unico sistema. Le conversioni da fisico a virtuale permettono un'ottimizzazione delle risorse, poiché molti sistemi scarsamente utilizzati possono essere combinati per creare un unico sistema con una utilizzazione maggiore.

Va considerato, inoltre, che gli strumenti di gestione che fanno parte di un vmm permettono agli amministratori di sistema di gestire molti più sistemi di quanto potrebbero fare altrimenti. Un ambiente virtuale potrebbe includere 100 server fisici con 20 server virtuali in esecuzione su ogni server fisico. Senza virtualizzazione, la presenza di 2.000 server richiederebbe diversi amministratori di sistema. Con la virtualizzazione e i suoi strumenti, lo stesso lavoro può essere gestito da uno o due amministratori. Uno degli strumenti utilizzati in questi casi è il templating, in cui un'immagine standard della macchina virtuale, che include un sistema operativo ospite installato e configurato e le sue applicazioni, viene salvata e utilizzata come sorgente per più macchine virtuali in esecuzione. Tra le altre caratteristiche menzioniamo la gestione degli aggiornamenti di tutti i guest, il loro backup, il loro ripristino e il controllo del loro utilizzo delle risorse.

La virtualizzazione, oltre a migliorare l'utilizzo delle risorse, ne migliora anche la gestione. Alcuni vmm includono una funzionalità di migrazione in tempo reale (live migration) che permette di spostare un guest in esecuzione da un server fisico a un altro senza interrompere il suo funzionamento o le connessioni di rete attive. Se un server è sovraccarico, la migrazione in tempo reale può liberare risorse sull'host senza interrompere il guest. Analogamente, quando l'hardware di un host deve essere riparato o aggiornato, i guest possono essere spostati su altri server, per poi effettuare la manutenzione dell'host e infine riportare i guest sull'host originale. Questa operazione avviene senza tempi morti e senza interruzioni per gli utenti.

Pensiamo ora ai possibili vantaggi della virtualizzazione nella distribuzione delle applicazioni. Se un sistema può facilmente aggiungere, rimuovere o spostare una macchina virtuale, allora perché installare le applicazioni direttamente sul sistema? L'applicazione potrebbe piuttosto essere preinstallata su un sistema operativo messo a punto e personalizzato in una macchina virtuale. Questo metodo offre diversi vantaggi agli sviluppatori di applicazioni: la gestione dell'applicazione diventerebbe più semplice, sarebbe richiesto meno tuning e il supporto tecnico sarebbe più immediato. Anche gli amministratori di sistema dovrebbero

trovare l'ambiente più facile da gestire: l'installazione sarebbe semplice e ridistribuire l'applicazione su altri sistemi sarebbe molto più facile rispetto alle solite fasi di disinstallazione e reinstallazione. Affinché questa metodologia venga adottata in maniera diffusa, però, il formato delle macchine virtuali deve essere standardizzato in modo che ogni macchina virtuale possa essere eseguita su qualsiasi piattaforma di virtualizzazione. L'*Open Virtual Machine Format* è un tentativo di fornire una tale standardizzazione e potrebbe riuscire a unificare i formati delle macchine virtuali.

La virtualizzazione ha gettato le basi per molti altri progressi nella realizzazione, nella gestione e nel monitoraggio di sistemi informatici. Il cloud computing, per esempio, è reso possibile da una virtualizzazione in cui risorse come la cpu, la memoria e l'i/o vengono offerti come servizi ai clienti, utilizzando tecnologie Internet. Utilizzando le api un programma può chiedere a una struttura cloud di creare migliaia di macchine virtuali, su ognuna delle quali è in esecuzione uno specifico sistema operativo guest e un'applicazione, a cui altri possono accedere via Internet. Molti giochi multiutente, siti di condivisione di fotografie e altri servizi web sfruttano queste caratteristiche.

Nell'ambito del mondo desktop, la virtualizzazione sta permettendo agli utenti di connettersi alle macchine virtuali situate in data-center remoti e accedere alle loro applicazioni come se fossero locali. Questa pratica può aumentare la sicurezza, perché non ci sono dati memorizzati sui dischi locali dell'utente, e il costo delle risorse di calcolo dell'utente può diminuire. L'utente deve disporre di rete, cpu e memoria, ma tutto ciò che queste componenti del sistema devono fare è visualizzare l'immagine del guest in esecuzione in remoto (attraverso un protocollo come rdp). Non c'è pertanto bisogno che queste componenti siano costose e ad alte prestazioni. Emergeranno sicuramente in futuro nuovi impieghi della virtualizzazione, grazie all'aumento della sua diffusione e al miglioramento del supporto hardware.

18.4 Blocchi costituenti

Il concetto di macchina virtuale è utile, ma è difficile da implementare. È necessario molto lavoro per fornire un duplice esatto della macchina sottostante e ciò è vero in particolare su sistemi dual-mode, dove la macchina sottostante dispone solo di modalità utente e modalità kernel. In questo paragrafo esaminiamo i componenti necessari per la costruzione di una virtualizzazione efficiente. Si noti che questi componenti non sono richiesti dall'hypervisor di tipo 0, come discusso nel Paragrafo 18.5.2.

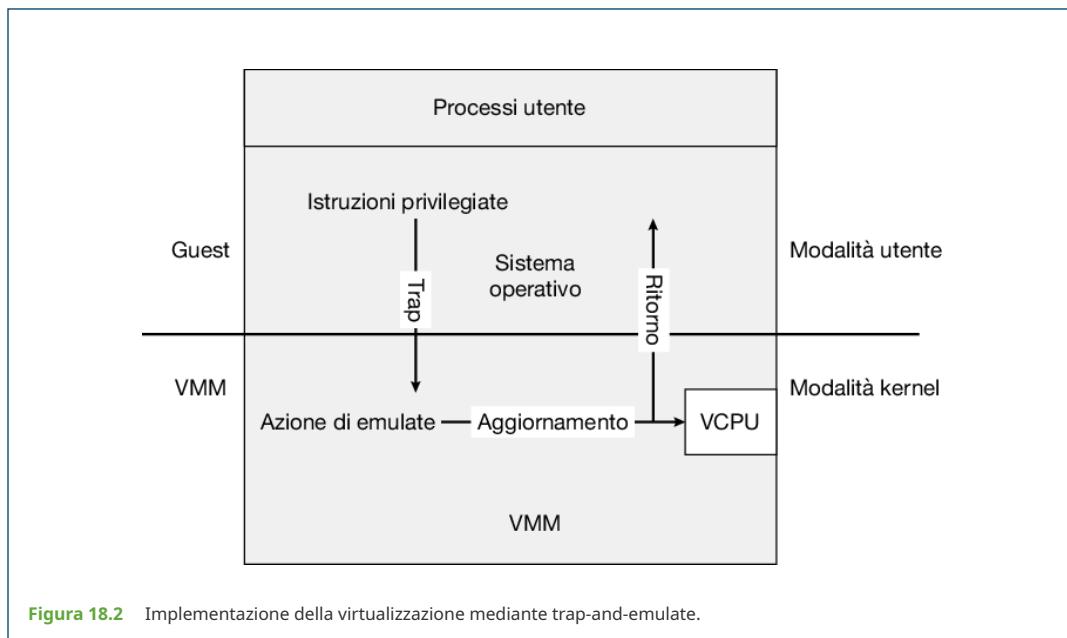
La possibilità di virtualizzare dipende dalle caratteristiche offerte dalla cpu. Se le caratteristiche sono sufficienti, allora è possibile scrivere un vmm in grado di fornire un ambiente guest; in caso contrario la virtualizzazione è impossibile. Un vmm utilizza diverse tecniche per implementare la virtualizzazione, tra cui trap-and-emulate e la traduzione binaria. In questo paragrafo analizziamo ciascuna di queste tecniche, oltre al supporto hardware necessario per supportare la virtualizzazione.

Nel leggere il paragrafo, ricordate che un importante concetto presente nella maggior parte delle varianti della virtualizzazione è l'implementazione di una cpu virtuale (vcpu). La vcpu non esegue il codice, ma rappresenta lo stato in cui la macchina guest ritiene che si trovi la cpu. Il vmm mantiene per ogni guest una vcpu che rappresenta lo stato corrente della cpu secondo il corrispondente guest. Quando il vmm assegna una cpu a un guest vengono utilizzate le informazioni della vcpu per caricare il giusto contesto, proprio come un sistema operativo general-purpose utilizzerebbe il pcb.

18.4.1 Trap-and-emulate

In un tipico sistema dual-mode il guest della macchina virtuale può essere eseguito solo in modalità utente (a meno che non sia fornito un supporto hardware aggiuntivo). Il kernel, naturalmente, viene eseguito in modalità kernel e non è sicuro consentire al codice a livello utente l'esecuzione in modalità kernel. Tuttavia, la macchina virtuale deve avere due modalità, proprio come la macchina fisica ha due modalità. Occorre avere di conseguenza una modalità utente virtuale e una modalità kernel virtuale, entrambe in esecuzione in modalità fisica utente. Le azioni che causano il passaggio dalla modalità utente alla modalità kernel su una macchina reale (per esempio una chiamata di sistema, un interrupt o un tentativo di eseguire un'istruzione privilegiata) devono causare nella macchina virtuale il passaggio dalla modalità utente virtuale alla modalità kernel virtuale.

Come può essere realizzato questo cambio di modalità? Quando il kernel del guest tenta di eseguire un'istruzione privilegiata si ha un errore (perché il sistema è in modalità utente) e ciò provoca una trap al vmm nella macchina reale. A questo punto il vmm ottiene il controllo ed esegue (o "emula") l'azione che è stata tentata dal kernel guest per conto del guest stesso. Il vmm ritorna poi il controllo alla macchina virtuale. Questo metodo è chiamato trap-and-emulate ed è illustrato nella Figura 18.2. La maggior parte dei prodotti di virtualizzazione utilizza questo metodo.



Quando si opera con le istruzioni privilegiate il tempo può diventare un problema. Tutte le istruzioni non privilegiate vengono eseguite nativamente sull'hardware, fornendo ai guest le stesse prestazioni delle applicazioni native. Le istruzioni privilegiate, tuttavia, creano overhead aggiuntivo e dunque il guest lavora più lentamente di quanto avverrebbe in maniera nativa. Inoltre, il fatto che la cpu venga multiprogrammata tra molte macchine virtuali può ulteriormente rallentare le macchine virtuali in modo imprevedibile.

Questo problema è stato affrontato in vari modi. Ibm vm, per esempio, permette alle istruzioni normali delle macchine virtuali di essere eseguite direttamente sull'hardware. Solo le istruzioni privilegiate (necessarie soprattutto per l'i/o) devono essere emulate e sono quindi eseguite più lentamente. In generale, con l'evoluzione dell'hardware, le prestazioni del meccanismo di trap-and-emulate sono state migliorate e sono stati ridotti i casi in cui è necessario utilizzare tale meccanismo. Molte cpu, per esempio, hanno ora modalità extra aggiuntive rispetto al loro funzionamento standard a due modalità. La vcpu non deve tenere traccia della modalità in cui si trova il sistema operativo guest, perché questa funzione è svolta dalla cpu fisica. Alcune cpu offrono ai guest la gestione dello stato della cpu via hardware, in modo che il vmm non abbia bisogno di fornire tale funzionalità ed eliminando così l'overhead.

18.4.2 Traduzione binaria

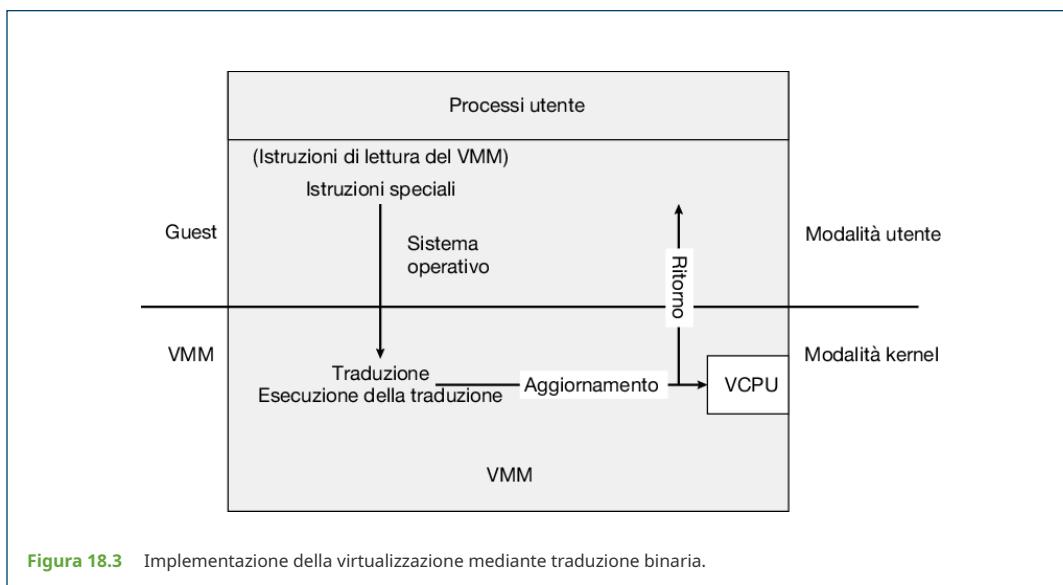
Alcune cpu non hanno una netta separazione tra istruzioni privilegiate e istruzioni non privilegiate. Sfortunatamente per gli implementatori di virtualizzazione, la linea di cpu Intel x86 fa parte di queste. Nel progetto dell'x86 non si è pensato alla virtualizzazione: in effetti la prima cpu della famiglia, l'Intel 4004, uscita nel 1971, è stata progettata per essere una calcolatrice. Il chip ha mantenuto la compatibilità con le versioni precedenti per tutta la sua esistenza, impedendo, per molte generazioni, le modifiche che avrebbero facilitato la virtualizzazione.

Consideriamo un esempio del problema. Il comando `popf` carica il registro flag dal contenuto dello stack. Se la cpu è in modalità privilegiata tutti i flag vengono sostituiti, mentre se la cpu è in modalità utente ne vengono sostituiti solo alcuni e gli altri vengono ignorati. Poiché non vengono generate trap se `popf` viene eseguito in modalità utente, la procedura di trap-and-emulate diventa inutile. Altre istruzioni x86 causano problemi simili. Ai fini di questa discussione, chiameremo questo insieme di istruzioni *istruzioni speciali*. Fino al 1998 l'utilizzo di trap-and-emulate per implementare la virtualizzazione su x86 era considerato impossibile a causa di queste istruzioni speciali.

Questo problema che sembrava insormontabile è stato risolto con l'applicazione della tecnica di traduzione binaria. La traduzione binaria è abbastanza semplice concettualmente, ma è complessa nella realizzazione. I passi fondamentali sono i seguenti:

1. se la vcpu guest è in modalità utente, il guest può eseguire le istruzioni in modo nativo su una cpu fisica;
2. se la vcpu guest è in modalità kernel, il guest crede di essere in esecuzione in modalità kernel. Il vmm esamina ogni istruzione eseguita dal guest in modalità kernel virtuale leggendo le prossime istruzioni che il guest sta per eseguire, basandosi sul program counter del guest. Le istruzioni diverse dalle istruzioni speciali vengono eseguite in modo nativo. Le istruzioni speciali vengono tradotte in un nuovo insieme di istruzioni che eseguono lo stesso task, per esempio la modifica dei flag nella vcpu.

La traduzione binaria, illustrata nella Figura 18.3, è implementata per mezzo di codice per la traduzione interno al vmm. Il codice legge dinamicamente dal guest le istruzioni binarie native, su richiesta, e genera codice nativo binario che viene eseguito al posto del codice originale.



Il metodo di base per la traduzione binaria appena descritto verrebbe eseguito correttamente, ma avrebbe uno scarso rendimento. Per fortuna la stragrande maggioranza delle istruzioni vengono eseguite in modo nativo. Ma come potrebbero essere migliorate le prestazioni per le altre istruzioni? Una risposta ci è data da una specifica implementazione della traduzione binaria: il metodo di vmware. La soluzione di vmware si basa sull'uso della cache. Il codice di sostituzione di ogni istruzione che deve essere tradotta è memorizzato in una cache. Tutte le successive esecuzioni di tali istruzioni sono eseguite dalla cache di traduzione senza bisogno di tradurle nuovamente. Se la cache è abbastanza grande, questo metodo può migliorare notevolmente le prestazioni.

Consideriamo un'altra problematica della virtualizzazione: la gestione della memoria, in particolare delle tabelle delle pagine. Come può il vmm mantenere lo stato della tabella delle pagine sia per i guest, che credono di gestire direttamente la tabella, che per il vmm

stesso? Un metodo comune, utilizzato sia con trap-and-emulate che con la traduzione binaria, è quello di utilizzare le tabelle delle pagine nidificate (*nested page table*, npt). Ogni sistema operativo guest mantiene una o più tabelle delle pagine da tradurre dal virtuale alla memoria fisica. Il vmm mantiene tabelle npt per rappresentare lo stato delle tabelle delle pagine del guest, proprio come crea una vcpu per rappresentare lo stato della cpu del guest. Il vmm sa quando il guest tenta di modificare la sua tabella delle pagine ed effettua la variazione equivalente nella npt. Quando l'ospite è sulla cpu, il vmm mette il puntatore alla npt appropriata nel registro della cpu opportuno per rendere tale tabella la tabella delle pagine attiva. Se il guest ha bisogno di modificare la tabella delle pagine (per esempio in caso di errore di pagina), questa operazione deve essere intercettata dal vmm e devono essere apportate opportune modifiche alle tabelle nidificate e alle tabelle delle pagine di sistema. Sfortunatamente l'uso delle npt può causare un aumento dei miss di tlb. Molte altre complicazioni devono essere affrontate per ottenere prestazioni ragionevoli.

Anche se a prima vista il metodo di traduzione binaria crea grandi quantità di overhead, tale metodo si comporta abbastanza bene da aver promosso la nascita di un nuovo settore tecnologico dedicato a virtualizzare sistemi basati su Intel x86. vmware ha testato l'impatto della traduzione binaria sulle prestazioni avviando uno di questi sistemi, Windows xp, e arrestandolo immediatamente, monitorando nel frattempo il tempo trascorso e il numero di traduzioni prodotte dal metodo di traduzione. Il risultato è stato di 950.000 traduzioni, ognuna della durata di 3 microsecondi, per un incremento totale di 3 secondi (circa il 5%) rispetto all'esecuzione nativa di Windows xp. Per ottenere questo risultato gli sviluppatori hanno fatto ricorso a diversi miglioramenti delle prestazioni che qui non saranno discussi. Per ulteriori informazioni consultate le note bibliografiche alla fine di questo capitolo.

18.4.3 Assistenza hardware

Senza un certo livello di supporto hardware la virtualizzazione sarebbe impossibile. Quanto maggiore è il supporto hardware disponibile all'interno di un sistema, tanto più stabili e ricche di funzionalità possono essere le macchine virtuali e tanto migliori possono risultare le prestazioni. A partire dal 2005 e per le generazioni successive Intel ha aggiunto alla famiglia di cpu Intel x86 il nuovo supporto della virtualizzazione (le istruzioni vt-x). Ora dunque non è più necessaria la traduzione binaria.

In pratica, tutte le principali cpu general-purpose stanno fornendo quantità sempre maggiori di supporto hardware per la virtualizzazione. Per esempio, la tecnologia di virtualizzazione amd (amd-V) è apparsa in numerosi processori amd a partire dal 2006. Questa tecnologia definisce due nuove modalità di funzionamento, host e guest: si passa da un processore a due modalità a un processore multimodale. Il vmm può attivare la modalità host, definire le caratteristiche di ogni macchina virtuale guest e quindi impostare il sistema in modalità guest, passando il controllo del sistema a un sistema operativo guest in esecuzione sulla macchina virtuale. In modalità guest il sistema operativo virtualizzato pensa di essere in esecuzione su hardware nativo e vede tutti i dispositivi inclusi nella definizione del guest fornita dall'host. Se il guest tenta di accedere a una risorsa virtualizzata il controllo viene passato al vmm per la gestione di tale interazione. Intel vt-x è simile a amd-V, in quanto fornisce le modalità root e nonroot, equivalenti alle modalità host e guest. Entrambe forniscono alla vcpu strutture di dati per il caricamento e la memorizzazione automatici dello stato della cpu durante i cambi di contesto del guest. Sono inoltre fornite le strutture di controllo della macchina virtuale (vmcs), per la gestione degli stati di host e guest e per i vari controlli di esecuzione del guest, i controlli di uscita e le informazioni sul perché i guest ritornano il controllo all'host. In quest'ultimo caso, per esempio, una violazione di tabella delle pagine nidificata causata da un tentativo di accedere a memoria indisponibile può provocare l'uscita del guest.

amd e Intel, nel loro ambiente virtuale, hanno anche indirizzato il problema della gestione della memoria. Con i miglioramenti nella gestione della memoria portati da rvi di amd e ept di Intel, il vmm non ha più necessità di implementare le tabelle npt via software. In sostanza, queste cpu implementano tabelle delle pagine nidificate nell'hardware per consentire al vmm di controllare completamente la paginazione mentre le cpu accelerano la traduzione di indirizzi virtuali in indirizzi fisici. Le npt aggiungono un nuovo livello che rappresenta il punto di vista del guest sulla traduzione da logico a fisico degli indirizzi. La funzione di walk sulle tabelle offerta dalla cpu include questo nuovo livello, permettendo di scorrere dalla tabella guest alla tabella vmm alla ricerca dell'indirizzo fisico desiderato. Un insuccesso della tlb porta a un degrado delle prestazioni, perché devono essere attraversate più tabelle (le tabelle delle pagine dell'host e del guest) per completare la ricerca. La Figura 18.4 mostra il lavoro di traduzione supplementare svolto dall'hardware per tradurre un indirizzo virtuale dell'ospite nel corrispondente indirizzo fisico.

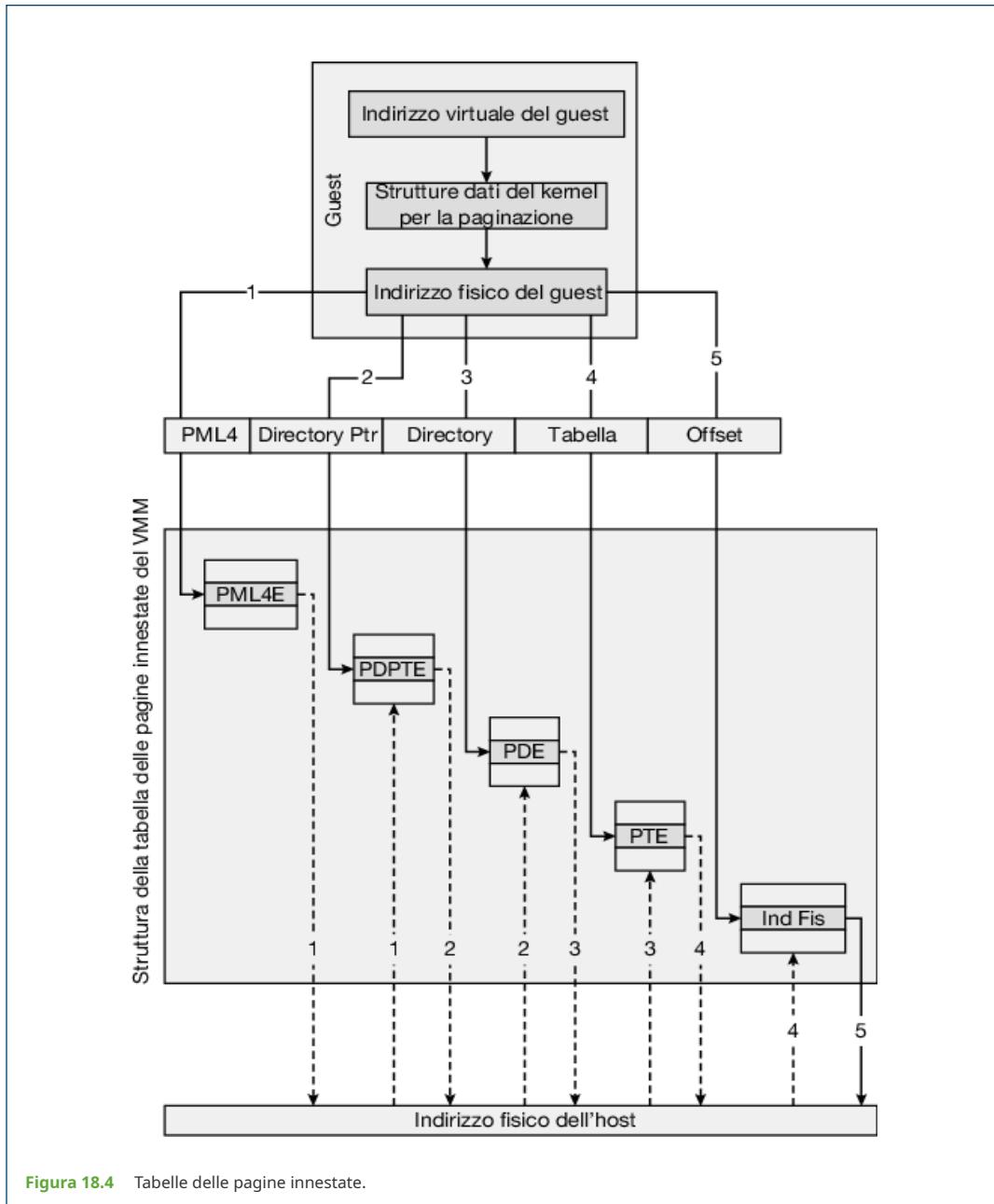


Figura 18.4 Tabelle delle pagine innestate.

L'i/o è un altro settore che ha subito miglioramenti grazie all'assistenza dell'hardware. Si consideri che il controller standard della memoria ad accesso diretto (dma) accetta un indirizzo di memoria di destinazione e un dispositivo sorgente di i/o e trasferisce i dati tra i due senza l'intervento del sistema operativo. Senza assistenza hardware, un guest potrebbe provare a impostare un trasferimento dma che interessa anche la memoria del vmm o di altri guest. In cpu che forniscono dma assistito dall'hardware (come le cpu Intel con vt-d), anche dma ha un livello di indirezione. In primo luogo, il vmm imposta domini di protezione per segnalare alla cpu quale parte della memoria fisica appartiene a ogni ospite. Successivamente, assegna i dispositivi di i/o ai domini di protezione, permettendo loro l'accesso diretto a quelle regioni di memoria (e solo a quelle). L'hardware trasforma poi l'indirizzo presente in una richiesta dma emessa da un dispositivo di i/o nell'indirizzo di memoria fisica dell'host associata all'i/o. In questo modo i trasferimenti dma vengono passati da un guest a un dispositivo senza l'interferenza del vmm.

Allo stesso modo, gli interrupt devono essere consegnati al guest appropriato e non devono essere visibili ad altri guest. Per mezzo di una funzione di rimappatura degli interrupt le cpu con assistenza hardware alla virtualizzazione inviano automaticamente un interrupt per un guest a un core dove è in esecuzione un thread di quel guest. In questo modo il guest riceve gli interrupt senza bisogno di un intervento del vmm. Senza la rimappatura degli interrupt i guest malevoli potrebbero generare interrupt allo scopo di ottenere il controllo del sistema host. Si vedano le Note bibliografiche alla fine di questo capitolo per maggiori dettagli.

Le architetture arm, in particolare arm v8 (64 bit), adottano un approccio leggermente diverso nel supporto hardware alla virtualizzazione, fornendo un intero livello di eccezioni (el2) con privilegi superiori a quelli del livello kernel (el1). Ciò consente l'esecuzione di un hypervisor isolato, con il proprio accesso alla mmu e una propria gestione delle interruzioni. Per consentire la paravirtualizzazione viene aggiunta un'istruzione speciale (hvc) che consente all'hypervisor di essere chiamato dal kernel dei guest. Questa istruzione può essere richiamata solo dalla modalità kernel (el1).

Un interessante effetto collaterale della virtualizzazione assistita dall'hardware è la possibilità di creare hypervisor leggeri. Un buon esempio è il framework di macos "HyperVisor.framework", una libreria fornita dal sistema operativo che consente la creazione di macchine virtuali con poche righe di codice. Il lavoro effettivo viene svolto tramite chiamate di sistema, che fanno in modo che l'esecuzione di istruzioni di virtualizzazione privilegiate sia invocata dal kernel per conto del processo hypervisor, consentendo la gestione di macchine virtuali senza che l'hypervisor debba caricare un modulo del kernel per eseguire le chiamate.

18.5 Tipologie di macchine virtuali e loro implementazioni

Abbiamo fin qui analizzato alcune delle tecniche utilizzate per implementare la virtualizzazione. Prendiamo ora in esame i principali tipi di macchine virtuali, la loro implementazione, le loro funzionalità e come vengono utilizzati i blocchi costituenti appena descritti per creare un ambiente virtuale. Naturalmente l'hardware su cui le macchine virtuali sono in esecuzione è causa di notevole variabilità tra i metodi di implementazione. Discuteremo qui le implementazioni in generale, lasciando sottointeso il fatto che il vmm può approfittare di assistenza hardware, ove disponibile.

18.5.1 Il ciclo di vita della macchina virtuale

Cominciamo la nostra trattazione con il ciclo di vita della macchina virtuale. Qualunque sia il tipo di hypervisor, al momento della creazione di una macchina virtuale vengono forniti determinati parametri al vmm. Questi parametri includono di solito il numero di cpu, la quantità di memoria, i dettagli della rete e dello storage. Di questi parametri il vmm terrà conto durante la creazione del guest. Un utente, per esempio, potrebbe voler creare un nuovo guest con due cpu virtuali, 4 gb di memoria, 10 gb di spazio su disco, una interfaccia di rete che ottiene il suo indirizzo ip via dhcp e l'accesso al drive dvd.

Il vmm crea la macchina virtuale secondo questi parametri. Nel caso di un hypervisor di tipo 0 le risorse sono generalmente dedicate. In questa situazione, se non esistono due cpu virtuali disponibili e non assegnate la richiesta di creazione fallisce. In altri casi, a seconda delle tipologie di hypervisor, le risorse possono essere dedicate o virtuali. Certamente, un indirizzo ip non può essere condiviso, ma le cpu virtuali sono di solito multiplexed (*multiplexed*) sulle cpu fisiche, come discusso nel Paragrafo 18.6.1. Allo stesso modo la gestione della memoria comporta solitamente l'allocazione ai guest di più memoria di quanta in realtà ne esista fisicamente. Questo meccanismo è più complicato, ed è descritto nel Paragrafo 18.6.2.

Infine, quando la macchina virtuale non è più necessaria può essere eliminata. Quando ciò accade, il vmm libera lo spazio disco utilizzato e poi rimuove la configurazione associata alla macchina virtuale, in sostanza dimenticando l'esistenza di questa macchina.

Questi passaggi sono abbastanza semplici in confronto alla costruzione, alla configurazione, all'esecuzione e alla rimozione di macchine fisiche. La creazione di una macchina virtuale a partire da una macchina virtuale esistente può consistere semplicemente nel fare clic sul pulsante “clone” e nel fornire un nuovo nome e un nuovo indirizzo ip. Questa semplicità nella creazione può portare a una *crescita disordinata delle macchine virtuali*, ovvero alla presenza su un sistema di così tante macchine virtuali che il loro utilizzo, la loro storia e il loro stato risultano confusi e difficili da monitorare.

18.5.2 Hypervisor di tipo 0

Gli hypervisor di tipo 0 sono esistiti per molti anni con diverse denominazioni, tra cui “partizioni” e “domini”. Si tratta di strumenti hardware e per questo vi sono lati positivi e negativi. I sistemi operativi non hanno bisogno di fare nulla di speciale per sfruttare le loro caratteristiche. Il vmm è codificato nel firmware, viene caricato al momento dell'avvio e, a sua volta, carica le immagini del guest da eseguire in ogni partizione. Le funzionalità di un hypervisor di tipo 0 tendono a essere meno di quelle degli altri tipi, perché è implementato in hardware. Per esempio, un sistema può essere suddiviso in quattro sistemi virtuali, ciascuno con cpu, memoria e dispositivi di i/o dedicati. Ogni guest ritiene di avere hardware dedicato proprio perché è così nella realtà. Ciò semplifica molti dettagli di implementazione.

L'i/o presenta qualche difficoltà, perché non è facile dedicare dispositivi di i/o ai guest se non ve ne sono a sufficienza. Che cosa succede se un sistema è dotato di due porte Ethernet e di più di due guest, per esempio? I casi sono due: o tutti gli ospiti devono avere propri dispositivi di i/o, o il sistema deve permettere la condivisione dei dispositivi. A seconda dei casi, l'hypervisor gestisce l'accesso condiviso oppure attribuisce i permessi di accesso di tutti i dispositivi a una partizione di controllo. Nella partizione di controllo un sistema operativo guest fornisce servizi (come il networking) ad altri guest per mezzo di demoni e l'hypervisor indirizza le richieste di i/o in modo appropriato. Alcuni hypervisor di tipo 0 sono ancora più sofisticati e possono muovere le cpu fisiche e la memoria tra i guest in esecuzione. In questi casi i guest sono paravirtualizzati, sono cioè consapevoli della virtualizzazione e contribuiscono alla sua esecuzione. Un guest, per esempio, deve monitorare i segnali provenienti dall'hardware o dal vmm che indicano un cambiamento hardware, sondare i propri dispositivi per rilevare il cambiamento e aggiungere o sottrarre cpu o memoria dalle proprie risorse disponibili.

Poiché la virtualizzazione di tipo 0 è molto vicina all'implementazione hardware cruda, andrebbe considerata separatamente dagli altri metodi discussi qui. Un hypervisor di tipo 0 può eseguire più sistemi operativi guest (uno in ogni partizione hardware). Tutti questi guest, poiché sono in esecuzione direttamente sull'hardware, possono essere a loro volta vmm. In sostanza, i sistemi operativi guest in un hypervisor di tipo 0 sono sistemi operativi nativi con un sottoinsieme di hardware a loro disposizione. Grazie a ciò, ciascuno può avere i propri sistemi operativi guest (Figura 18.5). Altri tipi di hypervisor di solito non sono in grado di fornire questa funzionalità di virtualizzazione nella virtualizzazione.

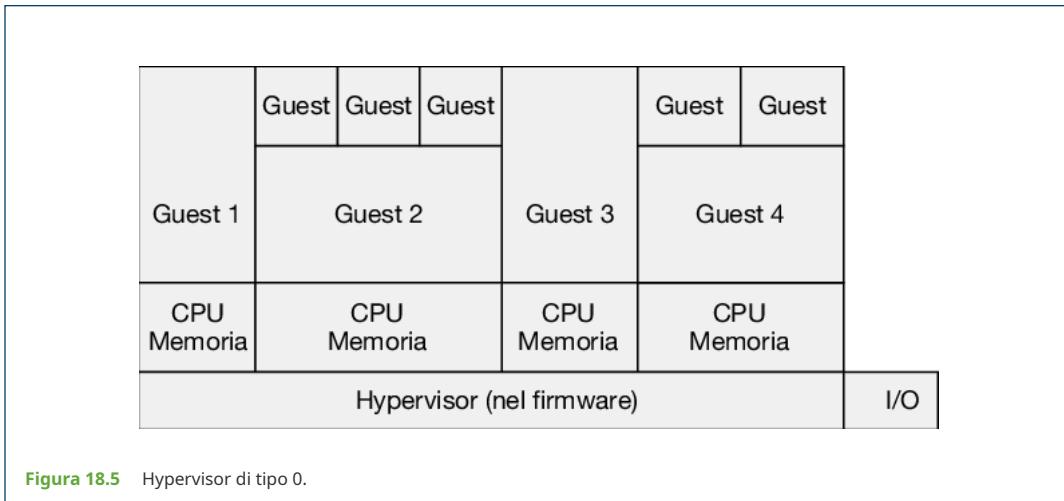


Figura 18.5 Hypervisor di tipo 0.

18.5.3 Hypervisor di tipo 1

Gli hypervisor di tipo 1 si trovano comunemente nei data center aziendali e stanno diventando in un certo senso “i sistemi operativi dei data center”. Si tratta di sistemi operativi per usi speciali che girano nativamente sull’hardware, ma piuttosto che fornire chiamate di sistema e altre interfacce per il funzionamento dei programmi, creano, eseguono e gestiscono sistemi operativi guest. Oltre a essere in esecuzione su hardware standard, essi possono essere eseguiti su hypervisor di tipo 0, ma non su hypervisor di tipo 1. Qualunque sia la piattaforma, in genere i guest non sanno di essere in esecuzione su qualcosa di diverso dall’hardware nativo.

Gli hypervisor di tipo 1 sono in esecuzione in modalità kernel e traggono vantaggio dalla protezione hardware. Quando la cpu host lo consente, usano molteplici modalità per permettere ai sistemi operativi guest il proprio controllo e per migliorare le prestazioni. Gli hypervisor di tipo 1 implementano i driver di periferica per l’hardware su cui girano, perché nessun altro componente potrebbe farlo. Visto che sono sistemi operativi, devono anche fornire scheduling della cpu, gestione della memoria, gestione dell’i/o, protezione e anche sicurezza. Spesso forniscono api a supporto di applicazioni dei guest o applicazioni esterne che forniscono funzionalità come backup, monitoraggio e sicurezza. Molti hypervisor di tipo 1 sono prodotti commerciali “closed source”, come per esempio vmware esx, mentre altri sono open source o ibridi, come per esempio Citrix XenServer e la sua controparte open source Xen.

Utilizzando hypervisor di tipo 1 i gestori dei data center possono controllare e gestire i sistemi operativi e le applicazioni in modi nuovi e sofisticati. Un vantaggio importante è la capacità di consolidare più sistemi operativi e applicazioni su un minor numero di sistemi. Per esempio, piuttosto che avere dieci sistemi utilizzati al 10% delle loro potenzialità, un data center potrebbe avere un unico server per la gestione dell’intero carico. Se cresce l’utilizzo, i guest e le loro applicazioni possono essere spostati su sistemi meno carichi senza interruzione del servizio. Con l’utilizzo di istantanee e clonazione il sistema può salvare gli stati dei guest e duplicarli, un compito molto più facile rispetto al ripristino da backup o all’installazione manuale o tramite script o altri strumenti. Il prezzo di questa maggiore gestibilità è il costo del vmm (se si tratta di un prodotto commerciale), la necessità di imparare nuovi strumenti e metodi di gestione e la maggiore complessità.

Un’altra tipologia di hypervisor di tipo 1 comprende vari sistemi operativi general-purpose dotati di funzionalità vmm. In questo caso, un sistema operativo come Linux RedHat Enterprise, Windows o Oracle Solaris svolge le sue normali funzioni, oltre a fornire un vmm per permettere ad altri sistemi operativi di essere eseguiti come guest. A causa delle loro funzioni supplementari, questi hypervisor in genere forniscono meno funzionalità di virtualizzazione di altri hypervisor di tipo 1. Essi trattano un sistema operativo guest come un normale processo, che gode inoltre di una gestione speciale nel caso in cui il guest tenti di eseguire istruzioni speciali.

18.5.4 Hypervisor di tipo 2

Gli hypervisor di tipo 2 sono meno interessanti per noi che siamo interessati ai sistemi operativi, perché c’è molto poco coinvolgimento del sistema operativo in questi gestori di macchine virtuali a livello applicativo. Questo tipo di vmm è semplicemente un processo eseguito e gestito dall’host e nemmeno l’host sa che è in atto una virtualizzazione all’interno del vmm.

Gli hypervisor di tipo 2 hanno dei limiti non riscontrabili in altri tipi. Per esempio, un utente ha bisogno di privilegi di amministratore per accedere a molte delle funzioni assistite dall’hardware nelle cpu moderne. Se il vmm viene eseguito da un utente standard senza privilegi aggiuntivi non può sfruttare queste caratteristiche. A causa di questa limitazione e del sovraccarico dato dall’esecuzione di un sistema operativo general-purpose insieme ad altri sistemi operativi guest, gli hypervisor di tipo 2 tendono ad avere prestazioni complessive più scarse rispetto agli hypervisor di tipo 0 o 1.

Come spesso accade, i limiti degli hypervisor di tipo 2 sono in parte compensati da alcuni vantaggi. Essi infatti possono essere eseguiti su una varietà di sistemi operativi general-purpose e la loro esecuzione non richiede nessuna modifica al sistema operativo host. Uno studente può utilizzare un hypervisor di tipo 2, per esempio, per testare un sistema operativo non nativo senza sostituire il sistema operativo nativo. In questo modo, uno studente potrebbe disporre su un portatile Apple di versioni di Windows, Linux, Unix e sistemi operativi meno comuni, tutti disponibili per l’apprendimento e la sperimentazione.

18.5.5 Paravirtualizzazione

Come abbiamo visto, la paravirtualizzazione segue una strada diversa rispetto agli altri tipi di virtualizzazione. Piuttosto che cercare di indurre un sistema operativo guest a credere di avere un sistema per sé, la paravirtualizzazione presenta all'ospite un sistema che è simile, ma non identico, al suo sistema preferenziale. Il guest deve essere modificato per funzionare sull'hardware virtuale paravirtualizzato. Il lavoro addizionale richiesto da queste modifiche offre in cambio un guadagno in termini di uso più efficiente delle risorse e snellezza del livello paravirtuale.

Il vmm Xen, leader nella paravirtualizzazione, ha implementato diverse tecniche per ottimizzare le prestazioni dei guest, così come del sistema host. Per esempio, come abbiamo già visto, alcuni vmm presentano agli ospiti dispositivi virtuali che sembrano essere dispositivi reali. Invece di seguire tale approccio, Xen presenta un'astrazione pulita e semplice dei dispositivi che consente un i/o efficiente e una buona comunicazione tra il guest e il vmm per ciò che concerne il dispositivo di i/o. Per ogni dispositivo utilizzato da ogni guest c'è un buffer circolare condiviso tra il guest e il vmm tramite memoria condivisa. I dati di lettura e scrittura vengono inseriti in questo buffer, come mostrato nella Figura 18.6.

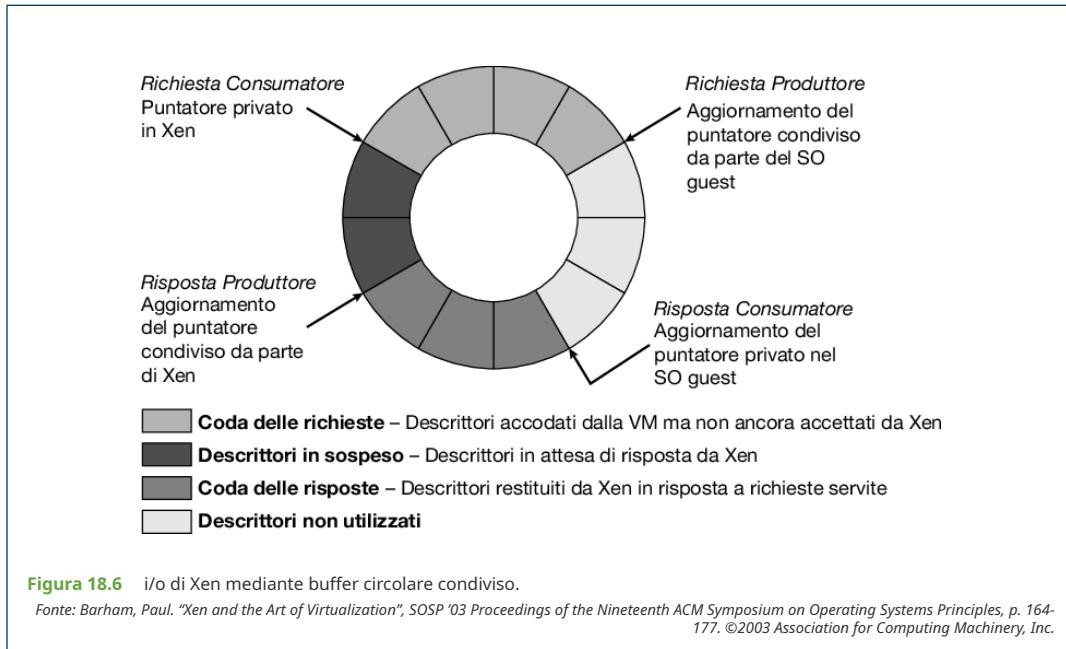


Figura 18.6 i/o di Xen mediante buffer circolare condiviso.

Fonte: Barham, Paul. "Xen and the Art of Virtualization", SOSP '03 Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, p. 164-177. ©2003 Association for Computing Machinery, Inc.

Per la gestione della memoria Xen non implementa tabelle delle pagine nidificate, ma ogni guest ha il proprio insieme di tabelle delle pagine, impostate in sola lettura. Quando è necessario un cambio di pagina della tabella, Xen richiede agli ospiti di usare un meccanismo specifico, una hypercall da parte del guest verso il vmm hypervisor. Il codice del kernel del sistema operativo guest deve quindi essere modificato per supportare questi metodi specifici di Xen. Per ottimizzare le prestazioni Xen permette ai guest di accodare modifiche multiple a una tabella delle pagine in modo asincrono tramite hypercall e quindi effettua controlli per assicurare che le modifiche siano completate prima di continuare le operazioni.

Xen permette la virtualizzazione delle cpu x86 senza l'uso di una traduzione binaria, richiedendo però alcune modifiche nei sistemi operativi guest, come descritto in precedenza. Nel corso del tempo, Xen ha saputo sfruttare le caratteristiche hardware a supporto della virtualizzazione. Come risultato, esso non richiede più la modifica dei guest ed essenzialmente non necessita più del meccanismo di paravirtualizzazione. La paravirtualizzazione è comunque ancora usata in altre soluzioni, per esempio negli hypervisor di tipo 0.

18.5.6 Virtualizzazione dell'ambiente di programmazione

Un altro tipo di virtualizzazione, basato su un modello di esecuzione diverso, è la *virtualizzazione* degli ambienti di programmazione. In questo caso un linguaggio di programmazione viene progettato per andare in esecuzione su un ambiente virtualizzato costruito ad hoc. Per esempio, Java di Oracle ha molte caratteristiche che dipendono dalla sua esecuzione nella macchina virtuale Java (jvm), tra cui i metodi specifici per la gestione della sicurezza e della memoria.

Se per virtualizzazione intendiamo esclusivamente la duplicazione di hardware, allora questo metodo non può essere realmente considerato come virtualizzazione. Non limitandoci a tale restrizione, possiamo definire un ambiente virtuale, basato su api, che fornisce un insieme di caratteristiche messe a disposizione di un particolare linguaggio e a programmi scritti in quel linguaggio. I programmi Java sono eseguiti all'interno dell'ambiente jvm e la jvm è compilata come un programma nativo sui sistemi su cui viene eseguita. Ciò significa che i programmi Java vengono scritti una sola volta e possono poi essere eseguiti su qualsiasi sistema.

(compresi tutti i principali sistemi operativi) su cui è disponibile una jvm. Lo stesso vale per i linguaggi interpretati, eseguiti all'interno di programmi che leggono ogni istruzione e la interpretano in operazioni native.

18.5.7 Emulazione

La virtualizzazione è probabilmente il metodo più comune per eseguire applicazioni progettate per un sistema operativo su un sistema operativo diverso, ma sulla stessa cpu. Questo metodo è abbastanza efficiente, perché le applicazioni sono state compilate per lo stesso insieme di istruzioni utilizzato dal sistema di destinazione.

Che cosa succede quando si ha bisogno di eseguire un'applicazione o un sistema operativo su una cpu diversa? In questo caso è necessario convertire tutte le istruzioni della cpu sorgente in istruzioni equivalenti della cpu destinazione. Un tale ambiente non è più virtualizzato, ma è piuttosto completamente emulato.

L'emulazione è utile quando il sistema host ha un'architettura di sistema che è diversa da quella per cui è stato compilato il sistema guest. Si supponga, per esempio, che una società abbia sostituito il suo sistema informatico obsoleto con un nuovo sistema, ma voglia continuare a eseguire alcuni importanti programmi che sono stati compilati per il vecchio sistema. I programmi possono essere eseguiti all'interno di un emulatore che converte tutte le istruzioni del sistema obsoleto nelle istruzioni native del nuovo sistema. L'emulazione può prolungare la vita dei programmi e ci permette di esplorare le vecchie architetture senza bisogno di possedere realmente una macchina con tale architettura.

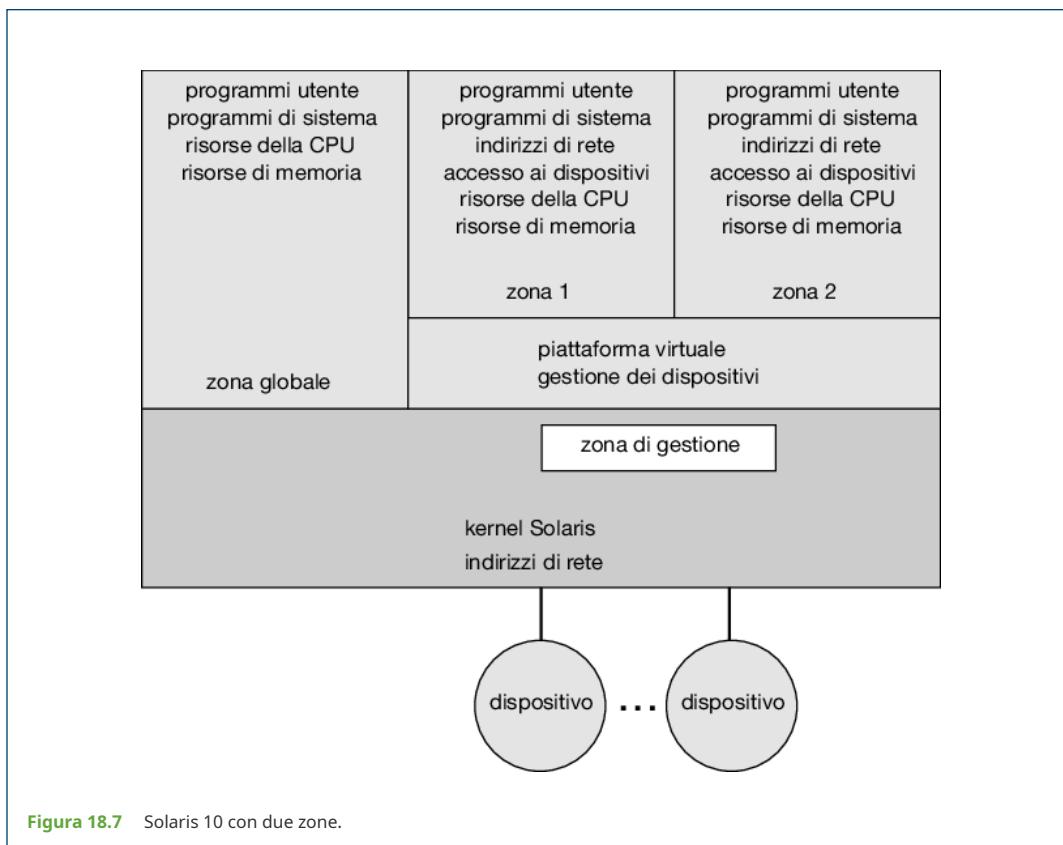
Come ci si può aspettare, il grande problema dell'emulazione sono le prestazioni. L'emulazione di un insieme di istruzioni viene eseguita molto più lentamente rispetto alle istruzioni native, perché possono essere necessarie anche dieci istruzioni sul nuovo sistema per leggere, analizzare e simulare un'istruzione dal vecchio sistema. Così, a meno che la nuova macchina sia dieci volte più veloce rispetto alla vecchia, il programma in esecuzione sulla nuova macchina sarà eseguito più lentamente di quanto avveniva sul suo hardware nativo. Un altro ostacolo per i programmatore di emulatori sta nella difficoltà di creare un emulatore corretto, perché, in sostanza, il loro lavoro comporta la scrittura di un'intera cpu via software.

Nonostante queste difficoltà l'emulazione è molto popolare, soprattutto nell'ambito dei videogiochi. Molti famosi videogiochi sono state scritti per piattaforme che ora non sono più in produzione. Gli utenti che desiderano eseguire quei giochi possono spesso trovare un emulatore della piattaforma interessata ed eseguire il gioco all'interno dell'emulatore senza doverlo modificare. I sistemi moderni sono così veloci rispetto alle vecchie console da gioco che esiste un emulatore di giochi persino per iPhone e vi sono giochi disponibili da eseguire al suo interno.

18.5.8 Contenitori di applicazioni

L'obiettivo della virtualizzazione è in alcuni casi quello di fornire un metodo per separare le applicazioni, controllare le loro prestazioni e il loro utilizzo delle risorse e creare un modo semplice per avviarle, arrestarle, spostarle e gestirle. In questi casi non è probabilmente necessaria una vera e propria virtualizzazione, in particolare se le applicazioni sono tutte compilate per lo stesso sistema operativo. Possiamo in alternativa utilizzare dei contenitori di applicazioni.

Consideriamo un esempio di contenitore di applicazioni. A partire dalla versione 10, Oracle Solaris ha incluso contenitori (*container*), o zone, che creano uno strato virtuale tra il sistema operativo e le applicazioni. In questo sistema è installato un unico kernel e l'hardware non è virtualizzato. A essere virtualizzato è il sistema operativo con i suoi dispositivi, in modo da offrire ai processi una zona in cui ognuno abbia l'impressione di essere l'unico processo in esecuzione sul sistema. Possono essere create una o più zone, ognuna con le proprie applicazioni, il proprio indirizzo di rete, le proprie porte, i propri account utente, e così via. Le risorse di cpu e memoria possono essere ripartite tra le zone e i processi di sistema. Ogni zona è in grado di eseguire il proprio scheduler per ottimizzare le prestazioni delle applicazioni sulle risorse assegnate. La Figura 18.7 mostra un sistema Solaris 10 con due zone e lo spazio utente standard, chiamato "globale".



I contenitori sono molto più leggeri rispetto ad altri metodi di virtualizzazione, usano meno risorse di sistema, sono più veloci da istanziare e distruggere e sono più simili ai processi rispetto alle macchine virtuali. Per questo motivo sono ora sempre più usati, specialmente nel cloud computing. Freebsd è stato forse il primo sistema operativo a includere una funzionalità simile a un contenitore (chiamata "jail") e anche aix dispone di una funzione simile. Linux ha introdotto nel 2014 il contenitore lxc, che è ora fornito nelle comuni distribuzioni Linux tramite un flag nella chiamata di sistema `clone()`. Il codice sorgente di lxc è disponibile su <https://linuxcontainers.org/lxc/downloads>.

I contenitori sono inoltre facili da automatizzare e gestire; per farlo sono stati sviluppati strumenti di orchestrazione come docker e Kubernetes. Si tratta di strumenti utili per automatizzare e coordinare sistemi e servizi. Il loro scopo è quello di semplificare l'esecuzione di intere suite di applicazioni distribuite, proprio come i sistemi operativi semplificano l'esecuzione di un singolo programma. Questi strumenti offrono un deployment rapido di applicazioni complete, consistenti in numerosi processi all'interno dei contenitori, e forniscono anche funzioni di monitoraggio e di amministrazione. Per ulteriori informazioni su docker si veda <https://www.docker.com/what-docker>. Informazioni su Kubernetes sono disponibili su <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes>.

18.6 Virtualizzazione e componenti dei sistemi operativi

Abbiamo finora esplorato i blocchi costituenti necessari per la virtualizzazione e i vari tipi di virtualizzazione. In questo paragrafo ci immergeremo negli aspetti della virtualizzazione che riguardano il sistema operativo. Tra questi vedremo come il vmm fornisce funzioni di base del sistema operativo, come lo scheduling e la gestione di i/o e memoria. Risponderemo in particolare a domande come le seguenti. Come effettua lo scheduling della cpu il vmm, quando i sistemi operativi guest credono di avere cpu dedicate? Come può funzionare la gestione della memoria quando molti guest richiedono grandi quantità di memoria?

18.6.1 Scheduling della cpu

Un sistema con la virtualizzazione, anche un sistema a singola cpu, si comporta spesso come un sistema multiprocessore. Il software di virtualizzazione presenta una o più cpu virtuali a ciascuna delle macchine virtuali in esecuzione sul sistema e poi pianifica l'utilizzo delle cpu fisiche tra le macchine virtuali.

Le differenze significative tra le tecnologie di virtualizzazione rendono difficile riassumere l'effetto della virtualizzazione sullo scheduling. Consideriamo in primo luogo il caso generale dello scheduling del vmm. Il vmm ha un certo numero di cpu fisiche disponibili e un certo numero di thread da eseguire su tali cpu. I thread possono essere del vmm o dei guest. I guest sono configurati con un certo numero di cpu virtuali al momento della creazione, ma questo numero può essere regolato nel corso dell'esistenza della vm. Quando ci sono abbastanza cpu per assegnare il numero richiesto da ogni guest, il vmm può trattare la cpu come dedicata e pianificare su di essa solo i thread del guest che la possiede. In questa situazione i guest si comportano proprio come sistemi operativi nativi in esecuzione su cpu native.

Naturalmente, in altre situazioni, il numero di cpu potrebbe non essere sufficiente. Lo stesso vmm ha bisogno di alcuni cicli di cpu per la gestione dei guest e dell'i/o e può rubare cicli ai guest pianificando i suoi thread su qualsiasi cpu del sistema, ma l'impatto di questa azione è relativamente basso. Più difficile da trattare è il caso di sovraassegnazione (*overcommitment*), in cui i guest sono configurati per usare più cpu di quante ne esistano nel sistema. In questo caso un vmm può utilizzare algoritmi standard di scheduling per far progredire ogni thread, ma può anche aggiungere elementi di equità a tali algoritmi. Per esempio, se ci fossero sei cpu hardware e 12 cpu assegnate ai guest, il vmm potrebbe allocare le risorse della cpu in modo proporzionale, dando a ciascun guest la metà delle risorse delle cpu che crede di avere. Il vmm può comunque presentare ai guest tutte e 12 le cpu virtuali, ma nella loro mappatura sulle cpu fisiche il vmm può utilizzare il proprio scheduler per condividere le cpu in modo appropriato.

Anche quando si utilizza uno scheduler che fornisce equità, qualsiasi algoritmo di schedulazione del sistema operativo guest che assume di ottenere una certa quantità di progresso nella elaborazione in un determinato periodo di tempo sarà influenzato negativamente dalla virtualizzazione. Si consideri un sistema operativo time-sharing che tenta di assegnare 100 millisecondi per ogni intervallo di tempo per garantire agli utenti un tempo di risposta ragionevole. All'interno di una macchina virtuale questo sistema operativo è in balia del sistema di virtualizzazione per quanto riguarda le risorse della cpu che effettivamente riceve. Un determinato intervallo di tempo di 100 millisecondi può richiedere molto di più di 100 millisecondi di tempo della cpu virtuale. A seconda di quanto il sistema è impegnato, l'intervallo di tempo può arrivare a un secondo o più, con tempi di risposta molto scarsi per gli utenti registrati su quella macchina virtuale. L'effetto su un sistema operativo real-time può essere ancora peggiore.

L'effetto netto di questo scheduling stratificato è che i singoli sistemi operativi virtualizzati ricevono solo una parte dei cicli di cpu disponibili, anche se credono di poter ricevere tutti i cicli e, anzi, credono di essere responsabili della pianificazione di tutti quei cicli. Capita di frequente che gli orologi time-of-day delle macchine virtuali non siano corretti, perché i timer impiegano più tempo a scattare rispetto a quanto avverrebbe su cpu dedicate. La virtualizzazione può quindi vanificare gli sforzi per una buona realizzazione degli algoritmi di scheduling nei sistemi operativi all'interno di macchine virtuali.

Per correggere questi problemi un vmm avrà, per ogni tipo di sistema operativo che gli amministratori intendono installare come guest, un'applicazione in grado di prevenire i ritardi dell'orologio di sistema, dotata anche di altre funzioni come la gestione virtuale delle periferiche.

18.6.2 Gestione della memoria

L'uso efficiente della memoria nei sistemi operativi general-purpose è una delle chiavi principali per ottenere prestazioni migliori. In ambienti virtualizzati la memoria ha più utenti (i guest e le loro applicazioni, nonché il vmm) e si induce quindi una maggior pressione sul suo utilizzo. Un ulteriore fattore di pressione è il fatto che il vmm tipicamente sovraassegna (overcommit) la memoria, in modo che la memoria totale secondo le configurazioni dei guest supera la quantità di memoria fisicamente presente nel sistema. Il maggior bisogno di un utilizzo efficiente della memoria non viene dimenticato dagli implementatori di vmm, che prendono importanti misure atte a garantirne un utilizzo ottimale.

vmware esx, per esempio, utilizza almeno tre metodi di gestione della memoria. Prima di occuparsi dell'ottimizzazione della memoria, il vmm deve stabilire quanta memoria reale deve utilizzare ogni guest. Per fare ciò, il vmm valuta prima la quantità massima di memoria di ogni guest secondo la sua configurazione. Dato che i sistemi operativi general-purpose non si aspettano variazioni nella quantità di memoria del sistema, i vmm devono mantenere l'illusione che il guest abbia a disposizione quella quantità di memoria. Successivamente, il vmm calcola un obiettivo per l'allocazione di memoria reale a ogni guest in base alla memoria configurata per quel guest e ad altri fattori, quali la sovraassegnazione di memoria e il carico del sistema. Il vmm utilizza quindi i tre meccanismi di basso livello descritti di seguito per recuperare la memoria dai guest.

1. Ricordate che un ospite crede di controllare l'allocazione della memoria tramite la gestione della sua tabella delle pagine, mentre in realtà il vmm mantiene una tabella delle pagine nidificata che ritraduce la tabella delle pagine del guest nella tabella delle pagine reale. Il vmm può utilizzare questo ulteriore livello per ottimizzare l'utilizzo della memoria da parte dei guest senza la conoscenza o l'aiuto da parte dei guest. Un approccio è quello di fornire una doppia paginazione, in cui il vmm ha i propri algoritmi di sostituzione delle pagine e utilizza pagine sul backing-store facendo credere al guest che queste si trovino nella memoria fisica. Naturalmente, il vmm ha meno conoscenza dei modelli di accesso alla memoria del guest rispetto al guest

stesso, dunque la sua paginazione è meno efficiente e crea problemi di prestazioni. I vmm utilizzano questo metodo quando non ce ne sono altri disponibili, o quando gli altri metodi non permettono di avere sufficiente memoria libera. Tuttavia, questo non è l'approccio preferito.

2. Una soluzione comune è la seguente. Il vmm installa in ogni guest uno pseudo-driver di periferica o un modulo kernel su cui ha il controllo. (Uno pseudo-driver di periferica utilizza interfacce di driver di periferica e appare al kernel come un vero driver, ma in realtà non controlla nessun dispositivo). Si tratta piuttosto di un modo semplice per aggiungere codice in modalità kernel senza modificare direttamente il kernel). Questo gestore di memoria, chiamato balloon, comunica con il vmm e riceve comandi di allocazione o deallocazione di memoria. Quando il comando è di allocazione, il balloon alocca la memoria e chiede al sistema operativo di fissare (*pinning*) nella memoria fisica le pagine allocate (ricordiamo che il pinning blocca una pagina nella memoria fisica in modo che non possa essere spostata o tolta dalla memoria). Il guest avverte la pressione dovuta alla presenza di queste pagine appuntate e reagisce, in sostanza, diminuendo la quantità di memoria fisica che ha a disposizione. Il guest potrà quindi essere indotto a liberare memoria fisica per essere sicuro di avere una quantità sufficiente di memoria libera. Nel frattempo il vmm, sapendo che le pagine bloccate non saranno mai utilizzate dal processo balloon, toglie queste pagine fisiche dal guest e le assegna a un altro guest. Allo stesso tempo il guest utilizza i propri algoritmi di gestione della memoria e paginazione per gestire la memoria disponibile in maniera più efficiente. Se l'utilizzo della memoria nell'intero sistema diminuisce il vmm dirà al processo balloon all'interno dell'ospite di sbloccare e liberare in tutto o in parte la memoria, concedendo al guest un maggior numero di pagine da utilizzare.
3. Un altro metodo utilizzato comunemente per ridurre la pressione sulla memoria consiste nel determinare se la stessa pagina è stata caricata più di una volta. In tal caso il vmm elimina tutte le copie della pagina tranne una e ridirige tutti gli utenti della pagina all'unico esemplare rimasto. vmware, per esempio, campiona casualmente la memoria di un guest e crea un hash per ogni pagina campionata. Questo valore hash è un'identificazione (*thumbprint*) della pagina. L'hash di ogni pagina esaminata viene confrontato con gli altri hash già memorizzati in una tabella hash. Se c'è una corrispondenza, le pagine vengono confrontate byte per byte per vedere se sono davvero identiche e se lo sono una pagina viene liberata e il suo indirizzo logico viene mappato sull'altro indirizzo fisico. Questa tecnica potrebbe sembrare, a prima vista, inefficace, ma occorre considerare il fatto che i guest eseguono sistemi operativi. Quando più guest eseguono lo stesso sistema operativo è sufficiente avere in memoria solo una sola copia delle pagine attive del sistema operativo. Analogamente, lo stesso insieme di applicazioni potrebbe essere in esecuzione su diversi guest, portando anche in questo caso a una possibile fonte di condivisione di memoria.

L'effetto complessivo è quello di consentire ai guest di comportarsi come se avessero l'intera quantità di memoria richiesta anche se in realtà ne hanno meno.

18.6.3 I/O

Per quanto concerne l'i/o, gli hypervisor hanno un certo margine di manovra e possono essere meno preoccupati di rappresentare esattamente l'hardware sottostante ai loro guest. A causa di tutte le variazioni nei dispositivi di i/o, i sistemi operativi sono abituati a servirsi di meccanismi di i/o vari e flessibili. I sistemi operativi sono dotati per esempio di un meccanismo per fornire un'interfaccia uniforme a qualunque dispositivo di i/o sottostante. Le interfacce dei driver di i/o sono progettate per consentire ai produttori di hardware di terze parti di fornire i driver di periferica che collegano i loro dispositivi al sistema operativo. Di solito i driver di periferica possono essere caricati e scaricati dinamicamente. La virtualizzazione si avvale di questa flessibilità integrata, fornendo specifici dispositivi virtualizzati per sistemi operativi guest.

Come descritto nel Paragrafo 18.5, i vmm differiscono notevolmente nel modo di fornire i/o ai loro guest. I dispositivi di i/o possono essere dedicati ai guest, per esempio, oppure il vmm può disporre di driver di periferica su cui mappare l'i/o dei guest. Il vmm può anche fornire ai guest dei driver di periferica idealizzati, permettendo così una facile erogazione e gestione dei servizi di i/o dei guest. In questo caso, l'ospite vede un dispositivo facile da controllare, ma in realtà questo semplice driver di dispositivo comunica con il vmm che invia le richieste a un dispositivo reale più complesso attraverso un driver reale più complesso. L'i/o in ambienti virtuali è complicato e richiede un'attenta progettazione e implementazione del vmm.

Consideriamo il caso di un hypervisor e di una combinazione hardware che permette ai dispositivi di essere dedicati a un guest e consente ai guest di accedere direttamente a tali dispositivi. Un dispositivo dedicato a un guest non è ovviamente a disposizione di altri guest, ma questo accesso diretto può comunque essere utile in alcune circostanze. La ragione per consentire l'accesso diretto è di migliorare le prestazioni di i/o: meno deve fare l'hypervisor per permettere l'i/o ai suoi guest, più veloce può essere l'i/o. Nel caso degli hypervisor di tipo 0 che forniscono accesso diretto al dispositivo, i guest possono spesso ottenere le stesse velocità dei sistemi operativi nativi. Quando invece un hypervisor di tipo 0 fornisce dispositivi condivisi, le prestazioni possono risentirne.

Utilizzando l'accesso diretto al dispositivo in hypervisor di tipo 1 e 2 le prestazioni possono essere simili a quelle dei sistemi operativi nativi se si è in presenza di un supporto hardware. L'hardware deve fornire dma pass-through, con servizi come vt-d, oltre alla consegna diretta di interrupt a specifici guest. Data la frequenza con cui si verificano gli interrupt, non dovrebbe sorprendere che su un hardware senza queste caratteristiche i guest hanno prestazioni peggiori di quelle che avrebbero se fossero in esecuzione nativamente.

Oltre all'accesso diretto, i vmm forniscono l'accesso condiviso ai dispositivi. Si consideri un disco a cui accedono più guest. Nel condividere il dispositivo il vmm deve fornire una certa protezione, assicurando che un guest possa accedere solo ai blocchi specificati nella sua configurazione. In queste situazioni, il vmm deve prendere parte in ogni i/o, verificando la correttezza e instradando i dati nella comunicazione tra i guest e gli opportuni dispositivi.

I vmm devono intervenire anche nella gestione della rete. I sistemi operativi general-purpose hanno in genere un solo indirizzo ip, anche se a volte ne hanno più d'uno, per esempio per connettersi a una rete di gestione, di backup e di produzione. Con la virtualizzazione, ogni ospite ha bisogno di almeno un indirizzo ip, perché questa è la modalità principale di comunicazione del guest. Un server che esegue un vmm può avere quindi decine di indirizzi. Il vmm agisce come uno switch virtuale che instrada i pacchetti di rete verso il guest destinatario.

I guest possono essere "direttamente" collegati alla rete mediante un indirizzo ip visibile dall'esterno (bridging) o, in alternativa, il vmm può fornire un indirizzo nat (*network address translation*). L'indirizzo nat è locale al server su cui il guest è in esecuzione e il vmm fornisce l'instradamento tra la rete esterna e il guest. Il vmm fornisce inoltre servizi di firewalling, mediando le connessioni tra i guest all'interno del sistema e tra i guest e i sistemi esterni.

18.6.4 Gestione dello storage

Una questione importante nel determinare come funziona la virtualizzazione è questa: se sono stati installati più sistemi operativi, cos'è e dov'è il disco di avvio? Gli ambienti virtualizzati devono chiaramente avere un approccio diverso nella gestione dello storage rispetto ai sistemi operativi nativi. Anche il metodo multiboot, che prevede il partizionamento del disco di root, l'installazione di un boot manager in una partizione e l'installazione di ogni altro sistema operativo in altre partizioni non è sufficiente, perché il partizionamento ha dei limiti che impedirebbero di lavorare con decine o centinaia di macchine virtuali.

La soluzione a questo problema dipende ancora una volta dal tipo di hypervisor. Gli hypervisor di tipo 0 tendono a consentire il partizionamento del disco di root, in parte perché questi sistemi tendono a eseguire un minor numero di guest rispetto ad altri sistemi. In alternativa, essi possono avere un gestore del disco come parte della partizione di controllo. Il gestore del disco fornisce spazio disco (compresi i dischi di avvio) alle altre partizioni.

Gli hypervisor di tipo 1 memorizzano il disco di root del guest (insieme alle informazioni di configurazione) in uno o più file all'interno dei file system forniti dal vmm. Gli hypervisor di tipo 2 memorizzano le stesse informazioni nel file system del sistema operativo host. In sostanza, un'immagine del disco con tutto il contenuto del disco principale del guest è memorizzata in un file del vmm. A parte per i potenziali problemi di prestazioni che ciò comporta, si tratta di una soluzione intelligente, perché semplifica la copia e lo spostamento dei guest. Se l'amministratore vuole un duplicato di un guest (per esempio per effettuare un test), deve semplicemente copiare l'immagine del guest e segnalare al vmm la presenza della nuova copia. L'avvio della nuova vm permette di avere a disposizione un guest identico. Spostare una macchina virtuale da un sistema a un altro che esegue lo stesso vmm è semplice quanto arrestare il guest, copiare l'immagine nell'altro sistema e avviare il guest nel nuovo ambiente.

I guest hanno a volte bisogno di più spazio su disco rispetto a quanto ve ne sia disponibile nella loro immagine. Per esempio, un server di database non virtualizzato potrebbe utilizzare diversi file system distribuiti su più dischi per memorizzare varie parti del database. La virtualizzazione di una simile banca dati di solito comporta la creazione di diversi file che il vmm presenta ai guest come dischi. Il guest viene eseguito normalmente e il vmm traduce le richieste di i/o provenienti dal guest in comandi di i/o per i file corretti.

Spesso i vmm forniscono un meccanismo per catturare un sistema fisico con la sua attuale configurazione e convertirlo in un guest che sono in grado di gestire ed eseguire. Sulla base di quanto descritto in precedenza, dovrebbe essere chiaro che questa conversione da fisico a virtuale (*P-to-V, Physical-to-Virtual*) legge i blocchi dei dischi del sistema fisico e li memorizza all'interno dei file sul sistema del vmm oppure su storage condiviso a cui il vmm può accedere. Non è forse così evidente la necessità di una procedura da virtuale a fisico (*V-to-P, Virtual-to-Physical*) per convertire un guest in un sistema fisico. Questa operazione è talvolta necessaria per il debugging: un problema potrebbe essere causato dal vmm o dai componenti a esso associati e l'amministratore può tentare di risolvere il problema rimuovendo la virtualizzazione dalle possibili cause. La conversione V-to-P può prendere i file contenenti tutti i dati del guest e generare blocchi sul disco di sistema, ricreando il guest come un sistema operativo nativo con le sue applicazioni. Una volta che il test si è concluso, il sistema nativo può essere riutilizzato per altri scopi quando la macchina virtuale ritorna in servizio oppure può essere eliminata la macchina virtuale lasciando che il sistema nativo continui a funzionare.

18.6.5 Migrazione in tempo reale

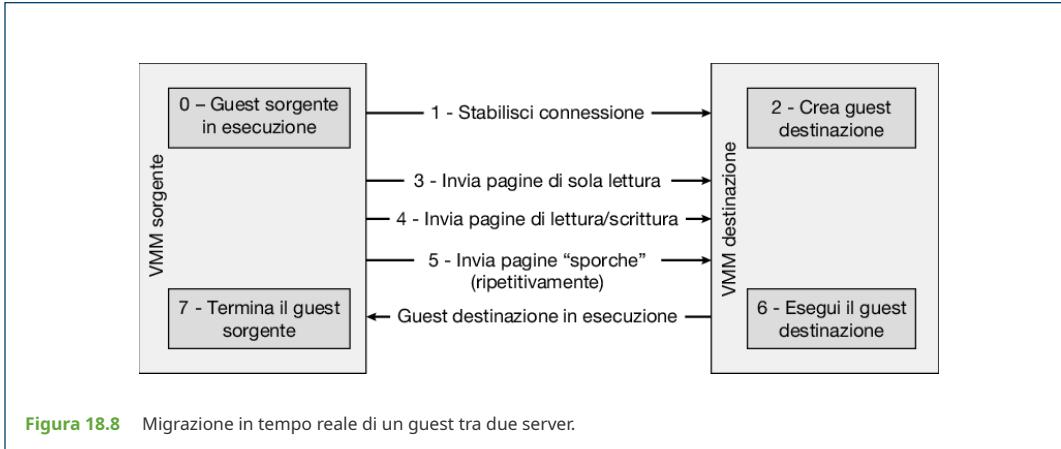
Una caratteristica che non si trova in sistemi operativi general-purpose, ma è presente negli hypervisor di tipo 0 e di tipo 1, è la migrazione in tempo reale (*live migration*) da un sistema all'altro di un guest in esecuzione. Abbiamo accennato a questa funzionalità in precedenza. Vediamo ora in dettaglio come funziona la migrazione in tempo reale e perché i vmm riescono a realizzarla in maniera relativamente semplice, mentre i sistemi operativi general-purpose, nonostante le ricerche e alcuni tentativi, non ne sono in grado.

Consideriamo innanzitutto il funzionamento della migrazione in tempo reale. Un guest in esecuzione su un sistema viene copiato su un altro sistema che esegue lo stesso vmm. La copia avviene con una così breve interruzione del servizio che gli utenti connessi al guest e le connessioni di rete continuano senza subire un significativo impatto negativo. Questa capacità piuttosto sorprendente è molto efficace nella gestione delle risorse e nell'amministrazione dell'hardware. Dopo tutto, basta fare un confronto con i passi necessari in assenza di virtualizzazione: avvisare gli utenti, arrestare i processi, spostare eventualmente i file binari e riavviare i processi sul nuovo sistema in modo che gli utenti, solo a questo punto, siano in grado di utilizzare nuovamente i servizi. Con la migrazione in tempo reale è possibile alleggerire un sistema sovraccarico o effettuare modifiche hardware o di sistema senza alcuna interruzione percepibile agli utenti.

La migrazione in tempo reale è resa possibile dalle interfacce ben definite tra guest e vmm e dalle limitate informazioni di stato che il vmm mantiene per il guest. Il vmm migra un ospite implementando le seguenti fasi.

1. Il vmm sorgente stabilisce una connessione con il vmm destinazione e riceve l'autorizzazione a inviare un guest.
2. La destinazione crea un nuovo guest, mediante la creazione, tra l'altro, di una nuova vcpu e di una nuova tabella delle pagine annidata.
3. La sorgente invia tutte le pagine di memoria di sola lettura alla destinazione.
4. La sorgente invia tutte le pagine di lettura-scrittura alla destinazione, contrassegnandole come "pulite".
5. La sorgente ripete il passo 4, perché in quella fase alcune pagine sono state probabilmente modificate dal guest e sono ora "sporche". Queste pagine devono essere inviate nuovamente e contrassegnate di nuovo come "pulite".
6. Quando il ciclo delle fasi 4 e 5 diventa molto breve, il vmm sorgente blocca il guest, invia lo stato finale della vcpu e altri dettagli sullo stato, invia le ultime pagine "sporche" e dice alla destinazione di avviare l'esecuzione del guest. Una volta che la destinazione segnala alla sorgente che il guest è in esecuzione, la sorgente può terminarlo.

Tale sequenza è mostrata nella Figura 18.8.



Concludiamo questa discussione con alcuni dettagli interessanti, evidenziando anche alcuni limiti della migrazione in tempo reale. Innanzitutto, affinché le connessioni di rete possano continuare senza interruzioni l'infrastruttura di rete deve accettare il fatto che un indirizzo mac – l'indirizzo dell'hardware di rete – può muoversi tra i sistemi. Prima della virtualizzazione questo non accadeva, perché l'indirizzo mac era legato all'hardware fisico. Con la virtualizzazione il mac deve essere mobile per permettere alle connessioni di rete esistenti di continuare senza necessità di un ripristino. Gli switch di rete moderni sono in grado di gestire questa mobilità e instradare il traffico ovunque si trovi l'indirizzo mac, seguendo dunque gli spostamenti.

Un limite della migrazione in tempo reale sta nel fatto che nessuno stato del disco viene trasferito. Uno tra i fattori che rendono possibile la migrazione in tempo reale è che la maggior parte dello stato del guest viene mantenuto all'interno del guest stesso – per esempio, la tabella dei file aperti, lo stato delle chiamate di sistema, lo stato del kernel, e così via. Siccome l'i/o su disco è molto più lento rispetto all'accesso alla memoria e lo spazio utilizzato su disco è di solito molto più grande rispetto alla memoria utilizzata, i dischi associati a un guest non possono essere spostati in una migrazione in tempo reale. Il disco del guest viene mantenuto in remoto e vi si accede attraverso la rete. In questo caso, lo stato di accesso al disco è mantenuto all'interno del guest e l'unica cosa che conta per il vmm sono le connessioni di rete. Visto che le connessioni di rete sono mantenute durante la migrazione, l'accesso al disco remoto può continuare. Per memorizzare le immagini della macchina virtuale e per realizzare tutte le altre memorizzazioni di cui il guest ha bisogno si utilizzano generalmente nfs, cifs o iscsi. Gli accessi a questi storage possono continuare in modo semplice se le connessioni di rete non vengono interrotte durante la migrazione del guest.

La migrazione in tempo reale rende possibili modalità interamente nuove nella gestione dei data center. Per esempio, gli strumenti di gestione della virtualizzazione sono in grado di monitorare tutti i vmm presenti in un ambiente e bilanciare automaticamente l'uso delle risorse spostando i guest tra i vmm. È inoltre possibile ottimizzare il consumo di energia elettrica e la gestione del raffreddamento migrando tutti i guest da alcuni server selezionati su altri server in grado di gestirne il carico, arrestando poi completamente i primi server. Se il carico aumenta, questi strumenti di gestione sono in grado di riaccendere i server e migrare di nuovo gli ospiti.

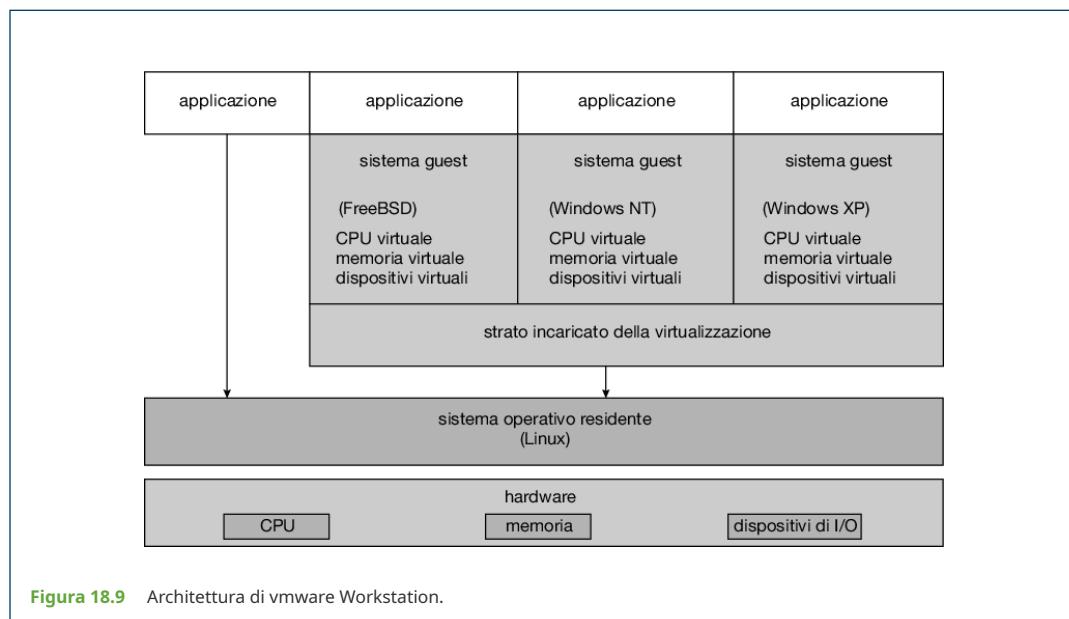
18.7 Esempi

Nonostante i vantaggi offerti dalle macchine virtuali, dopo il loro primo sviluppo esse sono state oggetto di poche attenzioni per diversi anni. Ciononostante, oggi le macchine virtuali sono diventate di moda come strumento per risolvere problemi di compatibilità di sistemi. In questo paragrafo analizziamo due popolari macchine virtuali: vmware workstation e la Java virtual machine. Queste macchine virtuali possono generalmente essere eseguite su sistemi operativi di qualsiasi tipologia descritta nei capitoli precedenti.

18.7.1 VMware

vmware Workstation è una nota applicazione commerciale che astrae hardware Intel x86 e compatibile in macchine virtuali distinte. vmware è un ottimo esempio di hypervisor di tipo 2 e funge da applicazione in un sistema operativo host, quale Windows o Linux, che può così eseguire diversi sistemi operativi guest contemporaneamente come macchine virtuali indipendenti.

Si può osservare l'architettura di un sistema siffatto nella Figura 18.9. In questo esempio, Linux è il sistema operativo host, mentre FreeBSD, Windows nt e Windows xp sono i sistemi guest. Lo strato incaricato della virtualizzazione è il fulcro di vmware, poiché grazie a esso l'hardware viene astratto in macchine virtuali a sé stanti che eseguono sistemi operativi guest. Ciascuna macchina virtuale, oltre a possedere una cpu virtuale, può contare su elementi virtuali propri quali la memoria, i dischi, le interfacce di rete e via di seguito.



Il disco fisico di cui l'ospite dispone è in realtà semplicemente un file all'interno del file system del sistema operativo host. Per creare un'istanza identica al guest è sufficiente copiare il file. Copiare il file in una nuova posizione protegge l'istanza dell'ospite da possibili danneggiamenti alla posizione originale. Spostando il file in una nuova posizione viene spostato il sistema guest. Questi scenari mostrano come la virtualizzazione può effettivamente aumentare l'efficienza nell'amministrazione di un sistema e nell'uso delle risorse di sistema.

18.7.2 Java virtual machine

Il linguaggio di programmazione Java, introdotto dalla Sun Microsystems alla fine del 1995, è un linguaggio orientato agli oggetti molto diffuso. Java fornisce, oltre alla specifica del linguaggio e a una vasta libreria api, anche la definizione della macchina virtuale Java (*Java virtual machine*, jvm). Java è dunque un esempio della virtualizzazione dell'ambiente di programmazione trattata nel Paragrafo 18.5.6.

Gli oggetti si specificano con il costrutto `class` e un programma consiste di una o più classi. Per ognuna di queste, il compilatore produce un file (`.class`) contenente il cosiddetto bytecode; si tratta di codice nel linguaggio macchina della jvm, indipendente dall'architettura sottostante, che viene per l'appunto eseguito dalla jvm.

La jvm è un calcolatore astratto che consiste di un caricatore delle classi e di un interprete del linguaggio che esegue il bytecode (Figura 18.10). Il caricatore delle classi carica i file `.class`, sia del programma scritto in Java sia dalla libreria api, affinché l'interprete possa eseguirli. Dopo che una classe è stata caricata, il verificatore delle classi controlla la correttezza sintattica del bytecode, che il

codice non produca accessi oltre i limiti dello stack e che non esegua operazioni aritmetiche sui puntatori, che potrebbero generare accessi illegali alla memoria. Se il controllo ha un esito positivo, la classe viene eseguita dall'interprete. La jvm gestisce la memoria in modo automatico procedendo alla sua “ripulitura” (garbage collection) che consiste nel recupero delle aree della memoria assegnate a oggetti non più in uso per restituirla al sistema. Al fine di incrementare le prestazioni dei programmi eseguiti dalla macchina virtuale una notevole attività di ricerca è focalizzata allo studio degli algoritmi di ripulitura della memoria.

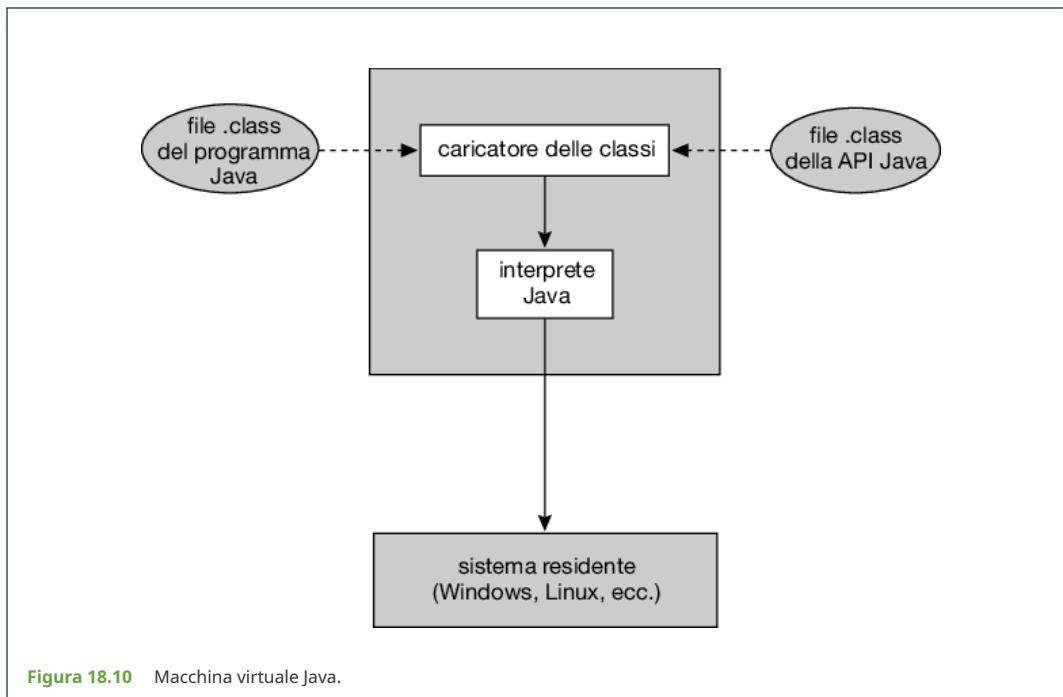


Figura 18.10 Macchina virtuale Java.

La jvm può essere implementata come software ospitato da un sistema operativo residente, per esempio Windows, Linux o macos, oppure all'interno di un browser web. In alternativa, può essere cablata in un circuito integrato espressamente progettato per l'esecuzione di programmi Java. Nel primo caso, l'interprete Java interpreta le istruzioni bytecode una alla volta. Una soluzione più efficiente consiste nell'uso di un compilatore istantaneo o just-in-time (jit). Alla prima invocazione di un metodo Java, il bytecode relativo è tradotto in linguaggio macchina comprensibile dalla macchina fisica ospitante. Il codice macchina relativo è poi salvato in una cache, in modo da essere direttamente riutilizzabile a una successiva invocazione del metodo Java, evitando di ripetere la lenta interpretazione del bytecode. Una soluzione potenzialmente ancora più veloce è implementare la jvm in un circuito integrato che esegua le istruzioni bytecode come codice macchina nativo, eliminando del tutto la necessità di interpreti e compilatori.

18.8 La ricerca sulla virtualizzazione

Come accennato in precedenza, la virtualizzazione ha goduto di una crescente popolarità negli ultimi anni come mezzo per risolvere i problemi di compatibilità tra sistemi. La ricerca si è estesa per coprire molti altri usi della virtualizzazione, tra cui il supporto ai microservizi in esecuzione su sistemi operativi *library* e il partizionamento sicuro delle risorse nei sistemi embedded. In questi ambiti sono in corso molte ricerche interessanti.

Spesso, nel contesto del cloud computing, la stessa applicazione viene eseguita su migliaia di sistemi. Per essere gestite al meglio, queste installazioni possono essere virtualizzate. Considerate lo stack di esecuzione quando un'applicazione si trova su un sistema operativo general-purpose ricco di servizi all'interno di una macchina virtuale gestita da un hypervisor: progetti come gli unikernel, costruiti su sistemi operativi *library*, mirano a migliorare l'efficienza e la sicurezza in questi ambienti. Gli unikernel sono immagini specializzate che utilizzano un singolo spazio di indirizzi e riducono la superficie di attacco e l'occupazione di risorse delle applicazioni. In sostanza, compilano l'applicazione insieme alle librerie di sistema di cui essa ha bisogno e ai servizi del kernel che utilizza in un singolo binario eseguito all'interno di un ambiente virtuale (o anche su macchina fisica). Le ricerche sull'interazione tra kernel, hardware e applicazioni non sono una novità (si veda, per esempio, <https://pdos.csail.mit.edu/6.828/2005/readings/engler95exokernel.pdf>), ma il cloud computing e la virtualizzazione hanno creato un rinnovato interesse per l'area. Si veda <http://unikernel.org> per maggiori dettagli.

Le istruzioni di virtualizzazione implementate nelle moderne cpu hanno dato origine a una nuova branca della ricerca che non si concentra più su un uso efficiente dell'hardware, ma piuttosto su un miglior controllo dei processi. Gli hypervisor di partizionamento suddividono le risorse fisiche della macchina esistente tra i guest, impegnando quindi completamente le risorse della macchina senza andare oltre le loro possibilità, e possono estendere in modo sicuro le funzionalità di un sistema operativo esistente tramite le funzionalità di un altro sistema operativo (eseguito in un dominio di macchina virtuale distinto), in esecuzione su un sottoinsieme di risorse fisiche della macchina. Questo evita l'onere di scrivere un intero sistema operativo da zero. Per esempio, un sistema Linux che non dispone di funzionalità per le attività critiche di sicurezza in tempo reale può essere esteso con un sistema operativo real-time leggero eseguito nella propria macchina virtuale. Gli hypervisor tradizionali generano un overhead superiore rispetto all'esecuzione di task nativi ed è quindi necessario un nuovo tipo di hypervisor.

Ogni task viene eseguito all'interno di una macchina virtuale, ma l'hypervisor si occupa solo di inizializzare il sistema e avviare i task e non è coinvolto nel procedere delle operazioni. Ogni macchina virtuale dispone del proprio hardware ed è libera di gestire l'hardware a essa allocato senza interferenze da parte dell'hypervisor. Poiché l'hypervisor non interrompe le operazioni dei task e non viene richiamato da questi, i task possono assumere l'aspetto di attività in tempo reale e possono essere molto più sicuri.

Fanno parte della famiglia di hypervisor di partizionamento i progetti Quest-V, evm, Xtratum e Siemens Jailhouse. Questi hypervisor di separazione, o *separation hypervisor* (si veda <http://www.csl.sri.com/users/rushby/papers/sosp81.pdf>), utilizzano la virtualizzazione per partizionare i componenti di sistema separati in un sistema distribuito a livello di chip. Vengono quindi implementati i canali di memoria condivisa sicuri, utilizzando le tabelle delle pagine estese in hardware, in modo che guest indipendenti in modalità sandbox possano comunicare tra loro. Gli obiettivi di questi progetti sono aree come la robotica, le auto a guida autonoma e l'Internet delle cose (IoT). Si veda <https://www.cs.bu.edu/richwest/papers/west-tocs16.pdf> per maggiori dettagli.

18.9 Sommario

- La virtualizzazione è un metodo per fornire a un guest un duplicato dell'hardware di un sistema. Diversi guest possono essere in funzione su un dato sistema e ciascuno crede di essere il sistema operativo nativo, con pieno controllo sul sistema.
- La virtualizzazione è nata come un metodo per consentire a ibm di isolare gli utenti e fornire loro i propri ambienti di esecuzione su mainframe ibm. Da allora, grazie al miglioramento dei sistemi e delle prestazioni delle cpu e attraverso tecniche software innovative, la virtualizzazione è diventata una caratteristica diffusa nei data center e anche sui personal computer. In seguito alla diffusione della virtualizzazione, i progettisti di cpu hanno aggiunto funzionalità per supportarla. Questo effetto "palla di neve" è destinato a continuare e farà crescere l'utilizzo della virtualizzazione e il supporto hardware alla stessa.
- Il gestore di macchine virtuali, o hypervisor, crea ed esegue le macchine virtuali. Gli hypervisor di tipo 0 sono implementati nell'hardware e richiedono modifiche al sistema operativo per garantire un corretto funzionamento. Alcuni hypervisor di tipo 0 offrono un esempio di paravirtualizzazione, in cui il sistema operativo è consapevole della virtualizzazione e ne assiste l'esecuzione.
- Gli hypervisor di tipo 1 forniscono l'ambiente e le caratteristiche necessarie per creare, eseguire e distruggere le macchine virtuali guest. Ogni guest include tutto il software tipicamente associato a un sistema nativo completo, compresi il sistema operativo, i driver dei dispositivi, le applicazioni, gli account utente, e così via.
- Gli hypervisor di tipo 2 sono semplicemente applicazioni che girano su altri sistemi operativi ignari del fatto che sta avvenendo una virtualizzazione. Questi hypervisor non godono del supporto dell'hardware o di un host e devono quindi eseguire tutte le attività di virtualizzazione nell'ambito di un processo.
- La virtualizzazione dell'ambiente di programmazione fa parte del progetto di un linguaggio di programmazione. Il linguaggio specifica un'applicazione contenitore in cui eseguire i programmi e questa applicazione fornisce servizi ai programmi.
- L'emulazione viene utilizzata quando un sistema host ha un'architettura diversa da quella in cui è stato compilato il sistema guest. Ogni istruzione che il guest intende eseguire deve essere tradotta dal suo insieme di istruzioni nell'insieme di istruzioni dell'hardware nativo. Anche se questo metodo porta a un calo delle prestazioni, la penalizzazione è bilanciata dalla possibilità di eseguire vecchi programmi su un nuovo hardware incompatibile o di far girare videogiochi progettati per le vecchie console su un hardware moderno.
- Implementare la virtualizzazione è impegnativo, soprattutto quando il supporto hardware è scarso. Per la virtualizzazione è richiesto un minimo supporto hardware, ma in presenza di un numero maggiore di funzionalità offerte dal sistema la virtualizzazione diventa più facile da implementare e le prestazioni dei guest migliorano.
- I vmm sfruttano qualsiasi supporto hardware a loro disposizione per ottimizzare lo scheduling della cpu, la gestione della memoria e i moduli di i/o, al fine di fornire ai guest un uso ottimale delle risorse, proteggendo contemporaneamente il vmm dai guest e i guest l'uno dall'altro .
- La ricerca attuale sta estendendo gli usi della virtualizzazione. Gli unikernel mirano ad aumentare l'efficienza e ridurre la superficie degli attacchi di sicurezza compilando un'applicazione, le sue librerie e le risorse del kernel di cui l'applicazione ha bisogno in un unico binario con un singolo spazio di indirizzi eseguito all'interno di una macchina virtuale. Gli hypervisor di partizionamento garantiscono un'esecuzione sicura, operazioni in tempo reale e altre funzionalità tradizionalmente disponibili solo per applicazioni eseguite su hardware dedicato.

Esercizi

18.1 Descrivete i tre tipi tradizionali di virtualizzazione.

18.2 Descrivete i quattro ambienti di esecuzione che somigliano ad ambienti virtuali, ma non costituiscono una vera e propria virtualizzazione. Specificate come e perché non si tratta di una “vera” virtualizzazione.

18.3 Descrivete quattro vantaggi della virtualizzazione.

18.4 Perché su alcune cpu un vmm non può implementare una virtualizzazione basata su trap-and-emulate? Senza la possibilità di utilizzare trap-and-emulate, quale metodo può essere utilizzato per implementare la virtualizzazione?

18.5 Quale assistenza hardware per la virtualizzazione può essere fornita dalle moderne cpu?

18.6 Perché la migrazione in tempo reale è possibile in ambienti virtuali, ma lo è molto meno per un sistema operativo nativo?

CAPITOLO 19

Reti e sistemi distribuiti

Aggiornamento a cura di Sarah Diesburg

Un sistema distribuito è un insieme di processori che non condividono la memoria o un clock. Ogni nodo ha la propria memoria locale e la comunicazione avviene tramite diversi tipi di reti, come bus ad alta velocità. I sistemi distribuiti sono ora più importanti che mai e tutti noi abbiamo quasi certamente utilizzato almeno qualche tipo di servizio distribuito. Le applicazioni dei sistemi distribuiti spaziano dall'accesso trasparente ai file all'interno di un'azienda, ai servizi di archiviazione di file e foto su cloud, all'analisi dei trend di mercato su grandi insiemi di dati, all'elaborazione parallela di dati scientifici e altro ancora. L'esempio fondamentale di un sistema distribuito è qualcosa con cui tutti abbiamo certamente familiarità: Internet.

In questo capitolo si tratta la struttura generale dei sistemi distribuiti e delle reti che li interconnettono e si evidenziano le principali differenze nelle tipologie e nei ruoli degli attuali progetti di sistemi distribuiti. Infine, si esaminano alcuni aspetti di base della progettazione dei file system distribuiti.

19.1 Vantaggi dei sistemi distribuiti

Un sistema distribuito è un insieme di nodi lasciamente connessi tramite una rete di comunicazione. Ogni nodo di un sistema distribuito considera remoti gli altri nodi del sistema e le rispettive risorse, mentre considera locali le proprie.

I nodi presenti in un sistema distribuito possono variare per dimensioni e funzioni; possono comprendere piccole unità d'elaborazione, personal computer e grandi calcolatori d'uso generale. I nodi sono chiamati in molti modi: processori, *siti*, *macchine* o *host*, secondo il contesto in cui sono citati. In questo testo si usa principalmente il termine *sito* per indicare una locazione fisica delle macchine, e *nodo* per riferirsi a uno specifico sistema in un sito. I nodi possono essere collocati in una configurazione client-server, in una configurazione peer-to-peer o in una configurazione ibrida. Nella configurazione client-server classica, un nodo su un sito, il server, ha una risorsa che un altro nodo, il client (o l'utente), vorrebbe utilizzare. Una struttura generale di un sistema distribuito client-server è mostrata nella Figura 19.1. In una configurazione peer-to-peer non ci sono server né client, ma i nodi condividono le stesse responsabilità e possono agire sia come client sia come server.

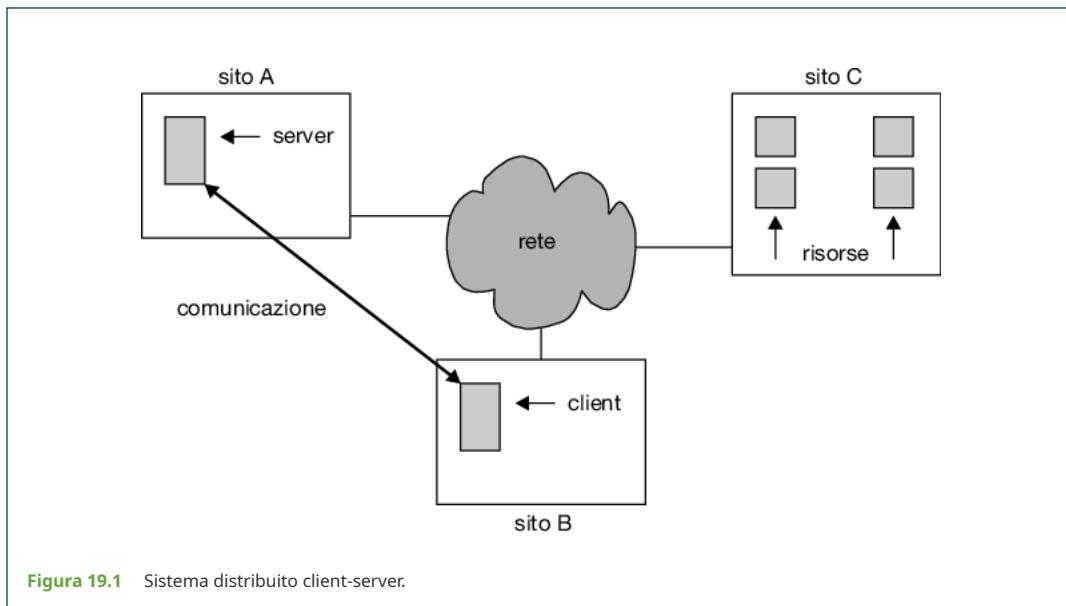


Figura 19.1 Sistema distribuito client-server.

Quando diversi siti sono collegati tra loro da una rete di comunicazione, gli utenti dei vari siti hanno l'opportunità di scambiarsi informazioni. A basso livello, lo scambio di **messaggi** tra più sistemi avviene all'incirca come lo scambio di messaggi tra i processi di un singolo computer, come discusso nel Paragrafo 3.4. Una volta fornito lo scambio di messaggi, tutte le funzionalità di livello superiore presenti nei sistemi standalone possono essere estese a un sistema distribuito. Tali funzioni comprendono la memorizzazione di file, l'esecuzione di applicazioni e le chiamate di procedure remote (rpc).

I tre motivi principali per costruire sistemi distribuiti sono: *condivisione delle risorse*, *velocizzazione delle elaborazioni*, *affidabilità*. Li discuteremo nel seguito di questo capitolo.

19.1.1 Condivisione delle risorse

Se siti diversi, con risorse diverse, sono collegati tra loro, l'utente di un sito può avere la possibilità di usare le risorse disponibili in un altro sito. Per esempio, un utente del sito A può usare una stampante laser presente nel sito B. Nel frattempo, un utente del sito B può accedere a un file che risiede nel sito A. In generale, la condivisione delle risorse di un sistema distribuito offre meccanismi per la condivisione dei file in siti remoti, per l'elaborazione delle informazioni in una base di dati distribuita, per la stampa dei file in siti remoti, per l'uso remoto di hardware specializzato (come un supercomputer o una gpu), e per eseguire altre operazioni.

19.1.2 Velocizzazione delle elaborazioni

Se una particolare computazione si può dividere in componenti eseguibili in modo concorrente, un sistema distribuito permette di ripartire queste attività elaborate tra i diversi siti; i componenti si possono eseguire in modo concorrente determinando una velocizzazione dell'elaborazione. Ciò è particolarmente rilevante quando si esegue un'elaborazione su larga scala di grandi dataset (per esempio, nel trattamento di grandi quantità di dati per analizzare le tendenze di mercato). Inoltre, se un particolare sito è attualmente sovraccarico, alcuni job si possono spostare in altri siti con carico inferiore. Lo spostamento dei job si chiama bilanciamento del carico (*load balancing*) ed è comune nei nodi di un sistema distribuito e in altri servizi disponibili su Internet.

19.1.3 Affidabilità

Se cade un sito di un sistema distribuito, i siti restanti possono continuare a funzionare, offrendo al sistema una maggiore affidabilità. Se il sistema distribuito è formato da più grandi sistemi elaborativi autonomi, vale a dire calcolatori d'uso generale, il guasto di uno di loro non influisce sugli altri sistemi elaborativi. Se invece il sistema è formato da piccoli calcolatori, ciascuno dei quali è responsabile di una funzione cruciale per il sistema (come il server web oppure il file system) un singolo guasto può fermare il funzionamento di tutto il sistema. In generale, con un opportuno livello di ridondanza (sia dei dispositivi sia dei dati) il sistema può continuare a funzionare anche se qualcuno dei siti si guasta.

Il guasto di un sito deve essere rilevato dal sistema, e per superare tale situazione si devono intraprendere azioni adeguate. Il sistema non deve più utilizzare i servizi di quel sito; inoltre, se la funzione del sito guasto può essere rilevata da un altro sito, il sistema deve assicurare che il trasferimento della funzione avvenga correttamente. Infine, quando il sito guasto viene ripristinato o riparato devono essere disponibili meccanismi che lo reintegrino facilmente nel sistema.

19.2 Struttura delle reti

Per comprendere interamente i ruoli e i tipi di sistemi distribuiti in uso oggi è necessario conoscere le reti che li collegano. Questo paragrafo introduce i concetti di base delle reti e le sfide da affrontare in relazione ai sistemi distribuiti. Il resto del capitolo tratta in modo specifico i sistemi distribuiti.

I due tipi di rete fondamentali sono le reti locali e le reti geografiche. La principale differenza tra le due consiste nella loro distribuzione geografica. Le reti locali (*local-area network, lan*) comprendono un certo numero di host distribuiti su piccole aree, come un unico edificio o alcuni edifici adiacenti. Le reti geografiche (*wide-area network, wan*), invece, comprendono sistemi distribuiti su vaste aree geografiche (per esempio, gli Stati Uniti). Queste differenze implicano notevoli variazioni nella velocità e nell'affidabilità delle reti di comunicazione, e si riflettono nella progettazione dei sistemi operativi distribuiti.

19.2.1 Reti locali

Le reti locali (lan) sono nate nei primi anni '70 con lo scopo di sostituire i sistemi basati su mainframe. Molte imprese trovano più economico avere tanti piccoli calcolatori, ciascuno con le proprie applicazioni autonome, anziché un unico sistema di grandi dimensioni. Poiché è probabile che ogni piccolo calcolatore necessiti di una serie completa di dispositivi periferici, come dischi e stampanti, e poiché è probabile che esista qualche forma di condivisione di dati in un'unica società, fu naturale collegare questi piccoli sistemi in una rete.

Come si è detto, generalmente le lan sono progettate per coprire piccole aree, e di solito sono usate in ambienti di uffici. In questi sistemi tutti i siti sono vicini fra loro, e di conseguenza i collegamenti tendono ad avere una maggior velocità e una minor frequenza di errori rispetto alle reti geografiche.

Una lan tipica è composta da diversi calcolatori, dai mainframe ai laptop e altri dispositivi portatili, vari dispositivi periferici condivisi (come stampanti laser o array di storage), e uno o più router (unità d'elaborazione specializzate per le reti) che danno accesso ad altre reti (Figura 19.2). Per costruire le lan comunemente si usano le tecnologie Ethernet e WiFi. I punti di accesso wireless collegano i dispositivi alla lan in modalità wireless e possono anche, talvolta, fungere da router.

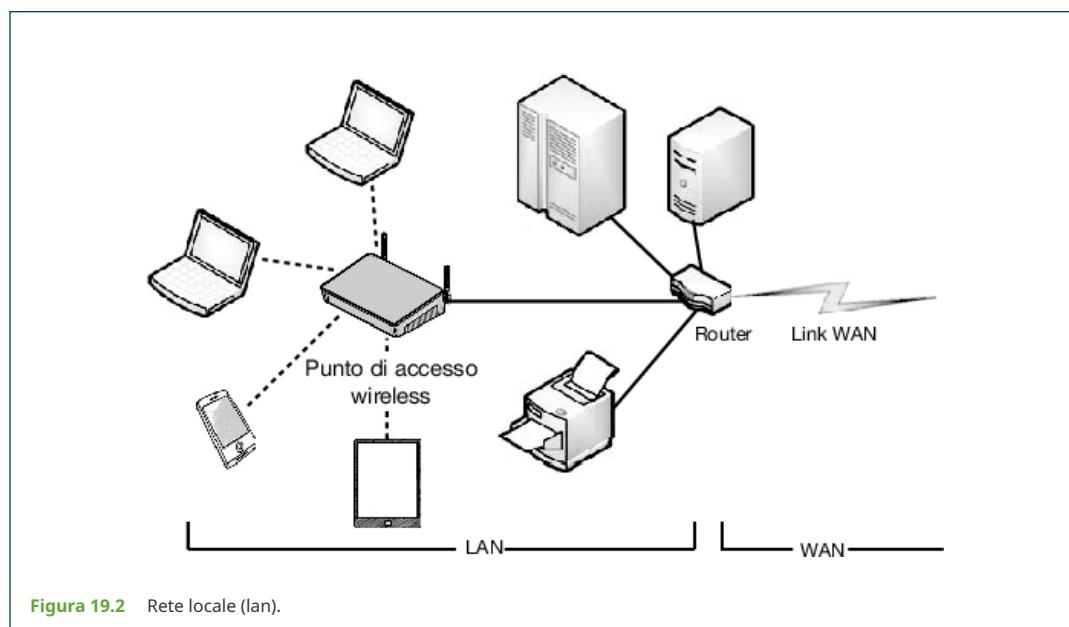


Figura 19.2 Rete locale (lan).

Le reti Ethernet si trovano generalmente in aziende e organizzazioni in cui computer e periferiche tendono a essere dispositivi fissi. Per inviare segnali queste reti utilizzano *cavi coassiali, doppini ("twisted pair") e/o fibra ottica*. Una rete Ethernet non ha un controller centrale, perché è costituita da un bus multiaccesso, quindi è possibile aggiungere facilmente nuovi host alla rete. Il protocollo Ethernet è definito dallo standard ieee 802.3. Le tipiche velocità di una rete Ethernet che utilizza il comune doppino possono variare da 10 Mbps a oltre 10 Gbps, mentre con altri tipi di cavi si raggiungono velocità di 100 Gbps.

Il WiFi è ormai onnipresente e viene utilizzato da solo o per integrare le reti Ethernet tradizionali. Nello specifico, il WiFi ci consente di costruire una rete senza utilizzare cavi fisici. Ogni host ha un trasmettitore e un ricevitore wireless che utilizza per partecipare alla rete. Il WiFi è definito dallo standard ieee 802.11. Le reti wireless sono diffuse nelle case e nelle aziende, così come in aree pubbliche.

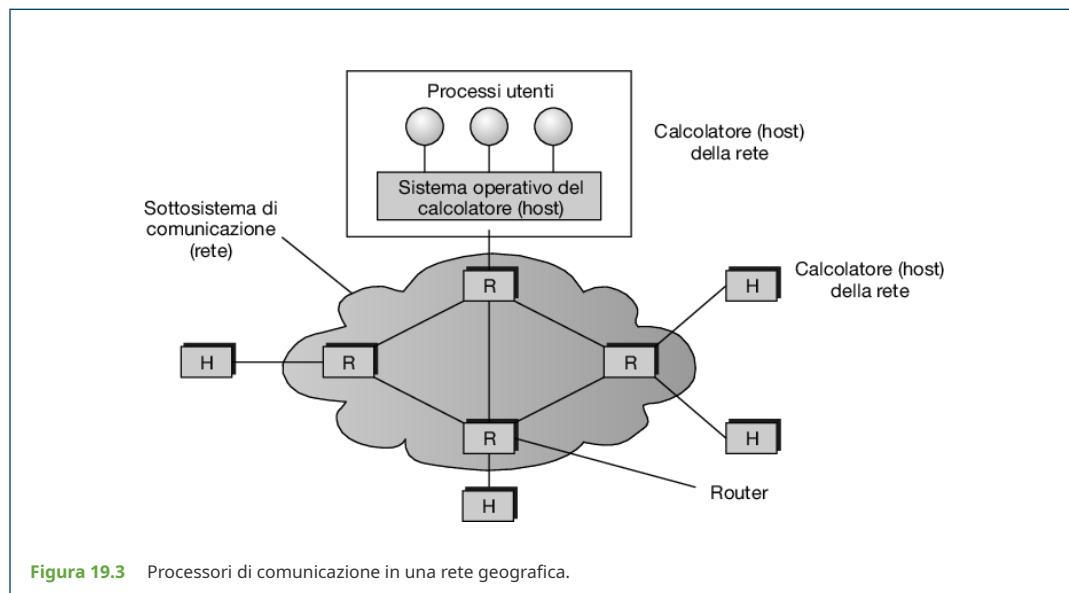
come biblioteche, Internet café, arene sportive e persino autobus e aeroplani. Le velocità di una rete WiFi possono variare da 11 Mbps a oltre 400 Mbps.

Entrambi gli standard ieee 802.3 e 802.11 sono in continua evoluzione. Per avere informazioni aggiornate sui vari standard e sulle loro velocità si vedano i riferimenti bibliografici forniti alla fine del capitolo.

19.2.2 Reti geografiche

Le reti geografiche (wan) sono nate alla fine degli anni '60 come progetto di ricerca accademica per fornire un sistema di comunicazione efficiente tra siti, che permettesse a un'ampia comunità di utenti di condividere in modo conveniente ed economico le risorse elaborative. La prima wan progettata e sviluppata è stata l'*Arpanet*, nata nel 1968. Partita come rete sperimentale a quattro siti, è cresciuta fino a diventare una "rete di reti" mondiale, l'*Internet* (noto anche come World Wide Web), che comprende, fra l'altro, milioni di sistemi elaborativi.

I siti di una wan sono distribuiti fisicamente su una vasta area geografica. I collegamenti tipici sono linee telefoniche, linee dedicate, cavi ottici, collegamenti a microonde, onde radio e canali satellitari. Questi canali di comunicazione sono controllati dai router (Figura 19.3), i responsabili dell'instradamento del traffico verso altri router e altre reti e del trasferimento di informazioni tra i vari siti.



Le aree residenziali possono connettersi a Internet via telefono, via cavo o tramite un Internet service provider specializzato che installa dei router per collegare le residenze ai servizi centrali. Ci sono ovviamente altre wan oltre a Internet. Una società può creare la sua wan privata per aumentare la sicurezza, le prestazioni o l'affidabilità.

Come accennato, le wan sono generalmente più lente delle reti lan, anche se le connessioni backbone delle wan che collegano le principali città possono avere velocità di trasferimento molto alte grazie ai cavi in fibra ottica. Molti fornitori di backbone, infatti, offrono connessioni da 40 o 100 Gbps in fibra ottica. (Generalmente i collegamenti più lenti sono quelli che vanno dai **provider di servizi Internet**, gli **isp**, verso le case e le aziende). Tuttavia, i collegamenti wan vengono costantemente aggiornati a tecnologie più veloci, poiché la richiesta di maggiore velocità continua a crescere.

Spesso le wan e le lan sono interconnesse ed è difficile dire dove finisce l'una e comincia l'altra. Si consideri la rete dati cellulare. I telefoni cellulari sono utilizzati sia per comunicazioni vocali sia per connessioni dati e, in una determinata zona, si collegano tramite onde radio a una cella che contiene ricevitori e trasmettitori. Questa parte della rete è simile a una lan tranne per il fatto che i telefoni cellulari non comunicano tra loro, a meno che due persone in comunicazione tra loro siano collegate alla stessa cella. Più spesso, le celle sono collegate ad altre celle e a hub che collegano le celle di comunicazione alle linee terrestri o ad altri mezzi e instradano i pacchetti verso le loro destinazioni. Questa parte della rete è più simile a una wan. Una volta che la cella appropriata riceve i pacchetti, si utilizzano i trasmettitori per inviarli al destinatario corretto.

19.3 Struttura della comunicazione

Dopo aver trattato gli aspetti fisici della comunicazione tramite reti si può considerarne il funzionamento interno.

19.3.1 Naming e risoluzione dei nomi

Il primo problema nelle comunicazioni tramite rete è il modo di indicare i nomi (*naming*) dei sistemi nella rete. Affinché due processi in esecuzione in due siti differenti *A* e *B* possano scambiarsi informazioni devono poter fare riferimento l'uno all'altro. All'interno di un calcolatore ciascun processo ha un identificatore di processo, quindi i messaggi si possono indirizzare con l'identificatore del processo. Poiché i sistemi presenti nella rete non condividono memoria, inizialmente un host non ha alcuna informazione riguardante processi in esecuzione su altri sistemi.

Per risolvere questo problema i processi di un sistema remoto sono generalmente identificati dalla coppia *<nome del calcolatore, identificatore>*, dove *nome del calcolatore* (host) è di solito un nome unico all'interno della rete, e *identificatore* può essere un identificatore di processo o un altro numero unico all'interno del calcolatore. Un *nome del calcolatore* è di solito un identificatore alfanumerico, anziché un numero, per renderne più semplice l'utilizzo da parte degli utenti. Per esempio, il sito *A* potrebbe contenere calcolatori con i nomi *program, student, faculty* e *cs*; è sicuramente più facile ricordare il nome *program* rispetto all'indirizzo numerico *128.148.31.100*.

Sebbene i nomi siano comodi per gli esseri umani, nei calcolatori è preferibile usare i numeri, sia per motivi di velocità sia di semplicità. Per questa ragione ci deve essere un meccanismo per la risoluzione del nome del calcolatore nel corrispondente identificatore, che descrive il sistema di destinazione ai dispositivi della rete. Questo meccanismo di risoluzione è simile all'associazione (*binding*) tra nomi e indirizzi definita durante compilazione, linking, caricamento ed esecuzione di un programma (Capitolo 9). Nel caso dei nomi di calcolatori esistono due possibilità; nella prima ciascun calcolatore può avere un file contenente i nomi e gli indirizzi di tutti i calcolatori raggiungibili tramite la rete (situazione simile al binding nella fase di compilazione). Il problema di questo modello è che l'aggiunta o la rimozione di un calcolatore dalla rete richiede l'aggiornamento in tutti i calcolatori dei suddetti file. In effetti, agli inizi di arpanet c'era un file host canonico che veniva copiato periodicamente su ogni sistema. Con il crescere della rete, tuttavia, questo metodo è diventato insostenibile.

La seconda possibilità consiste nel distribuire le informazioni tra i sistemi connessi in rete. La rete deve quindi adoperare un protocollo per distribuire e recuperare queste informazioni. Questo schema corrisponde all'associazione tra nomi e indirizzi nella fase d'esecuzione. Internet usa il dns (*domain name system*) per la risoluzione dei nomi degli host.

Il dns specifica la struttura di naming dei calcolatori e la risoluzione dei nomi in indirizzi. I calcolatori della rete Internet si indirizzano logicamente con un nome composto da più parti, detto nome di dominio. Le parti di un nome di dominio progrediscono dalla parte più specifica a quella più generale dell'indirizzo, con i diversi campi separati da un punto: *eric.cs.yale.edu* si riferisce al calcolatore *eric* del *Department of Computer Science* della Yale University nel dominio di primo livello *edu*. (Altri domini del primo livello comprendono *com*, per usi commerciali, e *org*, per le organizzazioni, oltre ai domini indicanti i Paesi connessi alla rete, per i sistemi identificati a seconda del Paese di appartenenza e non dalle loro finalità.). In generale, il sistema risolve l'indirizzo esaminando gli elementi del nome del calcolatore nell'ordine inverso. Per ciascun elemento c'è un server dei nomi – semplicemente un processo in un sistema – che accetta un nome e risponde con l'indirizzo del server dei nomi responsabile della risoluzione di quel nome. Come passo finale si interroga il server dei nomi associato al calcolatore in questione, che provvederà a riportare l'indirizzo del calcolatore. Per esempio, una richiesta di connessione a *eric.cs.yale.edu* effettuata da un processo sul sistema *A* vedrà eseguiti i seguenti passi.

1. La libreria di sistema o il kernel del sistema *A* inoltra una richiesta al server dei nomi del dominio *edu*, richiedendo l'indirizzo del server dei nomi di *yale.edu*. Per poter essere interrogato, il server dei nomi di *edu* deve avere un indirizzo noto a priori.
2. Il server dei nomi di *edu* riporta l'indirizzo del calcolatore in cui risiede il server dei nomi di *yale.edu*.
3. Il sistema *A* interroga il server dei nomi situato a questo indirizzo riguardo al dominio *cs.yale.edu*.
4. Si ottiene in risposta un indirizzo. Si invia una richiesta a tale indirizzo per risolvere *eric.cs.yale.edu* e si ottiene finalmente come risposta l'identificatore del calcolatore sotto forma di indirizzo Internet di quel sistema (per esempio, *128.148.31.100*).

Questo protocollo può sembrare inefficiente, ma ciascun server dei nomi mantiene cache locali per accelerare il processo, memorizzando gli indirizzi ip che sono già stati risolti. Naturalmente il contenuto di queste cache deve essere aggiornato col passare del tempo, poiché sia il server dei nomi sia il suo indirizzo potrebbero essere cambiati. In pratica questo servizio è così importante da indurre l'introduzione di numerose ottimizzazioni e molte protezioni nel protocollo. Si consideri che cosa accadrebbe nel caso di una caduta del server dei nomi primario *edu*: si potrebbe perdere la capacità di risolvere gli indirizzi dei calcolatori del dominio *edu*, che diventerebbero tutti irraggiungibili. La soluzione consiste nell'impiego di server dei nomi di backup, nei quali duplicare il contenuto dei server dei nomi primari.

Prima dell'introduzione dei server dei nomi, tutti i calcolatori connessi alla rete Internet avevano bisogno di copie di un file contenente i nomi e gli indirizzi di tutti i calcolatori della rete. Tutte le modifiche a questo file dovevano essere registrate presso un particolare sito, il sistema sri-nic, e periodicamente tutti i calcolatori dovevano prelevarvi la versione aggiornata di questo file per poter contattare nuovi sistemi o trovare i calcolatori che avevano cambiato indirizzo. Con il dns, il server dei nomi di ciascun sito è responsabile dell'aggiornamento delle informazioni riguardanti il proprio dominio. Per esempio, il server dei nomi di *yale.edu* è responsabile di tutte le modifiche riguardanti i calcolatori della Yale University, e tali modifiche non devono essere notificate ad alcuno. Le ricerche del dns recupereranno automaticamente le informazioni aggiornate, poiché contatteranno direttamente *yale.edu*. All'interno del dominio possono esistere sottodomini autonomi allo scopo di distribuire ulteriormente la responsabilità circa le variazioni dei nomi e degli identificatori dei calcolatori.

Java mette a disposizione la api necessaria per progettare programmi che associno nomi di dominio a indirizzi ip. Il programma della Figura 19.4 accetta in ingresso dalla riga di comando un nome di dominio (per esempio, *eric.cs.yale.edu*) e fornisce in uscita l'indirizzo ip della macchina, o un messaggio che indichi l'impossibilità di tradurre il nome. La classe Java *InetAddress* rappresenta un nome o un indirizzo ip; il suo metodo statico *getByName()* accetta come parametro un nome di dominio sotto forma di stringa, e

restituiscce il corrispondente `InetAddress`. Il programma invoca quindi il metodo `getHostAddress()`, la cui implementazione usa dns per cercare l'indirizzo della macchina indicata.

```
/**  
 * Uso: java DNSLookUp <nome IP>  
 * Esempio: java DNSLookup www.wiley.com  
 */  
  
public class DNSLookUp {  
  
    public static void main(String[] args) {  
  
        InetAddress hostAddress;  
  
        try {  
  
            hostAddress = InetAddress.getByName(args[0]);  
  
            System.out.println(hostAddress.getHostAddress());  
  
        }  
  
        catch (UnknownHostException uhe) {  
  
            System.err.println("Unknown host: " + args[0]);  
  
        }  
  
    }  
  
}
```

Figura 19.4 Programma Java che illustra l'uso di dns.

In generale, il sistema operativo è responsabile di accettare dai propri processi messaggi destinati a *<nome del calcolatore, identificatore>* e di trasferire il messaggio al calcolatore appropriato. Il kernel del calcolatore di destinazione è quindi responsabile del trasferimento del messaggio al processo specificato dall'identificatore. Questo scambio di informazioni è descritto nel Paragrafo 19.3.4.

19.3.2 Protocolli di comunicazione

Nel progettare una rete di comunicazione, è necessario considerare la complessità di coordinare operazioni asincrone che comunicano in un ambiente potenzialmente lento e soggetto a errori. Inoltre i sistemi nella rete devono accordarsi su un protocollo o su un insieme di protocolli per determinare i nomi dei calcolatori, individuare i calcolatori nella rete, stabilire le connessioni, e così via. Il problema della progettazione e della relativa realizzazione si può semplificare mediante una suddivisione in strati. Ciascuno strato in un sistema comunica con lo strato corrispondente negli altri sistemi. In genere ogni strato ha i propri protocolli e la comunicazione fra strati corrispondenti segue uno specifico protocollo. I protocolli si possono realizzare in hardware o possono essere programmi. La Figura 19.5 mostra, per esempio, uno schema delle comunicazioni logiche tra due calcolatori, con i tre strati di livello più basso realizzati in hardware. L'International Standards Organization ha creato il modello osi (Open System Interconnection) per descrivere gli strati di una rete di comunicazione. Anche se questi strati non sono implementati nella pratica, sono utili per capire come lavora la rete a livello logico. Tali strati sono definiti come segue.

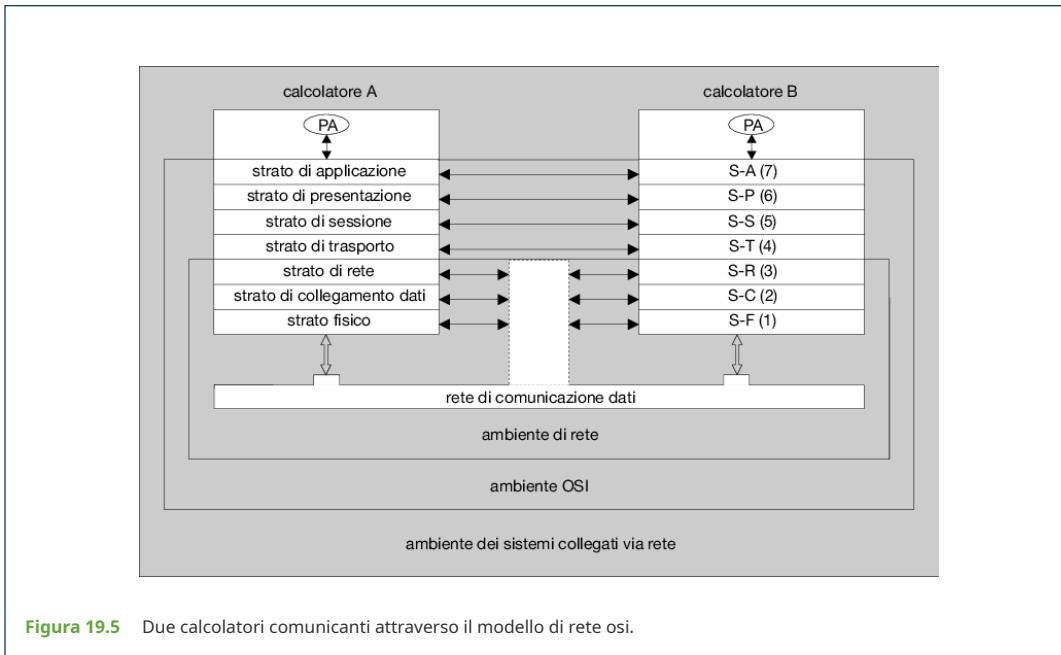


Figura 19.5 Due calcolatori comunicanti attraverso il modello di rete osi.

- Strato 1: Strato fisico. È responsabile della gestione degli aspetti meccanici ed elettrici della trasmissione fisica di un flusso di bit. Nello strato fisico i sistemi di comunicazione devono accordarsi sulla rappresentazione elettrica delle cifre binarie 0 e 1, in modo che quando s'inviano i dati come flusso di segnali elettrici il ricevitore possa interpretare i dati correttamente come dati binari. Questo strato si realizza nell'hardware dei dispositivi di rete ed è il responsabile dell'invio dei bit.
- Strato 2: Strato di collegamento dati. È responsabile delle funzioni di gestione dei *frame*, o di parti di pacchetti, comprese l'individuazione e la correzione degli errori che si sono verificati nello strato fisico. Spedisce frame fra siti collegati direttamente.
- Strato 3: Strato di rete. È responsabile della fornitura dei collegamenti e dell'instradamento dei pacchetti nella rete di comunicazione, comprese la gestione degli indirizzi dei pacchetti in uscita, la decodifica degli indirizzi dei pacchetti in arrivo e la gestione delle informazioni d'instradamento per rispondere adeguatamente ai cambiamenti dei livelli del carico. I router operano in questo strato.
- Strato 4: Strato di trasporto. È responsabile del trasferimento dei messaggi tra gli host, comprese la suddivisione dei messaggi in pacchetti, l'ordinamento dei pacchetti, il controllo del flusso per evitare la congestione.
- Strato 5: Strato di sessione. È responsabile della realizzazione di sessioni, ossia di protocolli di comunicazione da processo a processo.
- Strato 6: Strato di presentazione. È responsabile della risoluzione delle differenze di formato che si possono presentare tra diversi siti della rete, fra cui la conversione di caratteri e le modalità *half duplex-full duplex* (echo dei caratteri).
- Strato 7: Strato di applicazione. È responsabile dell'interazione diretta con gli utenti. Questo strato tratta il trasferimento di file, i protocolli per la gestione dei login remoti, la posta elettronica, e i protocolli per le basi di dati distribuite.

La Figura 19.6 riassume la pila di protocolli osi – un insieme di protocolli cooperanti – e illustra il flusso fisico dei dati. Come si è detto, dal punto di vista logico ciascuno strato di una pila di protocolli comunica con lo strato corrispondente negli altri sistemi. Fisicamente, però, un messaggio parte dallo strato di applicazione, o da sopra di esso, e passa attraverso ciascun livello inferiore. Ciascuno strato può modificare il messaggio e includere dati d'intestazione del messaggio per lo strato corrispondente del ricevente. Infine il messaggio raggiunge lo strato della rete di comunicazione e viene trasferito sotto forma di uno o più pacchetti (Figura 19.7). Lo strato di collegamento dati del sistema di destinazione riceve questi dati e il messaggio risale attraverso la pila di protocolli; viene analizzato, modificato e privato delle intestazioni; infine raggiunge lo strato di applicazione e può essere usato dal processo ricevente.

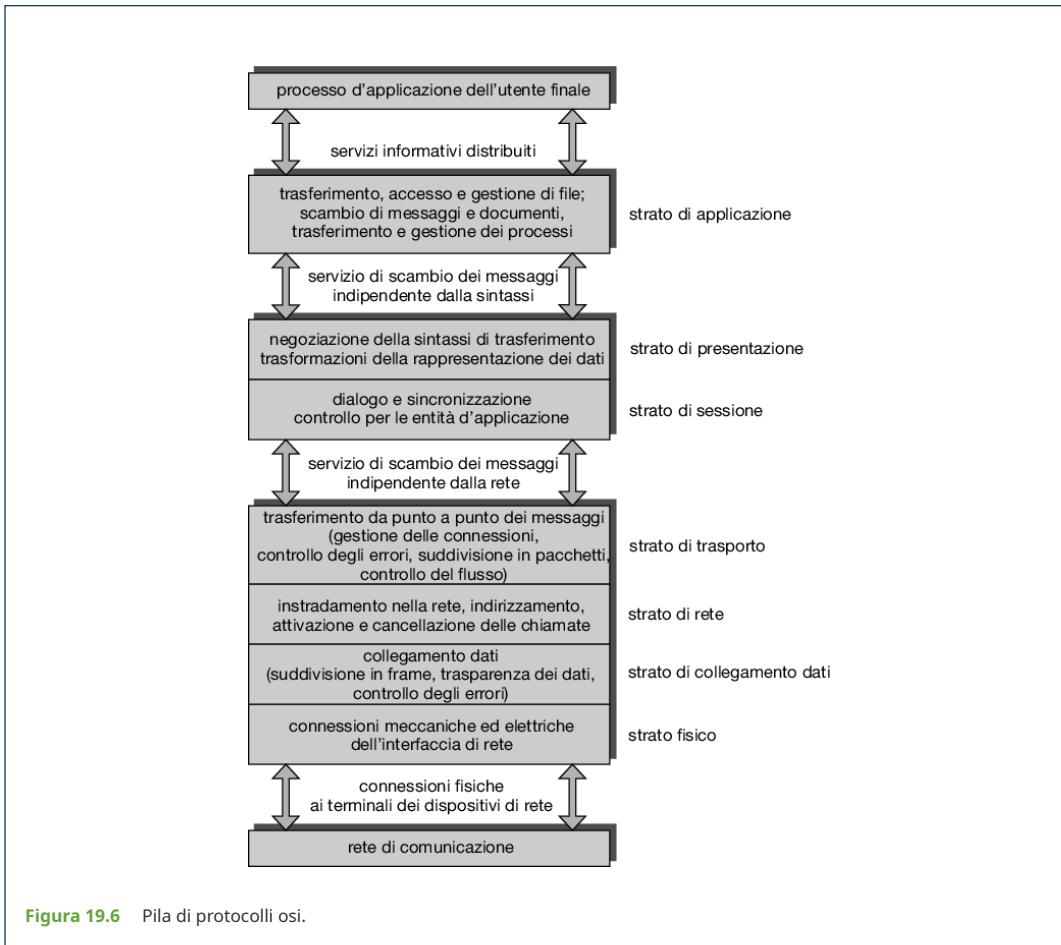
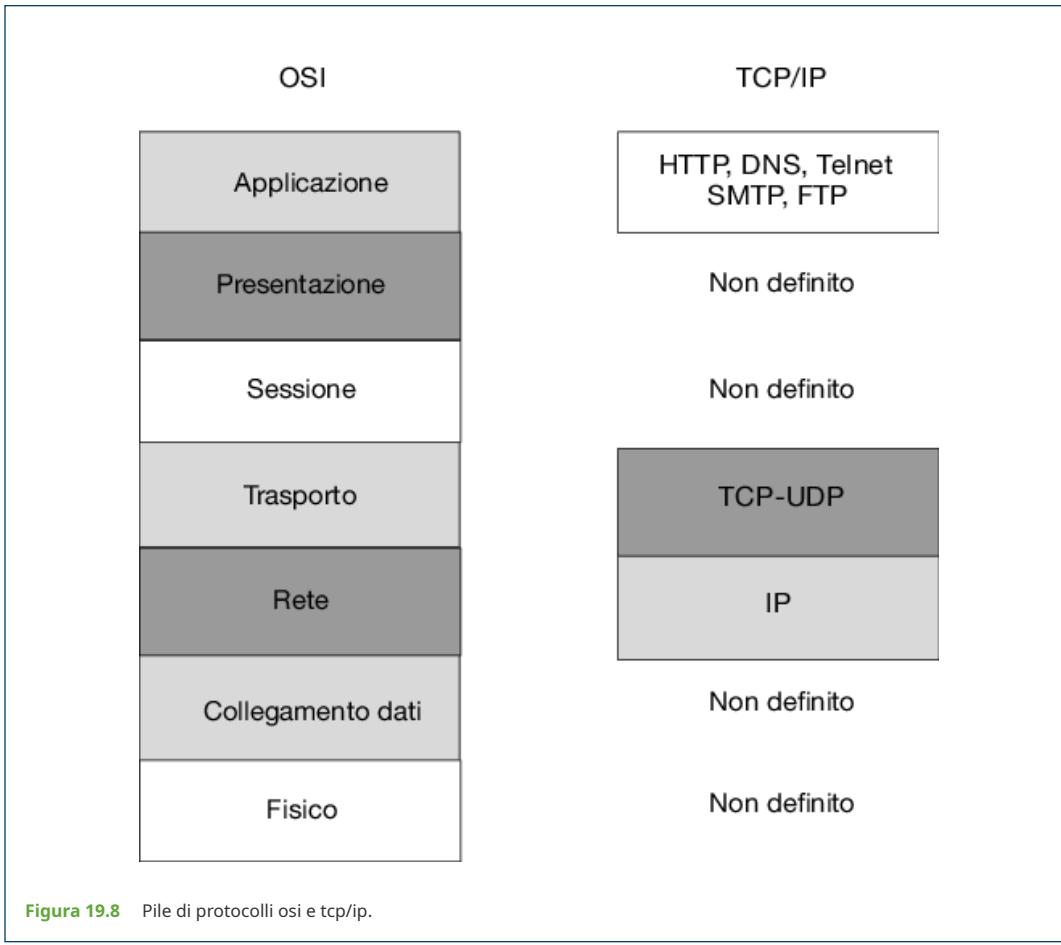




Figura 19.7 Messaggio di rete osi.

Il modello osi formalizza alcuni precedenti lavori svolti sui protocolli di rete; era stato sviluppato alla fine degli anni '70 e non è attualmente utilizzato; la pila di protocolli più ampiamente utilizzata è il modello tcp/ip (talvolta chiamato *modello Internet*), adottata da tutti i siti Internet. La pila di protocolli tcp/ip ha meno strati di quelli stabiliti dal modello osi, e poiché combina diverse funzioni in ciascuno strato è, teoricamente, più difficile da realizzare ma più efficiente delle reti osi. La relazione tra i modelli osi e tcp/ip è illustrata dalla Figura 19.8.



Lo strato applicativo tcp/ip comprende diversi protocolli molto diffusi su Internet, come http, ftp, ssh, dns e smtp. Lo strato di trasporto comprende i protocolli udp (*user datagram protocol*), inaffidabile e privo di connessione, e tcp (*transmission control protocol*), affidabile e orientato alla connessione. Il protocollo Internet (ip) è responsabile dell'instradamento dei datagrammi ip lungo Internet. Il modello tcp/ip non specifica formalmente lo strato di collegamento dati e fisico, permettendo al traffico tcp/ip di fluire su qualunque rete fisica. Nel Paragrafo 19.3.3 si considera il modello tcp/ip implementato su una rete Ethernet.

Nel disegno e nell'implementazione di un moderno protocollo di comunicazione la preoccupazione per la sicurezza assume una notevole importanza. Per una comunicazione sicura sono necessari sia un'autenticazione forte sia la crittografia. L'autenticazione forte assicura che il mittente e il destinatario di una comunicazione siano quelli desiderati. La crittografia protegge i contenuti della comunicazione da eventuali intercettazioni. L'autenticazione debole e la comunicazione in chiaro sono comunque, per diverse ragioni, ancora molto utilizzate. Quando la maggior parte dei protocolli oggi comunemente utilizzati fu progettata, la sicurezza era spesso meno importante rispetto alle prestazioni, alla semplicità e all'efficienza. L'eredità lasciata da queste scelte è tangibile ancora oggi: aggiungere sicurezza a infrastrutture esistenti è un'operazione difficile e complessa.

L'autenticazione forte richiede un protocollo di negoziazione a più passi o un dispositivo di autenticazione, rendendo così il protocollo più complesso. Le cpu moderne possono eseguire la crittografia in maniera efficiente, spesso offrendo istruzioni per velocizzarla, e i sistemi demandano spesso questa attività a processori dedicati, così che le prestazioni non siano compromesse. Le comunicazioni a lunga distanza possono essere rese sicure autenticando gli estremi della connessione e crittografando il flusso di pacchetti in una rete privata virtuale, come discusso nel Paragrafo 16.4.2. Le comunicazioni in una lan restano in chiaro nella maggior parte dei casi, ma protocolli come nfs nella sua versione 4, che include un'autenticazione forte incorporata e la crittografia dei dati, possono aiutare a migliorare anche la sicurezza di una lan.

19.3.3 Un esempio: TCP/IP

A questo punto è utile tornare al problema della risoluzione dei nomi per esaminarne il funzionamento nella pila di protocolli tcp/ip nella rete Internet. Consideriamo poi le elaborazioni necessarie al trasferimento di un pacchetto tra due calcolatori appartenenti a due diverse reti Ethernet. La nostra descrizione è basata sui protocolli ipv4 che sono i più utilizzati al giorno d'oggi.

In una rete tcp/ip, ciascun calcolatore possiede un nome e un indirizzo Internet a esso associato. Entrambi i valori devono essere unici e sono suddivisi per facilitare la gestione dello spazio dei nomi. Il nome è gerarchico (come descritto in precedenza) e specifica sia il

nome del calcolatore sia le organizzazioni a cui il calcolatore è associato. L'indirizzo ip del calcolatore è suddiviso in un numero di rete e un numero di macchina. Le proporzioni della suddivisione variano secondo la dimensione della rete. Una volta che gli amministratori di Internet hanno assegnato il numero di rete, nel sito con quel numero si è liberi di assegnare gli identificatori dei calcolatori come meglio si crede.

Il sistema mittente ricerca all'interno delle proprie tabelle d'instradamento un router cui spedire il pacchetto. I router sfruttano la parte di rete dell'identificatore del calcolatore per trasferire il pacchetto dalla rete di partenza a quella di destinazione. Il sistema destinatario riceve quindi il pacchetto, che può essere il messaggio completo o semplicemente una sua parte. In quest'ultimo caso, prima di ricostruire il messaggio e passarlo al processo destinatario, è necessario attendere l'arrivo degli altri pacchetti.

A questo punto si sa come un pacchetto passa dalla rete di partenza alla sua destinazione, ma ci si può chiedere come fa un pacchetto, all'interno di una rete, a passare dal mittente (calcolatore o router) al ricevente. Ogni dispositivo Ethernet ha un numero unico che lo individua detto indirizzo mac (*medium access control address*). Due dispositivi di una rete locale comunicano tra loro servendosi unicamente di questo numero. Se un sistema necessita di inviare dati a un altro, il software di rete genera un pacchetto arp (*address resolution protocol*) contenente l'indirizzo ip del sistema di destinazione. Questo pacchetto viene diffuso (*broadcast*) a tutti gli altri sistemi della rete Ethernet.

Il broadcast adopera uno speciale indirizzo di rete (di solito l'indirizzo massimo) per segnalare a tutti i calcolatori connessi di ricevere ed elaborare il pacchetto. I pacchetti trasmessi per diffusione non sono ritrasmessi dai gateway, perciò sono ricevuti solamente dai sistemi connessi alla rete locale. Solo il sistema il cui indirizzo ip corrisponde all'indirizzo ip dell'arp risponde e invia il proprio indirizzo mac al sistema che ha fatto l'interrogazione. Per motivi di efficienza, il calcolatore copia in una tabella cache interna la coppia <indirizzo ip, indirizzo mac>. Gli elementi della cache sono fatti "invecchiare" (*aged*) fino a essere rimossi qualora in un certo tempo non sia richiesto un accesso a quel sistema destinatario. In questo modo i calcolatori rimossi dalla rete vengono *dimenticati*. Per aumentare le prestazioni, si possono appuntare nella cache arp gli elementi relativi ai calcolatori più usati.

Dopo che un dispositivo Ethernet ha annunciato il suo identificatore del calcolatore e il suo indirizzo, la comunicazione può cominciare. Un processo può specificare il nome del calcolatore con cui intende comunicare, il software di rete prende questo nome e determina l'indirizzo ip della destinazione attraverso un'interrogazione al dns o una voce nel file locale `hosts`, in cui le traduzioni possono essere registrate manualmente. Il messaggio passa dallo strato di applicazione all'interfaccia hardware, attraversando tutti gli strati di programmi e protocolli. Al livello hardware, il pacchetto (o i pacchetti) contiene l'indirizzo Ethernet al suo inizio e una coda, che indica la fine del pacchetto e contiene checksum che serve a rilevare eventuali errori nel pacchetto stesso (Figura 19.9). Il pacchetto viene posto nella rete dal dispositivo Ethernet. La sezione dei dati del pacchetto può contenere tutto il messaggio o solamente una parte dei suoi dati, così come può contenere alcune delle intestazioni di livello superiore che compongono il messaggio. In altre parole, tutti gli elementi del messaggio originale si devono inviare dall'indirizzo di provenienza alla destinazione e tutte le intestazioni poste sopra lo strato 802.3 (lo strato di collegamento dati) sono incluse nel pacchetto Ethernet sotto forma di dati.

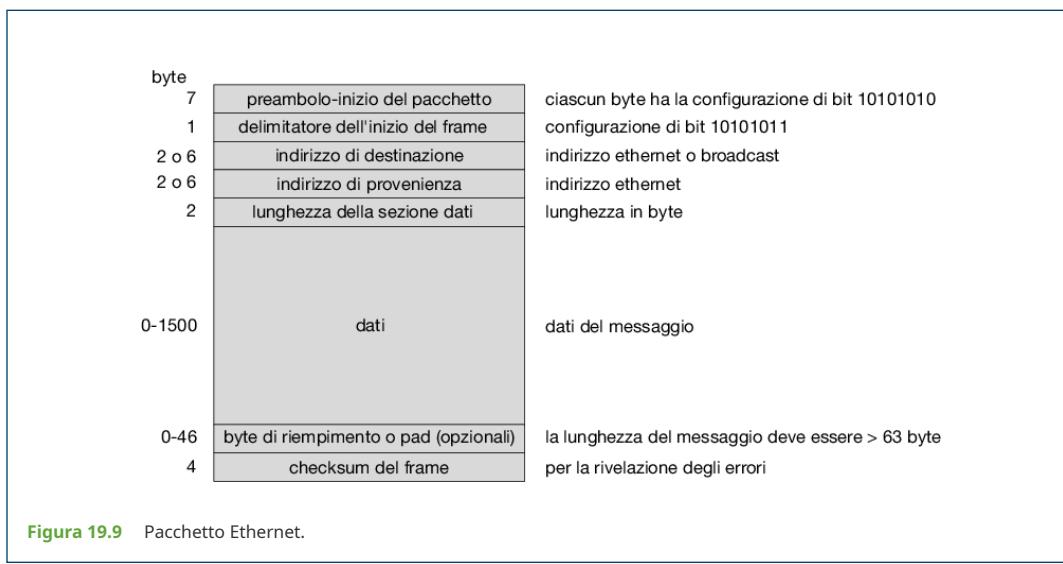


Figura 19.9 Pacchetto Ethernet.

Se la destinazione si trova nella stessa rete locale dell'indirizzo di provenienza, il sistema può analizzare la propria cache di arp, trovare l'indirizzo Ethernet del calcolatore e immettere il pacchetto sulla linea. Il dispositivo Ethernet della destinazione riconosce il proprio indirizzo e legge il pacchetto, passandolo ai livelli superiori della pila di protocolli.

Se il sistema di destinazione si trova in una rete diversa da quella di provenienza, il sistema di partenza cerca il router appropriato sulla propria rete e gli spedisce il pacchetto. I router trasmettono quindi il pacchetto lungo la wan finché esso raggiunge la sua rete di destinazione. Il router che connette la rete di destinazione esamina la propria cache di arp, cerca il numero Ethernet della destinazione e spedisce il pacchetto al calcolatore. Attraverso tutti questi trasferimenti, l'intestazione dello strato di collegamento dati può cambiare man mano viene usato l'indirizzo Ethernet del successivo router della catena, mentre le altre intestazioni del pacchetto restano invariate finché il pacchetto non venga ricevuto ed elaborato dalla pila di protocolli e quindi passato dal kernel al processo ricevente.

19.3.4 I protocolli di trasporto UDP e TCP

Quando un host con uno specifico indirizzo ip riceve un pacchetto deve in qualche modo trasferirlo al corretto processo in attesa. I protocolli di trasporto tcp e udp identificano i processi destinatari (e mittenti) attraverso l'uso di un numero di porta. Pertanto, un host con un singolo indirizzo ip può avere diversi processi server in esecuzione e in attesa di pacchetti, purché ciascun processo specifichi un numero di porta diverso. Per impostazione predefinita, molti servizi comuni utilizzano un numero di porta noto; tra gli altri, menzioniamo ftp (21), ssh (22), smtp (25) e http (80). Per esempio, se ci si vuole connettere a un sito web "http" tramite il browser web, quest'ultimo tenterà automaticamente di connettersi alla porta 80 sul server, utilizzando 80 come numero di porta nell'intestazione tcp. Per avere un elenco completo delle porte note, potete accedere alla vostra macchina Linux o unix preferita e osservare il contenuto del file `/etc/services`.

Il livello di trasporto può svolgere altre funzioni oltre al semplice collegamento di un pacchetto di rete a un processo in esecuzione. Può per esempio, se lo si desidera, incrementare l'affidabilità di un flusso di pacchetti di rete. Per spiegare come ciò avviene, descriviamo in seguito alcuni comportamenti generali dei protocolli di trasporto udp e tcp.

19.3.4.1 UDP (User Datagram Protocol)

Il protocollo di trasporto udp *non è affidabile*, in quanto è un'estensione minimale di ip con l'aggiunta di un numero di porta. L'intestazione udp è in effetti molto semplice e contiene solo quattro campi: numero di porta di origine, numero di porta di destinazione, lunghezza e checksum. Usando udp i pacchetti possono essere inviati rapidamente a una destinazione, ma possono anche andare persi, poiché non ci sono garanzie di consegna fornite dagli strati inferiori della pila di rete. I pacchetti possono inoltre arrivare a un destinatario fuori sequenza. Spetta all'applicazione individuare questi errori e correggerli (o decidere di non farlo).

La Figura 19.10 illustra uno scenario comune che comporta la perdita di un pacchetto tra un client e un server quando si utilizza il protocollo udp. Si noti che questo protocollo è anche detto protocollo senza connessione (*connectionless*), perché non avviene alcuna configurazione della connessione all'inizio della trasmissione per impostare lo stato: il client inizia semplicemente a inviare i dati. Allo stesso modo, non vi è alcuna chiusura della connessione.

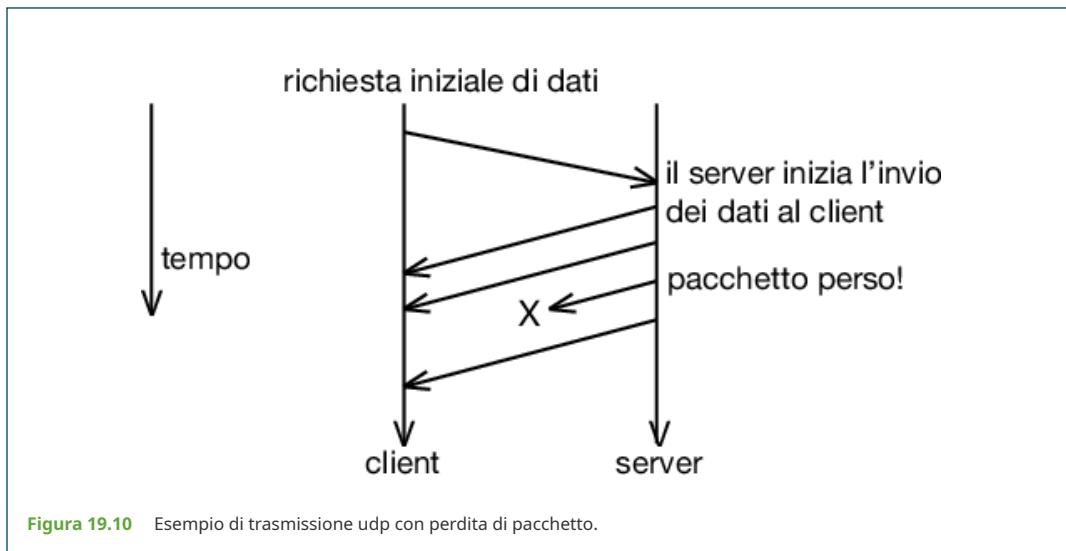


Figura 19.10 Esempio di trasmissione udp con perdita di pacchetto.

Il client inizia inviando una sorta di richiesta di informazioni al server. Il server risponde inviando quattro datagrammi, o pacchetti, al client ma, sfortunatamente, uno dei pacchetti viene scartato da un router sovraccarico. Il client deve accontentarsi di tre pacchetti oppure utilizzare i meccanismi implementati nell'applicazione per richiedere il pacchetto mancante. Dobbiamo quindi utilizzare un protocollo di trasporto differente se le nostre esigenze richiedono garanzie aggiuntive di affidabilità gestite dalla rete.

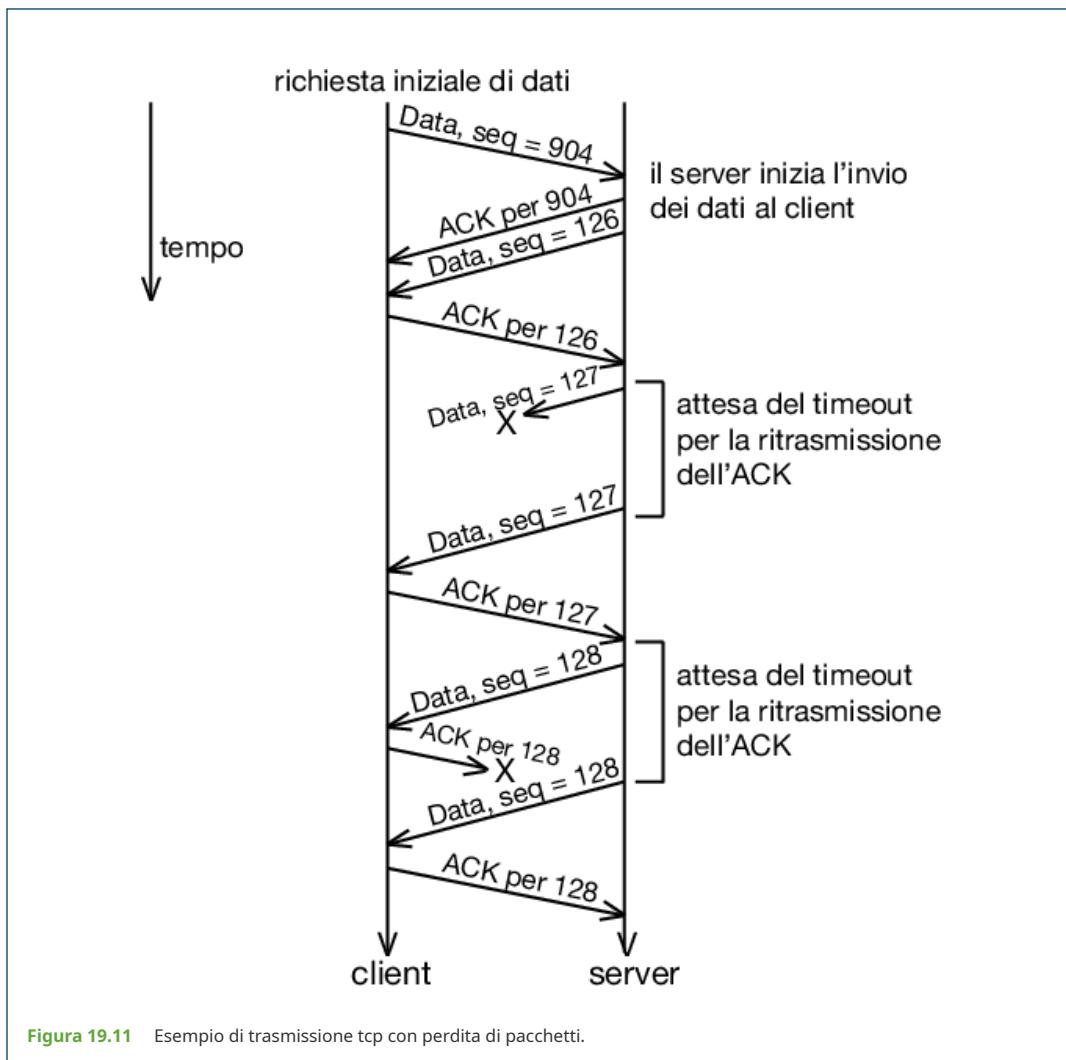
19.3.4.2 TCP (Transmission Control Protocol)

tcp è un protocollo di trasporto *affidabile* e orientato alla connessione. Oltre a specificare i numeri di porta per identificare i processi mittente e destinatario su host diversi, tcp fornisce un'astrazione che consente a un processo mittente di un host di inviare un flusso di byte ordinato e ininterrotto attraverso la rete a un processo di destinazione di un altro host. Queste operazioni sono realizzate attraverso i seguenti meccanismi.

- Ogni volta che un host invia un pacchetto, il destinatario deve inviare un pacchetto di acknowledgment, o ack, per notificare al mittente che il pacchetto è stato ricevuto. Se l'ack non viene ricevuto prima della scadenza di un timer, il mittente invia nuovamente il pacchetto.
- tcp introduce **numeri di sequenza** nell'intestazione tcp di ogni pacchetto. Questi numeri consentono al destinatario di (1) mettere i pacchetti in ordine prima di trasmettere i dati al processo richiedente e (2) rilevare i pacchetti mancanti nel flusso di byte.

- Le connessioni tcp vengono avviate con uno scambio di pacchetti di controllo tra il mittente e il destinatario (spesso chiamato *three-way handshake*) e terminate mediante pacchetti di controllo responsabili di chiudere la connessione. Questi pacchetti di controllo consentono sia al mittente sia al destinatario di modificare lo stato.

La Figura 19.11 mostra un possibile scambio con tcp (sono omesse le fasi di apertura e chiusura della connessione). Dopo che la connessione è stata stabilita, il client invia un pacchetto di richiesta al server con numero di sequenza 904. A differenza dell'esempio relativo a udp, il server deve ora inviare un pacchetto ack al client. Successivamente, il server inizia a inviare il suo flusso di pacchetti di dati, iniziando con un numero di sequenza diverso. Il client invia un pacchetto ack per ogni pacchetto di dati che riceve. Sfortunatamente, il pacchetto con numero di sequenza 127 viene perso e nessun pacchetto ack viene inviato dal client. Quando scade il tempo di attesa del pacchetto ack, il mittente deve quindi reinviare il pacchetto dati 127. In seguito, il server invia il pacchetto dati con numero di sequenza 128, ma l'ack di risposta viene perso. Poiché il server non riceve l'ack, deve inviare nuovamente il pacchetto dati 128. Il client riceve quindi un pacchetto duplicato, ma sapendo che ha precedentemente ricevuto un pacchetto con lo stesso numero di sequenza, scarta il duplicato. Tuttavia, è necessario inviare un altro ack al server per consentirgli di continuare.



Nell'effettiva specifica tcp, non è richiesto un ack per ogni singolo pacchetto, ma è possibile inviare un ack cumulativo per segnalare una serie di pacchetti ricevuti. Il server può inoltre inviare numerosi pacchetti dati in sequenza prima di attendere gli ack, per sfruttare il throughput della rete.

tcp aiuta anche a regolare il flusso dei pacchetti grazie a due meccanismi chiamati controllo di flusso e controllo della congestione. Il controllo di flusso impedisce che il mittente superi la capacità del destinatario. Per esempio, il destinatario potrebbe avere una connessione più lenta o componenti hardware (come una scheda di rete o un processore) più lenti. Lo stato del controllo di flusso può essere inviato al mittente all'interno dei pacchetti ack del destinatario, per richiedere di rallentare o accelerare la trasmissione. Il controllo della congestione tenta di rilevare lo stato delle reti (e in genere dei router) tra il mittente e il destinatario. Quando un router viene sovraccaricato di pacchetti, tende a scartarli. Ciò determina un timeout dell'ack, che porta all'invio di ulteriori pacchetti che, a

loro volta, saturano la rete. Per evitare questa condizione, il mittente tiene monitorata la connessione al fine di individuare i pacchetti scartati, verificando quanti pacchetti non vengono corrisposti da un ack. Se ci sono troppi pacchetti scartati, il mittente abbassa la velocità di invio, contribuendo così a garantire che la connessione tcp sia equa nei confronti delle altre connessioni che sono attive nello stesso momento.

Utilizzando un protocollo di trasporto affidabile come tcp, un sistema distribuito non ha bisogno di una logica aggiuntiva per gestire pacchetti persi o fuori sequenza. Tuttavia, tcp è più lento di udp.

19.4 Sistemi operativi di rete e distribuiti

In questo paragrafo si descrivono due categorie generali di sistemi operativi orientati alle reti: i sistemi operativi di rete e i sistemi operativi distribuiti. I sistemi operativi di rete sono più semplici da realizzare, ma spesso sono più difficili da usare dei sistemi operativi distribuiti, che offrono più servizi.

19.4.1 Sistemi operativi di rete

Un sistema operativo di rete offre un ambiente nel quale gli utenti, che sanno della presenza di più calcolatori, possono accedere alle risorse remote iniziando una sessione di lavoro sugli appropriati calcolatori remoti oppure trasferendo dati dai calcolatori remoti al proprio. Al giorno d'oggi tutti i sistemi operativi general-purpose e persino i sistemi operativi mobili come Android e iOS sono sistemi operativi di rete.

19.4.1.1 Login remoto

Una funzione importante dei sistemi operativi di rete è consentire agli utenti di effettuare un login remoto. A tale scopo la rete Internet fornisce la funzione `ssh`. Per illustrare questa funzione si supponga che un utente, al Westminster College, voglia compiere alcune elaborazioni servendosi di `kristen.cs.yale.edu`, un calcolatore che si trova alla Yale University. Perché ciò sia possibile, l'utente deve avere un account valido sul calcolatore remoto. Per effettuare il login remoto l'utente usa il comando

```
ssh kristen.cs.yale.edu
```

Questo comando fa sì che si realizzi una connessione socket criptata tra il calcolatore locale del Westminster College e il calcolatore `kristen.cs.yale.edu`. Stabilita tale connessione, il software di rete crea un collegamento bidirezionale, trasparente, tale che tutti i caratteri inseriti dall'utente siano inviati a un processo in `kristen.cs.yale.edu`, e tutti i dati emessi da questo processo siano rispediti all'utente. Il processo nel calcolatore remoto chiede all'utente il suo nome utente e la relativa password; ricevute le corrette informazioni, il processo agisce per conto dell'utente, che può compiere le proprie elaborazioni servendosi del calcolatore remoto come ogni utente locale.

19.4.1.2 Trasferimento di file remoti

Un'altra tra le funzioni principali di un sistema operativo di rete consiste nel fornire un meccanismo per il trasferimento di file remoti tra calcolatori. In un ambiente di questo tipo ogni calcolatore mantiene il proprio file system locale. Se un utente in un sito (per esempio, Kurt su `albion.edu`) vuole accedere a un file di Becca che si trova in un altro calcolatore (per esempio, `colby.edu`), deve copiare esplicitamente il file dal calcolatore da Colby, nel Maine, al calcolatore di Albion, nel Michigan.

La rete Internet offre meccanismi, il programma `ftp` (*file transfer protocol*) e il più sicuro `sftp` (*secure file transfer protocol*), che permettono di compiere tale trasferimento. Si supponga che l'utente Carla di `wesleyan.edu` voglia copiare un file posseduto da Owen su `kzoo.edu`; l'utente deve prima invocare il programma `sftp`:

```
sftp owen@kzoo.edu
```

Quindi il programma richiede il nome utente e la relativa password; una volta che il programma ha ricevuto le corrette informazioni, l'utente può utilizzare una serie di comandi per trasferire file e navigare il file system remoto, tra cui:

- `get` trasferisce un file da un calcolatore remoto a uno locale;
- `put` trasferisce un file da un calcolatore locale a uno remoto;
- `ls` o `dir` elencano i file presenti nella directory corrente nel calcolatore remoto;
- `cd` cambia la directory corrente nel calcolatore remoto.

Inoltre vari comandi consentono di cambiare i modi di trasferimento (per file binari o ascii) e per determinare lo stato della connessione.

19.4.1.3 Archiviazione in cloud

Le applicazioni base di archiviazione in cloud consentono agli utenti di trasferire file in modo molto simile a come avviene con `ftp`. Gli utenti possono caricare file su un server cloud, scaricare file sul computer locale e condividere file con altri utenti del servizio cloud tramite un collegamento web o un altro meccanismo di condivisione, mediante un'interfaccia grafica. Esempi comuni includono Dropbox e Google Drive.

Un aspetto importante di `ssh`, `ftp` e delle applicazioni di archiviazione in cloud è che esse richiedono all'utente di modificare i paradigmi. `ftp`, per esempio, richiede all'utente di conoscere un insieme di comandi completamente diverso dai normali comandi del sistema operativo. Con `ssh` l'utente deve conoscere i comandi appropriati sul sistema remoto. Per esempio, un utente su un computer Windows che si connette da remoto a un dispositivo unix deve usare comandi unix per l'intera durata della sessione `ssh` (nel linguaggio delle reti, una sessione è un ciclo completo di comunicazione, che inizia spesso con un accesso e un'autenticazione e

termina con una disconnessione che chiude la comunicazione). Nel caso delle applicazioni di archiviazione basate su cloud, gli utenti devono solitamente accedere al servizio cloud (spesso attraverso un browser web o un'applicazione dedicata) e poi utilizzare una serie di comandi grafici per caricare, scaricare o condividere file. Ovviamente, gli utenti trovano più conveniente evitare di utilizzare un insieme di comandi diversi. I sistemi operativi distribuiti sono progettati per risolvere questo problema.

19.4.2 Sistemi operativi distribuiti

In un sistema operativo distribuito gli utenti accedono alle risorse remote nello stesso modo in cui accedono alle risorse locali. I trasferimenti di dati e processi tra siti sono sotto il controllo del sistema operativo distribuito. A seconda degli obiettivi del sistema, il sistema può implementare la migrazione dei dati, la migrazione delle computazioni, la migrazione dei processi o qualsiasi combinazione di queste funzioni.

19.4.2.1 Migrazione dei dati

Si supponga che un utente del sito *A* voglia accedere a dati (per esempio, contenuti in un file) che risiedono nel sito *B*. Il sistema può trasferire i dati impiegando uno fra due metodi di base. Un approccio alla migrazione dei dati consiste nel trasferire tutto il file al sito *A*. Da questo punto in poi l'intero accesso al file è di tipo locale. Quando l'utente non ha più necessità di accedere al file, se il file era stato modificato, si ritrasmette una sua copia al sito *B*. Si devono trasferire tutti i dati anche se il file ha subito solo una lieve modifica. Questo metodo, che si può considerare come un sistema ftp automatico, era usato nel file system Andrew, ma è stato ritenuto troppo inefficiente.

L'altro approccio consiste nel trasferire al sito *A* solo le parti del file immediatamente necessarie. Se in seguito è richiesta un'altra sua parte, si esegue un nuovo trasferimento. Quando l'utente non avrà più bisogno di accedere al file, si ritrasmetterà al sito *B* qualsiasi sua parte sia stata modificata (si noti l'analogia con la paginazione su richiesta). Questo metodo è adoperato dai più moderni sistemi distribuiti.

Qualunque metodo si utilizzi, la migrazione dei dati comprende assai più del mero trasferimento dei dati da un sito all'altro. Se i due siti coinvolti nel trasferimento non fossero direttamente compatibili (per esempio, se impiegassero codici dei caratteri diversi o rappresentassero i numeri interi con un diverso numero o ordinamento dei bit), il sistema dovrebbe eseguire anche diverse traduzioni dei dati.

19.4.2.2 Migrazione delle computazioni

In alcune circostanze si può voler trasferire le computazioni anziché i dati; questo metodo si chiama migrazione delle computazioni. Si consideri, per esempio, un job che necessiti di accedere a diversi file di grandi dimensioni che risiedono in diversi siti per ottenere un sommario di tutti quei file. È più conveniente accedere ai file nei siti in cui risiedono e riportare i risultati al sito che ha iniziato il calcolo. In generale s'impiega il comando remoto se il tempo di trasferimento dei dati è maggiore del tempo d'esecuzione del comando remoto.

Una computazione di questo genere si può eseguire in diversi modi. Si supponga che il processo *P* voglia accedere a un file presente nel sito *A*. L'accesso al file si esegue nel sito *A* e può essere avviato da una rpc. Una rpc usa protocolli di rete per far eseguire una procedura in un sistema remoto (Paragrafo 3.8.2). Il processo *P* invoca una procedura predefinita nel sito *A*; tale procedura esegue adeguatamente il proprio compito e riporta i risultati a *P*.

In alternativa il processo *P* può inviare un *messaggio* al sito *A*; il sistema operativo di *A* crea un nuovo processo *Q* la cui funzione è quella di eseguire il compito designato; quando il processo *Q* termina l'esecuzione, invia il risultato richiesto a *P* per mezzo del sistema di messaggistica. In questo schema il processo *P* si può eseguire in modo concorrente al processo *Q* e, in effetti, più processi si possono eseguire in modo concorrente in diversi siti.

Entrambi i metodi si possono usare per accedere a più file residenti in vari siti. Una rpc può causare l'invocazione di un'altra rpc, o addirittura il trasferimento di messaggi a un altro sito. Analogamente il processo *Q*, durante la propria esecuzione, può inviare un messaggio a un altro sito, che a sua volta crea un altro processo. Questo processo può inviare un messaggio di risposta a *Q*, oppure ripetere il ciclo.

19.4.2.3 Migrazione dei processi

Una logica estensione della migrazione delle computazioni è la migrazione dei processi. Quando si esegue un processo, non sempre l'esecuzione si svolge nel sito nel quale è stata avviata. L'intero processo, o parti di esso, si possono eseguire in siti diversi. Questo schema può essere utilizzato per diversi scopi.

- Bilanciamento del carico. I processi (o sottoprocessi) si possono distribuire tramite la rete per equilibrare il carico di lavoro.
- Velocizzazione dell'elaborazione. Se un singolo processo si può dividere in un certo numero di sottoprocessi che si possono eseguire in modo concorrente in siti diversi, è possibile ridurre il tempo di completamento totale.
- Preferenze hardware. Il processo può avere caratteristiche che lo rendono più adatto a un'esecuzione in un'unità d'elaborazione specializzata (per esempio, un'inversione di matrice in un processore vettoriale anziché in un'ordinaria cpu).
- Preferenze software. Il processo può richiedere un programma disponibile solo in un particolare sito, ma o il programma non può essere spostato, o è meno costoso spostare il processo.
- Accesso ai dati. Come nella migrazione delle computazioni, se i dati da usare nella computazione sono numerosi, l'esecuzione remota del processo può essere più efficiente del trasferimento dei dati.

Per spostare i processi in una rete di calcolatori si possono usare due metodi complementari. Col primo, il sistema può cercare di nascondere al client la migrazione del processo. Il client non ha quindi bisogno di codificare il suo programma esplicitamente per realizzare la migrazione. Questo schema si usa normalmente per ottenere un bilanciamento del carico e una velocizzazione delle elaborazioni tra sistemi omogenei, poiché essi non richiedono un'interazione con l'utente nell'esecuzione remota dei programmi.

L'altro metodo consiste nel permettere (o richiedere) all'utente di specificare esplicitamente come il processo debba migrare. Questo metodo si usa normalmente nelle situazioni in cui si deve trasferire un processo per soddisfare una preferenza hardware o software.

Il lettore si è probabilmente accorto del fatto che il World Wide Web ha molte caratteristiche di un ambiente d'elaborazione distribuito. Di certo offre la migrazione dei dati (tra un server web e un client) così come la migrazione delle computazioni: un client può per esempio attivare un'operazione su una base di dati di un server web. Infine, con Java, Javascript e linguaggi simili, fornisce una forma di migrazione dei processi: le *Java applet* e gli script Javascript sono inviati dal server al client dove vengono eseguiti. Un sistema operativo di rete offre la maggior parte di queste funzioni, ma un sistema operativo distribuito le integra in maniera uniforme e le rende più facilmente accessibili. Il risultato è uno strumento potente e facile da usare, una delle ragioni dell'enorme crescita del Web.

19.5 Progettazione di sistemi distribuiti

I progettisti di un sistema distribuito devono affrontare una serie di sfide progettuali: il sistema deve essere robusto, in modo da poter sopportare i guasti; il sistema deve essere trasparente in termini di posizione dei file e mobilità degli utenti; il sistema deve essere scalabile per consentire l'incremento di potenza di calcolo, spazio di archiviazione o utenti. Presenteremo brevemente tali questioni nel seguente paragrafo, mentre nel successivo le tratteremo nel contesto di progetti specifici di file system distribuiti.

19.5.1 Robustezza

Un sistema distribuito può soffrire di guasti fisici di diverso tipo. Il guasto di un collegamento, di un sito e la perdita di un messaggio sono quelli più frequenti. Per assicurare la robustezza del sistema occorre individuare tutti questi guasti, riconfigurare il sistema per poter continuare la computazione ed effettuare il ripristino una volta riparato il collegamento o il sito.

Un sistema è tollerante ai guasti quando può tollerare un determinato livello di errore e continuare a funzionare normalmente. Il grado di tolleranza ai guasti dipende dal progetto del sistema distribuito e dallo specifico guasto. Ovviamente, un sistema è tanto migliore quanto maggiore è la tolleranza ai guasti.

Il termine *tolleranza ai guasti* si usa in senso lato: gli errori di comunicazione, alcuni guasti della macchina, la rottura dei dispositivi di memoria e il deterioramento degli strumenti di memorizzazione dovrebbero essere tutti in una certa misura tollerabili. Un sistema tollerante ai guasti (*fault-tolerant*) deve continuare a funzionare, eventualmente con efficacia ridotta, anche quando si presentano tali guasti. Le conseguenze negative possono riguardare le prestazioni, il funzionamento, o entrambi gli aspetti; ma in ogni caso, il peggioramento dovrebbe essere proporzionale ai guasti che l'hanno causato. Un sistema che si ferma quando si guasta uno solo dei suoi componenti non si può certo considerare tollerante ai guasti.

Sfortunatamente la tolleranza ai guasti è costosa e difficile da realizzare. A livello di rete, per evitare interruzioni alla comunicazione sono necessari diversi percorsi di comunicazione冗余 e dispositivi di rete come switch e router. Un guasto nei dispositivi di memoria può causare la perdita del sistema operativo, delle applicazioni o dei dati. Le unità di memorizzazione possono utilizzare componenti hardware冗余 in grado di sostituirsi a vicenda in maniera automatica in caso di guasti. Inoltre i sistemi raid sono anche in grado di garantire un accesso continuo ai dati in caso di guasto di uno o più dischi (Paragrafo 11.8).

19.5.1.1 Rilevamento dei guasti

In un ambiente senza memoria condivisa non si è generalmente capaci di distinguere tra il guasto di un collegamento, il guasto di un sito e la perdita di un messaggio. Normalmente si può stabilire solo che si è verificato uno di questi guasti, ma non è possibile identificarne il tipo. Una volta rilevato un guasto è necessario intraprendere azioni appropriate, che dipendono dalla particolare applicazione.

Per rilevare un guasto in un collegamento o in un sito si usa una procedura detta di heartbeat. Si supponga che i siti *A* e *B* abbiano tra loro un collegamento fisico diretto. A intervalli fissi entrambi i siti s'inviano un messaggio *I-am-up*. Se il sito *A* non riceve questo messaggio entro un dato periodo di tempo, può supporre che il sito *B* sia guasto, che sia guasto il collegamento tra *A* e *B*, oppure che sia andato perduto il messaggio proveniente da *B*. A questo punto il sito *A* ha due possibilità: può attendere un altro periodo di tempo per ricevere un messaggio *I-am-up* da *B*, oppure può inviare un messaggio *Are-you-up?* a *B*.

Se il sito *A* non riceve un messaggio *I-am-up* oppure una risposta alla propria domanda, si può ripetere la procedura. L'unica conclusione che il sito *A* può trarre con sicurezza è che si è verificato un guasto.

Il sito *A* può tentare di distinguere tra la possibilità di un guasto al collegamento e quella di un guasto al sito inviando a *B* il messaggio *Are-you-up?* per un altro percorso, se questo esiste. Se e quando riceve questo messaggio, *B* risponde subito positivamente. Questa risposta positiva indica ad *A* che *B* è attivo, e che il guasto si trova nel collegamento diretto. Poiché non si può sapere a priori quanto tempo impieghi il messaggio per andare da *A* a *B* e tornare indietro, è necessario usare uno schema d'attesa (*time-out*). Quando *A* invia il messaggio *Are-you-up?*, specifica un intervallo di tempo entro il quale è disposto ad attendere la risposta di *B*. Se *A* riceve il messaggio di risposta entro quell'intervallo di tempo, può concludere con sicurezza che *B* è attivo; altrimenti *A* può concludere che si è verificata una (o più) delle seguenti situazioni:

- il sito *B* è fuori servizio;
- il collegamento diretto, se esiste, tra *A* e *B* è fuori servizio;
- il percorso alternativo da *A* a *B* è fuori servizio;
- il messaggio è andato perduto.

Il sito *A* non può in ogni caso determinare quale di questi eventi si sia verificato.

19.5.1.2 Riconfigurazione

Si supponga che il sito *A* abbia scoperto, per mezzo del meccanismo appena descritto, che si è verificato un guasto. Deve allora cominciare una procedura che permetta al sistema di riconfigurarsi e continuare a funzionare normalmente.

- Se si è guastato un collegamento diretto tra *A* e *B*, questa informazione deve essere diffusa a ogni sito del sistema, in modo da poter aggiornare adeguatamente le tabelle di routing.
- Se *A* ritiene che il sito *B* sia fuori servizio, poiché tale sito non è più raggiungibile, si deve informare della situazione ciascun sito del sistema, in modo che non tenti più di usare i servizi del sito guasto. Il guasto di un sito usato come coordinatore centrale di qualche attività, come il rilevamento delle situazioni di stallo, implica l'elezione di un nuovo coordinatore. Analogamente, se il sito guasto fa parte di un anello logico, è necessario costruire un nuovo anello logico. Occorre notare che, se il sito non si è guastato (vale a dire che è funzionante ma non raggiungibile), è possibile incorrere nella situazione spiacevole in cui due siti agiscono da coordinatore. Se la rete è partizionata, i due coordinatori, uno per ogni partizione, possono condurre

ad azioni conflittuali. Se i coordinatori sono responsabili, per esempio, della realizzazione della mutua esclusione, si può avere una situazione in cui due processi sono in esecuzione contemporaneamente nelle rispettive sezioni critiche.

19.5.1.3 Ripristino dopo un guasto

Quando un collegamento o un sito guasto è stato riparato, deve essere reintegrato nel sistema in modo semplice e lineare.

- Si supponga che si sia guastato un collegamento tra *A* e *B*. Una volta riparato il guasto, è necessario informare della riparazione sia *A* sia *B*. Quest'informazione può essere data ripetendo continuamente la procedura di heartbeat descritta nel Paragrafo 19.5.1.1.
- Si supponga che si sia guastato il sito *B*. Questo, una volta ripristinato, deve informare tutti gli altri siti che è di nuovo attivo. Il sito *B* può quindi ricevere informazioni da altri siti per aggiornare le sue tabelle locali; per esempio, possono servirgli le informazioni contenute nella tabella di routing, un elenco di siti fuori servizio, messaggi e posta non consegnati, oppure un log delle transazioni contenente le transazioni non eseguite. Se il sito non era guasto, ma era semplicemente irraggiungibile, queste informazioni devono comunque essergli inviate.

19.5.2 Trasparenza

Rendere **trasparenti** agli utenti i numerosi processori e dispositivi di archiviazione presenti in un sistema distribuito è stato lo scopo di molti progettisti. Idealmente, un sistema distribuito dovrebbe apparire ai suoi utenti come un sistema centralizzato convenzionale. L'interfaccia utente di un sistema distribuito trasparente non deve fare distinzione tra risorse locali e remote. In altre parole, gli utenti devono essere in grado di accedere a risorse remote come se queste risorse fossero locali e spetta al sistema distribuito localizzare le risorse e organizzare le corrette interazioni.

Un altro aspetto della trasparenza riguarda la mobilità dell'utente. È conveniente consentire agli utenti di accedere a qualsiasi macchina nel sistema, piuttosto che costringerli a utilizzare una macchina specifica. Un sistema distribuito trasparente facilita la mobilità degli utenti trasferendo il loro ambiente (per esempio, la home directory) ovunque si colleghino. Protocolli come ldap forniscono un sistema di autenticazione per utenti locali, remoti e mobili. Una volta che l'utente si è autenticato, funzionalità come la virtualizzazione del desktop gli consentono di operare sulle proprie sessioni di lavoro da remoto.

19.5.3 Scalabilità

Un altro problema è la **scalabilità**, ovvero la capacità di un sistema di adattarsi a un maggior carico di servizio. I sistemi hanno risorse limitate e possono saturarle completamente quando il carico aumenta. Per esempio, per quanto riguarda un file system, si verifica una saturazione delle risorse sia quando la cpu di un server ha un tasso di utilizzo elevato, sia quando le richieste di accesso ai dischi sovraccaricano il sottosistema di i/o. La scalabilità è una caratteristica relativa, ma può essere misurata con precisione. Un sistema scalabile reagisce più gradualmente a un carico maggiore rispetto a un sistema non scalabile. In primo luogo, le sue prestazioni si degradano più moderatamente; in secondo luogo, le sue risorse si saturano più tardi. Tuttavia, anche un sistema perfettamente progettato non è in grado di contenere un carico sempre crescente. L'aggiunta di nuove risorse può risolvere il problema, ma può anche generare ulteriore carico indiretto su altre risorse (per esempio, l'aggiunta di macchine a un sistema distribuito può intasare la rete e aumentare i carichi di servizio). Ancor peggio, l'espansione del sistema può richiedere costose modifiche strutturali. Un sistema scalabile deve avere la potenzialità di crescere evitando questi problemi. In un sistema distribuito la capacità di scalare in maniera dolce è di particolare importanza, poiché l'espansione di una rete con l'aggiunta di nuove macchine o l'interconnessione di due reti sono pratiche comuni. In breve, un progetto scalabile deve resistere a un alto carico di servizio, favorire la crescita della comunità degli utenti e permettere una semplice integrazione di risorse aggiuntive.

La scalabilità è legata alla tolleranza ai guasti, discussa in precedenza. Un componente estremamente carico può paralizzarsi e comportarsi come un componente guasto. Inoltre, lo spostamento del carico da un componente guasto a quello di riserva può saturare anche quest'ultimo. In generale, disporre di risorse di riserva è essenziale per garantire l'affidabilità e gestire con gradualità i picchi di carico. Pertanto, le risorse replicate in un sistema distribuito rappresentano un vantaggio intrinseco, perché danno al sistema maggiori potenzialità di tolleranza ai guasti e scalabilità. Tuttavia, una progettazione inappropriata può occultare queste potenzialità. Le considerazioni relative alla tolleranza ai guasti e alla scalabilità richiedono una progettazione caratterizzata dalla distribuzione del controllo e dei dati.

La scalabilità può anche essere legata a schemi di archiviazione efficienti. Per esempio, molti provider di cloud storage utilizzano la compressione o la deduplicazione per ridurre la quantità di memoria utilizzata. La compressione riduce la dimensione di un file. Per esempio, un file di archivio `zip` può essere generato a partire da uno o più file lanciando un comando `zip` che esegue un algoritmo di compressione senza perdita di dati sui file specificati (la *compressione senza perdita di dati* consente di ricostruire perfettamente i dati originali dai dati compressi): il risultato è un file `zip` più piccolo del file non compresso. Per ripristinare il file originale, un utente esegue un comando di decompressione sul file `zip`. La deduplicazione cerca di ridurre le richieste di spazio di archiviazione rimuovendo i dati ridondanti. Con questa tecnologia solo un'istanza dei dati viene archiviata in un intero sistema (anche tra i dati di proprietà di più utenti). Compressione e deduplicazione possono essere eseguite a livello di file o a livello di blocco e possono essere utilizzate insieme. Queste tecniche sono integrabili automaticamente in un sistema distribuito per comprimere le informazioni senza che gli utenti impartiscano comandi esplicitamente, risparmiando così spazio di archiviazione e riducendo i costi di comunicazione di rete senza aggiungere complessità all'utente.

19.6 File system distribuiti

Sebbene il World Wide Web sia il sistema distribuito decisamente più diffuso, non è l'unico. Un altro importante esempio elaborazione distribuita è dato dai file system distribuiti, o dfs.

Per descrivere la struttura di un dfs occorre definire innanzitutto i termini *servizio*, *server* e *client* nel contesto dei file system distribuiti. Il servizio è il software in esecuzione in una o più macchine che fornisce un tipo particolare di funzione a client sconosciuti a priori; il server è il programma di servizio in esecuzione in una singola macchina; il client è un processo che può richiedere un servizio servendosi di una serie di operazioni che costituiscono la sua interfaccia, detta interfaccia del client. Talvolta si definisce un'interfaccia di livello inferiore per l'effettiva interazione tra le macchine; si tratta dell'interfaccia intermacchina.

Secondo questa terminologia, diciamo che un file system fornisce servizi relativi ai file ai suoi client. Un'interfaccia del client per un servizio relativo ai file è composta da un insieme di operazioni primitive su file, come creazione di un file, cancellazione, lettura e scrittura. Il principale elemento fisico controllato da un file server è composto da un insieme di dispositivi locali di memoria secondaria, generalmente dischi magnetici, nei quali si memorizzano i file e dai quali si recuperano gli stessi secondo le richieste dei client.

Un dfs è un file system i cui client, server e dispositivi di memorizzazione sono sparsi tra le macchine di un sistema distribuito. Di conseguenza l'attività di servizio si deve eseguire attraverso la rete e, anziché un unico sito di memorizzazione dei dati centralizzato, il sistema ha più dispositivi di memorizzazione indipendenti. La configurazione e la realizzazione concrete di un dfs possono essere di vario tipo. In alcune configurazioni i server sono eseguiti su macchine specifiche, in altre una macchina può essere sia un server sia un client.

Le caratteristiche che contraddistinguono un dfs sono la molteplicità e l'autonomia dei client e dei server del sistema. Tuttavia un dfs deve, idealmente, apparire ai client come un file system centralizzato convenzionale. Ciò significa che l'interfaccia del client di un dfs non deve distinguere tra file locali e file remoti. Spetta al dfs localizzare i file e predisporre il trasporto dei dati. Un dfs trasparente, come i sistemi distribuiti trasparenti menzionati in precedenza, facilita la mobilità dell'utente portando il suo ambiente, cioè la directory iniziale, ovunque egli apra una sessione.

La misura delle prestazioni più importante per un dfs è rappresentata dal tempo necessario a soddisfare le richieste di servizio. Nei sistemi convenzionali questo tempo è dato dal tempo d'accesso al disco e da una piccola quantità di tempo di cpu. In un dfs, invece, un accesso remoto risente anche di un ulteriore carico dovuto alla struttura distribuita. In questo overhead rientrano il tempo necessario per inviare la richiesta a un server e il tempo necessario per ricevere la risposta. Pertanto per ogni direzione occorre considerare, oltre al trasferimento dell'informazione, anche il carico della cpu dovuto all'esecuzione del protocollo di comunicazione. Le prestazioni di un dfs si possono considerare come un'altra misura della sua trasparenza. Ciò significa che le prestazioni di un dfs ideale devono essere paragonabili a quelle di un file system convenzionale.

L'architettura di base di un dfs dipende dai suoi obiettivi finali. Due modelli architettonici ampiamente utilizzati che trattiamo nel seguito sono il modello client-server e il modello basato su cluster. L'obiettivo principale di un'architettura client-server è di consentire la condivisione trasparente dei file tra uno o più client, come se i file fossero archiviati localmente sui singoli client. I file system distribuiti nfs e Openafs sono due importanti esempi. nfs è il dfs basato su unix più utilizzato. nfs ha diverse versioni, e noi faremo riferimento alla versione 3, se non diversamente specificato.

Quando molte applicazioni devono essere eseguite in parallelo su data set di grandi dimensioni con elevata disponibilità e scalabilità, il modello basato su cluster è più indicato rispetto al modello client-server. Due esempi ben noti sono il Google file system e l'open-source hdfs, che è parte del framework Hadoop.

19.6.1 Il modello client-server di DFS

La Figura 19.12 illustra un semplice modello client-server di dfs. Il server memorizza sia i file sia i metadati sul suo dispositivo di archiviazione. In alcuni sistemi è possibile utilizzare più di un server per memorizzare file diversi. I client sono connessi al server per mezzo di una rete e possono richiedere l'accesso ai file nel dfs contattando il server tramite un protocollo noto, per esempio nfs versione 3. Il server esegue l'autenticazione, verifica le autorizzazioni richieste per l'accesso al file e, se permesso, consegna il file al client che ne ha fatto richiesta. Quando un client apporta modifiche al file deve in qualche modo consegnare le modifiche al server (che conserva la copia originale del file). Le versioni del file del client e del server devono essere mantenute coerenti, con un metodo che riduca al minimo il traffico di rete e il carico di lavoro del server.

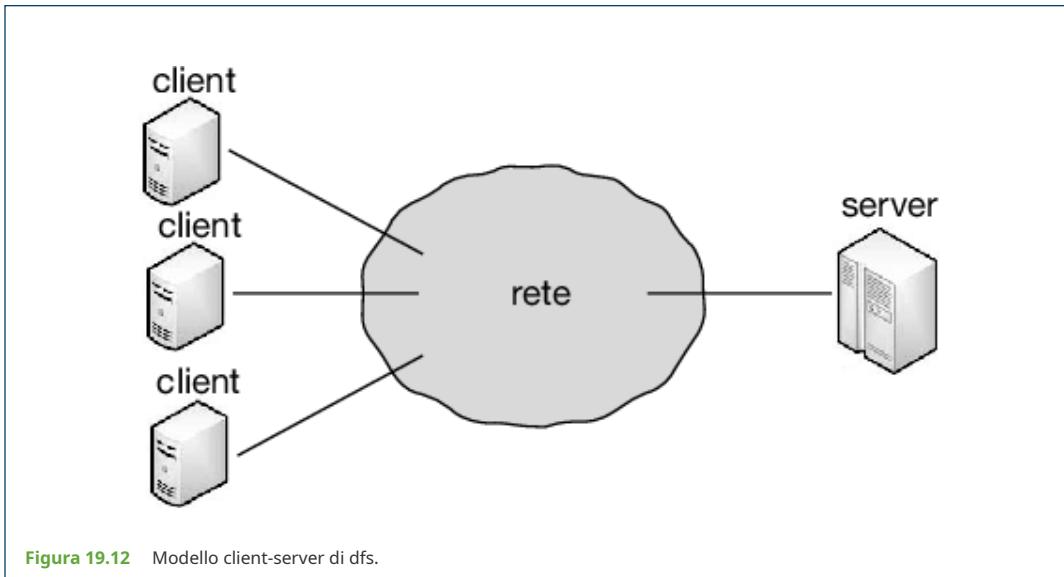


Figura 19.12 Modello client-server di dfs.

Il protocollo nfs (*network file system*) è stato originariamente sviluppato da Sun Microsystems come un protocollo aperto e questo fattore ne ha incoraggiato la pronta adozione su architetture e sistemi diversi. Fin dall'inizio, nfs è stato focalizzato sul ripristino rapido e semplice degli arresti anomali del server. Per implementare questa funzionalità, il server nfs è stato progettato per essere privo di informazioni di stato: non tiene traccia di quale client sta accedendo a quale file o di elementi come i descrittori di file aperti e i puntatori ai file. Ciò significa che ogni volta che un client effettua un'operazione su file (per esempio, una lettura), questa operazione deve essere *idempotente* di fronte a un crash del server, ovvero deve poter essere eseguita più di una volta restituendo sempre lo stesso risultato. Nel caso di un'operazione di lettura, il client tiene traccia dello stato (per esempio, mantiene il puntatore al file) e può semplicemente ripetere la richiesta quando il server si arresta in modo anomalo e successivamente ritorna on-line. Ulteriori informazioni sull'implementazione di nfs si trovano nel Paragrafo 15.8.

Il file system Andrew (Openafs) è stato creato presso la Carnegie Mellon University con particolare attenzione alla scalabilità. Nello specifico, i ricercatori hanno voluto progettare un protocollo che consentisse al server di supportare quanti più client possibile. Per farlo, è stato necessario ridurre le richieste e il traffico verso il server. Quando un client richiede un file, i contenuti del file vengono scaricati dal server e memorizzati nella memoria locale del client. Gli aggiornamenti al file vengono inviati al server quando il file viene chiuso e le nuove versioni del file vengono inviate al client quando il file viene aperto. In confronto, nfs scambia più informazioni, continuando a inviare al server richieste di lettura e scrittura di blocchi mentre il file viene utilizzato dal client.

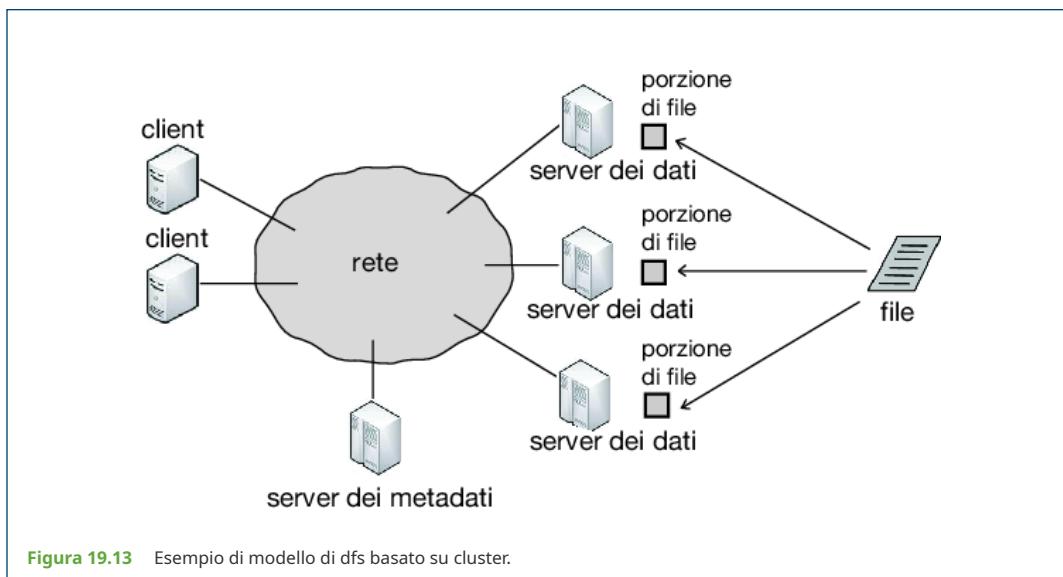
Sia Openafs sia nfs sono pensati per essere utilizzati in aggiunta ai file system locali. In altre parole, non si formatta una partizione del disco rigido con il file system nfs. Sul server occorre formattare la partizione con un file system locale di propria scelta, come ext4, ed esportare le directory condivise tramite il dfs. Nel client è sufficiente collegare le directory esportate alla struttura del file system locale. In questo modo, il dfs non è responsabile del file system locale e può concentrarsi sulle attività distribuite.

Il modello client-server di dfs, per come è progettato, può soffrire di un singolo punto di guasto in caso di arresto anomalo del server. Il clustering può aiutare a risolvere questo problema utilizzando componenti ridondanti e metodi di clustering per fare in modo che i guasti vengano rilevati e le operazioni del server vengano passate alle componenti di riserva funzionanti. Inoltre, il server rappresenta un collo di bottiglia per tutte le richieste sia di dati sia di metadati, con conseguenti problemi di scalabilità e larghezza di banda.

19.6.2 Il modello DFS basato su cluster

All'aumentare della quantità di dati, del carico di i/o e delle attività computazionali aumenta anche la necessità che un dfs sia tollerante ai guasti e scalabile. Non è possibile tollerare grandi colli di bottiglia e ci si deve aspettare che i componenti del sistema si guastino. L'architettura basata su cluster è stata sviluppata in parte per soddisfare queste esigenze.

La Figura 19.13 illustra un esempio di modello di dfs basato su cluster. Si tratta del modello di base utilizzato dal Google file system (gfs) e dal file system distribuito Hadoop (hdfs). Uno o più client sono collegati tramite una rete a un server principale di metadati e a diversi server di dati che ospitano "chunk", o porzioni, di file. Il server dei metadati mantiene una mappa delle porzioni dei file e dei server di dati che le contengono, oltre a una tradizionale mappa gerarchica di directory e file. Ogni porzione di file viene archiviata su un server di dati e replicata un certo numero di volte (per esempio tre volte) per offrire protezione dai guasti dei componenti e per un accesso più rapido ai dati (i server contenenti le porzioni replicate hanno accesso rapido a tali porzioni).



Per ottenere l'accesso a un file, un client deve prima contattare il server dei metadati, che restituisce al client le identità dei server di dati che contengono le porzioni del file richiesto. Il client può quindi contattare il server (o i server) di dati più vicino per ricevere le informazioni del file. È possibile leggere o scrivere in parallelo diverse porzioni di file se queste sono memorizzate su server dati distinti ed è talvolta sufficiente contattare il server dei metadati una sola volta nell'intero processo. Ciò rende meno probabile che il server dei metadati diventi un collo di bottiglia per le prestazioni. Il server dei metadati è anche responsabile della ridistribuzione e del bilanciamento delle porzioni di file tra i server dati.

gfs è stato rilasciato nel 2003 per supportare applicazioni che utilizzino grandi quantità di dati distribuiti. Il progetto di gfs è stato influenzato dalle seguenti quattro osservazioni principali.

- I guasti dei componenti hardware sono la norma piuttosto che l'eccezione e devono essere trattati come ordinaria amministrazione.
- I file memorizzati su un sistema di questo tipo sono molto grandi.
- La maggior parte dei file viene modificata aggiungendo nuovi dati in coda al file, anziché sovrascrivendo i dati esistenti.
- La riprogettazione delle applicazioni e dell'api del file system aumenta la flessibilità del sistema.

Coerentemente con la quarta osservazione, gfs esporta la propria api e richiede che le applicazioni siano programmate con essa.

Poco dopo aver sviluppato gfs, Google ha implementato uno strato software modulare chiamato MapReduce da collocare immediatamente sopra a gfs per consentire agli sviluppatori di eseguire più facilmente calcoli paralleli su larga scala, sfruttando i vantaggi del file system al livello inferiore. Successivamente, sulla base del lavoro di Google, sono stati creati hdfs e il framework Hadoop (che include moduli che si appoggiano su hdfs, come MapReduce). Come gfs e MapReduce, Hadoop supporta l'elaborazione di grandi data set in ambienti di calcolo distribuito. Come suggerito in precedenza, questo sviluppo è stato necessario perché i sistemi tradizionali non potevano crescere fino alla capacità e alle prestazioni richieste dai progetti di "big data" (o almeno non a prezzi ragionevoli). Esempi di progetti di big data includono la scansione e l'analisi dei social media, dei dati dei clienti e di grandi quantità di dati scientifici al fine di studiarne le tendenze.

19.7 Naming e trasparenza nei DFS

Il naming è una funzione di associazione tra oggetti logici e oggetti fisici. Per esempio, gli utenti trattano oggetti logici contenenti dati, rappresentati dai nomi dei file, mentre il sistema manipola blocchi fisici di dati, memorizzati sulle tracce di un disco. Generalmente l'utente fa riferimento a un file attraverso un nome testuale. A quest'ultimo si fa corrispondere un identificatore numerico di livello inferiore che a sua volta si fa corrispondere ai blocchi dei dischi. Quest'associazione a più livelli fornisce agli utenti un'astrazione del file che nasconde i particolari concernenti la sua memorizzazione.

In un dfs trasparente, all'astrazione si aggiunge un nuovo aspetto: nascondere la posizione all'interno della rete in cui è posizionato il file. In un file system convenzionale il risultato della funzione di naming è un indirizzo in un disco. In un dfs si estende l'insieme dei valori risultanti in modo da comprendere anche la specifica macchina nel cui disco è contenuto il file. Un passo successivo nel trattare i file come oggetti astratti, porta alla possibilità di replicazione dei file. Dato il nome di un file, il sistema di associazione riporta una serie di locazioni delle repliche di questo file. In quest'astrazione sono nascoste sia l'esistenza di copie multiple sia la loro locazione.

19.7.1 Strutture di naming

Occorre distinguere due nozioni correlate riguardo al sistema di associazione dei nomi di un dfs.

1. Trasparenza rispetto alla locazione. Il nome di un file non rivela alcun indizio sulla sua locazione fisica.
2. Indipendenza dalla locazione. Non si deve modificare il nome di un file se cambia la sua locazione di memoria fisica.

Entrambe le definizioni si riferiscono al livello di naming esaminato precedentemente, poiché i file hanno nomi diversi a livelli diversi: a livello utente si usano nomi testuali, mentre a livello del sistema si usano gli identificatori numerici. Uno schema di naming indipendente dalla locazione è un'associazione dinamica poiché può far corrispondere locazioni diverse in due momenti diversi allo stesso nome di file. Perciò l'indipendenza dalla locazione è una caratteristica più forte di quanto non lo sia la trasparenza rispetto alla locazione.

In pratica la maggior parte degli attuali dfs offre un'associazione statica con trasparenza di locazione per i nomi a livello utente. Alcuni dfs supportano la migrazione dei file, cioè il cambio automatico di posizione di un file, garantendo così l'indipendenza dalla locazione. Openafs, per esempio, supporta l'indipendenza dalla locazione e la mobilità dei file. Il file system distribuito Hadoop (hdfs), uno speciale file system progettato recentemente per il framework Hadoop, supporta la migrazione dei file, ma lo fa senza seguire gli standard posix, fornendo una maggiore flessibilità nell'implementazione e nell'interfaccia. hdfs tiene traccia della posizione dei dati, ma nasconde questa informazione ai client. Questa strategia permette al meccanismo sottostante un tuning automatico. Un altro esempio è l'infrastruttura di cloud storage di Amazon, che fornisce blocchi di storage on demand per mezzo di api, mettendo lo storage dove lo ritiene opportuno e spostando i dati secondo le necessità, al fine di migliorare le prestazioni, l'affidabilità e le esigenze di capacità.

L'indipendenza dalla locazione e la trasparenza di locazione statica si differenziano ancora per altri aspetti.

- La separazione dei dati dalla locazione, come nel caso dell'indipendenza dalla locazione, offre una migliore astrazione per i file. Il nome di un file deve indicare gli attributi più significativi del file stesso, legati al suo contenuto, piuttosto che la sua locazione. I file indipendenti dalla locazione si possono considerare come contenitori di dati logici che non si assegnano a una specifica locazione di memoria secondaria. Se è prevista soltanto la trasparenza di locazione statica, il nome del file continua a indicare un gruppo specifico, anche se nascosto, di blocchi fisici nei dischi.
- La trasparenza di locazione statica fornisce agli utenti un modo per condividere convenientemente i dati. Gli utenti possono condividere file remoti semplicemente assegnandogli nomi in maniera trasparente rispetto alla locazione, esattamente come se questi fossero locali. Dropbox e altre soluzioni di cloud storage lavorano in questo modo. L'indipendenza dalla locazione consente la condivisione dello spazio di memorizzazione stesso e degli oggetti di dati. Quando i file si possono rendere mobili, lo spazio di memorizzazione dell'intero sistema assume l'aspetto di una singola risorsa virtuale. Uno dei vantaggi che ne conseguono è la possibilità di bilanciare l'utilizzo dei dischi all'interno del sistema.
- L'indipendenza dalla locazione separa la gerarchia di naming dalla gerarchia dei dispositivi di memorizzazione e dalla suddivisione tra i calcolatori. Al contrario, usando la trasparenza di locazione statica, si può facilmente evidenziare la corrispondenza tra unità componenti e macchine, anche se i nomi sono trasparenti. Le macchine sono configurate impiegando un modello simile alla struttura di naming; questa configurazione può forzare l'architettura del sistema a rispettare vincoli non necessari, ed è anche in conflitto con altre considerazioni. Un server che gestisca una directory radice è un esempio di struttura dettata dalla gerarchia di naming e contraddice le direttive di decentramento.

Una volta completata la separazione tra nome e locazione, i client possono accedere a file che risiedono in sistemi server remoti. Infatti questi client possono essere *privi di dischi* e adoperare un server per accedere a tutti i file, compreso il kernel del sistema operativo. Per la sequenza d'avviamento sono però necessari protocolli speciali; si consideri il problema di caricare il kernel su una stazione di lavoro priva di dischi. Poiché tale stazione non ha il kernel, non può usare il codice del dfs per recuperarlo. Invece s'impiega uno speciale protocollo d'avviamento, contenuto in memoria di sola lettura (rom) del client, che abilita la connessione alla rete e recupera solo un file speciale, il kernel o il codice d'avviamento da una locazione fissa. Una volta che il kernel è stato copiato attraverso la rete e caricato, il suo dfs rende disponibili tutti gli altri file del sistema operativo. I vantaggi dati da client privi di dischi sono molti, tra i quali il minor costo (le macchine sono appunto private di dischi) e una maggior facilità di gestione (quando si aggiorna un sistema operativo, occorre modificare solo il server anziché tutti i client). Lo svantaggio invece è dato dalla maggiore complessità dei protocolli d'avviamento e dal calo delle prestazioni dovuto all'uso di una rete anziché di un disco locale.

19.7.2 Schemi di naming

In un dfs sono possibili tre metodi principali per gli schemi di naming. Il metodo più semplice prevede di nominare i file per mezzo di una combinazione del loro nome di macchina e nome locale; ciò garantisce un unico nome su tutto il sistema. Nel sistema Ibis, per esempio, un file è identificato unicamente dal nome *macchina:nome-locale*, dove nome-locale indica un percorso di tipo unix. Anche gli url di Internet seguono questo approccio. Questo schema di naming non gode della trasparenza di locazione e tanto meno dell'indipendenza dalla locazione. La struttura del dfs è rappresentata da un insieme di unità componenti isolate costituite da interi

file system convenzionali. Le unità componenti rimangono isolate, anche se sono disponibili mezzi per fare riferimento ai file remoti. Non proseguiremo oltre la descrizione di questo schema.

Il secondo metodo si è diffuso con il network file system di Sun (nfs). nfs si trova in molti sistemi, tra cui unix e Linux e offre i mezzi per unire le directory remote alle directory locali, dando all'utente l'impressione di un albero di directory coerente. Nelle prime versioni era possibile accedere in modo trasparente solo a directory remote montate in precedenza. Con l'avvento della funzione di autmontaggio i montaggi si eseguono su richiesta secondo una tabella di punti di montaggio e nomi di strutture di file. I componenti sono integrati per gestire la condivisione trasparente, sebbene tale integrazione sia limitata e non uniforme, poiché ogni macchina può aggiungere diverse directory remote al proprio albero. La struttura risultante è flessibile.

L'integrazione totale dei componenti del file system si ottiene per mezzo del terzo metodo. Una sola struttura globale di nomi si estende a tutti i file del sistema. Idealmente, la struttura composta del file system è isomorfa alla struttura di un file system convenzionale. In pratica, tuttavia, esistono molti file speciali (per esempio i file che rappresentano i dispositivi e le directory dei file eseguibili della specifica macchina) che rendono difficile il conseguimento di questo scopo.

Per valutare le strutture di mappaggio ne consideriamo la complessità amministrativa. La struttura più complessa e più difficile da mantenere è la struttura dell'nfs. Poiché ogni directory remota può essere montata in qualsiasi punto dell'albero di una directory locale, la gerarchia risultante può essere quasi priva di struttura. Se un server diventa non disponibile, diventa non disponibile anche un insieme arbitrario di directory di diverse macchine. Inoltre, un meccanismo separato di accreditamento controlla quale macchina può aggiungere una determinata directory al suo albero. Così un utente può ottenere l'accesso a un albero di directory remoto in un client, ma vederselo negare in un altro.

19.7.3 Realizzazione

La realizzazione del naming trasparente richiede un meccanismo per far corrispondere i nomi alle locazioni dei file. Per mantenere gestibile tale corrispondenza occorre aggregare insiemi di file in unità componenti e fornire una corrispondenza secondo l'unità componente anziché secondo il singolo file. Questa aggregazione è utile anche per scopi amministrativi. I sistemi del tipo unix impiegano l'albero gerarchico delle directory per fornire la corrispondenza tra nomi e locazioni e per aggregare i file nelle directory in modo ricorsivo.

Per aumentare la disponibilità delle informazioni fondamentali su tali associazioni si possono usare metodi come replicazione, uso di cache locali, o entrambi. Come si è detto, l'indipendenza dalla locazione implica che l'associazione cambi con il tempo; quindi, replicando l'associazione, diventa impossibile ottenere un aggiornamento semplice e coerente di queste informazioni. Per superare quest'ostacolo si può adoperare una tecnica che introduce identificatori di file di basso livello indipendenti dalla locazione (Openafs utilizza questo approccio). Ai nomi dei file si fanno corrispondere identificatori di file di basso livello, che indicano a quale unità componente appartiene il file; tali identificatori sono ancora indipendenti dalla locazione. Si possono replicare liberamente e copiare in una cache, senza che siano invalidati dalla migrazione delle unità componenti. L'inevitabile prezzo di tale tecnica è un secondo livello di associazione che metta in corrispondenza le unità componenti con le locazioni e richiede un meccanismo di aggiornamento semplice e coerente. Realizzare alberi di directory di tipo unix impiegando questi identificatori di basso livello indipendenti dalla locazione, rende invariante l'intera gerarchia nel caso della migrazione di unità componenti. L'unica variazione riguarda l'associazione delle locazioni delle unità componenti.

Un metodo diffuso per realizzare questi identificatori di basso livello consiste nell'uso di nomi strutturati. Questi nomi sono composti da sequenze di bit formate normalmente da due parti: la prima identifica l'unità componente a cui appartiene il file; la seconda identifica il file all'interno dell'unità. Sono possibili varianti con più parti. L'aspetto invariante dei nomi strutturati, tuttavia, è che ciascuna parte del nome è unica in ogni momento soltanto nel contesto delle parti restanti. L'unicità in ogni momento si può ottenere avendo cura di non riutilizzare un nome già usato, oppure aggiungendo ulteriori bit (metodo utilizzato in Openafs) o impiegando una marca temporale (timestamp) come se fosse una parte del nome (come accadeva nell'Apollo Domain). Un altro modo di vedere questo processo è dire che si prende un sistema con trasparenza di locazione, come l'Ibis, e vi si aggiunge un altro livello di astrazione per produrre uno schema di naming con indipendenza dalla locazione.

19.8 Accesso ai file remoti

Si consideri un utente che richieda l'accesso a un file remoto. Supponendo che il server che contiene il file sia stato localizzato dallo schema di naming, ora è necessario compiere l'effettivo trasferimento dei dati.

Tale trasferimento si può ottenere con il meccanismo del servizio remoto, in cui le richieste d'accesso sono inviate al server che esegue gli accessi e ritrasmette i risultati all'utente. Uno dei modi più diffusi per realizzare il servizio remoto è il paradigma della chiamata di procedura remota (rpc), descritto nel Capitolo 3. Esiste un'analogia diretta tra il metodo d'accesso al disco nei file system convenzionali e il metodo del servizio remoto in un dfs: l'uso del servizio remoto è analogo al compiere un accesso al disco per ciascuna richiesta d'accesso.

Per assicurare prestazioni ragionevoli del meccanismo di servizio remoto si usa un meccanismo di caching. Nei file system convenzionali lo scopo delle cache consiste nel ridurre gli accessi ai dischi, aumentando così le prestazioni; nei dfs lo scopo è quello di ridurre sia il traffico nella rete sia gli accessi ai dischi. In seguito si considera la gestione delle cache in un dfs, a confronto del paradigma del servizio remoto di base.

19.8.1 Caching di base

Il concetto di caching è semplice: se i dati necessari a soddisfare la richiesta d'accesso non sono già stati copiati nella cache, si trasferisce una copia di quei dati dal server al sistema client. Gli accessi richiesti si eseguono sulla copia in cache. L'idea è di tenere nella cache blocchi di disco a cui si è fatto riferimento di recente, gestendo localmente gli accessi ripetuti alle stesse informazioni, e riducendo il traffico di rete. Un criterio di sostituzione (per esempio lru) limita la dimensione della cache. Non esiste una corrispondenza diretta fra gli accessi e il traffico verso il server: i file sono ancora identificati con una copia principale che risiede nella macchina server, ma altre copie del file o di parti di esso sono sparse in diverse cache. Se si modifica una copia contenuta in una cache, le modifiche vanno riportate anche sulla copia principale, in modo da conservare la coerenza dei dati. Il problema di mantenere le copie coerenti con il file principale è detto problema di coerenza della cache (Paragrafo 19.8.4). L'impiego delle cache in un dfs si può chiamare semplicemente memoria virtuale di rete: infatti funziona in modo analogo alla memoria virtuale con paginazione su richiesta, con la differenza che in questo caso la memoria ausiliaria non è un disco locale, ma un server remoto. nfs permette il montaggio a distanza dell'area di swap: può quindi effettivamente implementare la memoria virtuale sulla rete, anche se a scapito di una calo nelle prestazioni.

Nel caso di un dfs, la dimensione dei dati da copiare nelle cache può variare dai blocchi di file ai file interi. In genere si copiano più dati di quanti non siano necessari a soddisfare un singolo accesso, in modo da aumentare le probabilità di soddisfare più accessi a tali dati. Questo procedimento è molto simile alla lettura anticipata dei dati dai dischi (Paragrafo 14.6.2). Openafs copia i file in grandi sezioni (64 kb); gli altri sistemi che abbiamo citato copiano blocchi singoli in modo controllato dalle richieste dei client. Aumentando la quantità unitaria di dati copiati, aumenta il tasso di successi, ma aumenta anche il costo di ogni insuccesso nella ricerca nella cache, poiché ogni insuccesso richiede il trasferimento di una maggiore quantità di dati. Inoltre aumentano le possibilità dei problemi di coerenza. Nella scelta della quantità di dati unitaria da copiare nelle cache occorre considerare parametri come l'unità di trasferimento della rete e l'unità di servizio del protocollo delle rpc (nel caso in cui sia usato un protocollo rpc). L'unità di trasferimento della rete (in Ethernet è un pacchetto) è di circa 1,5 kb, perciò le unità di dati più grandi devono essere scomposte prima di essere inviate e quindi ricomposte al momento della ricezione.

Naturalmente la dimensione dei blocchi e la dimensione totale della memoria cache sono importanti per gli schemi di gestione delle cache a blocchi. Nei sistemi di tipo unix le dimensioni normali dei blocchi sono di 4 kb o 8 kb. Per cache di grandi dimensioni (oltre 1 mb) convengono blocchi più grandi (oltre 8 kb); per cache più piccole, i blocchi di grandi dimensioni sono meno utili, poiché implicano un numero inferiore di blocchi contenuto nelle cache, e un tasso di successi inferiore.

19.8.2 Locazione delle cache

Ci si può chiedere se i dati della cache vadano memorizzati su disco o in memoria centrale. Le cache su dischi hanno un evidente vantaggio rispetto alle cache in memoria centrale: sono affidabili. Se la cache è nella memoria volatile, le modifiche ai dati in essa contenuti si perdono nel caso di una caduta del sistema. Inoltre, se la cache è mantenuta in memoria secondaria, i dati continuano a trovarvisi anche durante il ripristino, e non è necessario prelevarli di nuovo. D'altra parte le cache in memoria centrale hanno parecchi specifici vantaggi.

- Le cache in memoria centrale permettono l'uso di stazioni di lavoro prive di dischi.
- È possibile accedere più rapidamente ai dati in una cache in memoria centrale, che ai dati in una cache su disco.
- Le attuali tecnologie tendono a produrre memorie più grandi e meno costose. Si prevede che l'incremento delle prestazioni supererà in importanza i vantaggi offerti dalle cache su dischi.
- Le cache dei server, impiegate per accelerare l'i/o dei dischi, si trovano in memoria centrale a prescindere dalla locazione delle cache degli utenti; se si usano le cache in memoria centrale anche per le macchine degli utenti, si può realizzare un unico meccanismo di gestione utilizzabile sia dai server sia dagli utenti.

Molti sistemi d'accesso remoto si possono considerare come un ibrido fra l'uso di cache e servizio remoto. Nell' nfs, per esempio, l'implementazione è basata sul servizio remoto, ma è affiancata dall'uso di cache in memoria sia da parte del client sia da parte del server per incrementare le prestazioni. Pertanto, per valutare i due metodi è necessario valutare il grado in cui uno dei due è enfatizzato. Il protocollo nfs e la maggior parte delle implementazioni non forniscono il caching del disco (ma Openafs lo fa).

19.8.3 Criteri di aggiornamento delle cache

La politica che si usa per riscrivere blocchi di dati modificati nella copia principale sul server ha un effetto critico sulle prestazioni e sull'affidabilità del sistema. Il criterio più semplice consiste nello scrivere direttamente i dati nei dischi non appena questi vengono modificati in una cache qualsiasi. Il vantaggio della scrittura immediata (*write-through policy*) è l'affidabilità; se un sistema client si

guasta, si perdono solo poche informazioni. Tuttavia questo criterio richiede che ogni accesso per scrittura attenda l'invio delle informazioni al server, il che implica scarse prestazioni delle operazioni di scrittura. L'uso di cache write-through corrisponde all'uso del servizio remoto per le scritture e allo sfruttamento delle cache per le sole letture.

Un'alternativa è il criterio di scrittura differita (o *write-back caching*), che consiste nel differire gli aggiornamenti della copia principale. Le modifiche si scrivono nella cache e più tardi nel server. Questo criterio ha due vantaggi rispetto alla scrittura diretta: innanzitutto, poiché le scritture sono effettuate sulla cache, gli accessi per scrittura sono molto più rapidi; in secondo luogo, si possono sovrascrivere i dati prima che siano riportati al server, in tal caso è necessario riportare solo l'ultimo aggiornamento. Sfortunatamente gli schemi di scrittura differita causano problemi di affidabilità, poiché se una macchina utente subisce un guasto si perdono i dati non scritti.

Le varianti del criterio della scrittura differita si differenziano secondo il momento in cui i blocchi di dati modificati sono inviati al server. Una possibilità è quella che prevede l'invio di un blocco quando questo sta per essere espulso dalla cache del client. Ciò può avere esiti positivi sulle prestazioni, ma può accadere che alcuni blocchi risiedano per un lungo periodo nella cache del client prima di essere riscritti nel server. Un compromesso tra questo metodo e la scrittura immediata consiste nell'esaminare la cache a intervalli regolari e inviare al server solo i blocchi modificati dopo l'ultima verifica, esattamente come in unix si scandisce la cache locale. Nel sistema Sprite s'impiega questo criterio con un intervallo di verifica di 30 secondi. L' nfs adopera questo criterio per i dati nei file, ma una volta che una scrittura è stata inviata al server durante uno svuotamento della cache, per essere considerata completa, deve giungere al disco del server. L' nfs tratta i metadati (i dati nelle directory e i dati sugli attributi dei file) in modo differente: tutti i cambiamenti dei metadati s'inviano in modo sincrono ai relativi server. In questo modo si evitano le perdite di file e le alterazioni delle strutture delle directory in seguito a un crollo di un client o di un server.

Un'altra variante della scrittura differita consiste nello scrivere i dati nel server quando si chiude un file. Questo criterio, detto di scrittura su chiusura (write-on-close policy), è adottato dal sistema Openafs. Nel caso in cui si aprano i file per periodi brevi oppure si modifichino solo di rado, questo criterio non riduce significativamente il traffico presente in rete. Inoltre il criterio di scrittura su chiusura richiede che il processo in chiusura sia ritardato per permettere la scrittura diretta del file; ciò riduce i vantaggi della scrittura differita. Le migliori prestazioni di questo criterio, rispetto a quelli della scrittura differita con un più frequente invio al server, sono evidenti quando i file rimangono aperti per lunghi periodi e vengono modificati spesso.

19.8.4 Coerenza della cache

Una macchina client deve talvolta affrontare il problema di decidere se una copia in una cache sia o no coerente con la copia principale, e quindi se possa essere usata. Se la macchina client stabilisce che i suoi dati nella cache non sono aggiornati, occorre ottenere una copia aggiornata dei dati e inserirli nella cache prima di consentire ulteriori accessi. Per compiere tale verifica si possono seguire due metodi.

1. Verifica iniziata dal client. Il client inizia un controllo di validità mettendosi in contatto con il server per controllare se i dati locali siano coerenti con la copia principale. La frequenza del controllo di validità è il fulcro di questo metodo e determina la conseguente semantica della coerenza. Si passa da un controllo prima di ogni accesso, fino a scendere a un solo controllo durante il primo accesso a un file (normalmente quando il file viene aperto). Ogni accesso unito a un controllo di validità è ritardato rispetto a un accesso servito immediatamente dalla memoria cache. In alternativa si può iniziare un controllo a intervalli di tempo prefissati. A seconda della frequenza d'esecuzione il controllo di validità può caricare sia la rete sia il server.
2. Verifica iniziata dal server. Il server registra, per ogni client, i file (o le parti di file) che copia nella cache e interviene se individua una potenziale incoerenza. Una situazione di questo tipo si verifica quando due diversi client copiano un file in una cache in modi conflittuali. Se viene implementata la semantica unix (Paragrafo 15.7), si possono risolvere le potenziali incoerenze facendo in modo che il server svolga un ruolo attivo. Il server deve essere informato ogni volta che un file viene aperto, e per ogni apertura deve essere indicata la modalità (lettura o scrittura). Il server può intervenire quando individua un file che viene aperto contemporaneamente in modalità conflittuali, disabilitando la possibilità di copiare quel file nelle cache. Tale disabilitazione significa un passaggio a una modalità di funzionamento di servizio remoto.

In un dfs basato su cluster il problema della coerenza della cache è reso più complicato dalla presenza di un server dei metadati e di numerose porzioni replicate di parti di file distribuite su più server dati. Possiamo osservare alcune differenze tra hdfs e gfs, gli esempi che abbiamo trattato in precedenza. hdfs consente operazioni di scrittura esclusivamente in coda ai file (in append) ed effettuate da un singolo scrittore, mentre gfs consente scritture casuali e concorrenti. In gfs è quindi molto più complicato garantire la coerenza delle scritture, mentre risulta più semplice in hdfs.

19.9 Considerazioni finali sui file system distribuiti

Il confine tra architetture di dfs client-server e architetture basate su cluster è sfocato. La specifica nfs Versione 4.1 include un protocollo per una versione parallela di nfs chiamata pnfs, ma al momento della stesura di questo testo la sua adozione procede a rilento.

gfs, hdfs e altri dfs di grandi dimensioni forniscono un'api non posix e non possono quindi mappare in modo trasparente le directory sulle macchine degli utenti come fanno nfs e Openafs. I sistemi che accedono a questi dfs hanno infatti bisogno di un client installato. Tuttavia, vengono rapidamente sviluppati altri livelli software per consentire a nfs di essere montato su tali dfs. Questa soluzione è molto interessante, poiché trarrebbe vantaggio dalla scalabilità e da altri pregi dei dfs basati su cluster, consentendo al tempo stesso agli utenti e alle funzionalità native del sistema operativo di accedere ai file direttamente su dfs.

A oggi, il gateway hdfs nfs open-source supporta nfs versione 3 e funge da proxy tra hdfs e il software del server nfs. Poiché hdfs non supporta attualmente scritture casuali, nemmeno il gateway le supporta. Ciò significa che un file deve essere cancellato e ricreato da zero anche se viene modificato solo un byte. Le organizzazioni commerciali e i ricercatori stanno affrontando questo problema costruendo framework impilabili che consentono di sovrapporre un dfs, moduli di calcolo parallelo (come MapReduce), database distribuiti e volumi di file esportati tramite nfs.

Un altro tipo di file system, meno complesso di un dfs basato su cluster, ma più complesso di un dfs client-server, è il clustered file system (cfs), o file system parallelo (pfs). Un cfs viene generalmente eseguito su una lan. Questi sistemi sono importanti e ampiamente utilizzati e meritano quindi di essere menzionati qui, anche se non li copriamo in dettaglio. Tra i cfs più diffusi ci sono Lustre e gpfs, ma ne esistono molti altri. Un cfs essenzialmente tratta N sistemi che memorizzano dati e Y sistemi che accedono a tali dati come un'unica istanza client-server. Mentre nfs, per esempio, ha un naming per ogni server, e due server nfs distinti forniscono generalmente due schemi di naming diversi, un cfs inserisce vari contenuti di archiviazione presenti su diversi dispositivi di memorizzazione su server distinti in un unico spazio di nomi uniforme e trasparente. gpfs ha una propria struttura di file system, mentre Lustre utilizza file system esistenti, come zfs per l'archiviazione e la gestione dei file. Per saperne di più, si veda <http://lustre.org>.

I file system distribuiti sono oggi di uso comune e consentono la condivisione di file all'interno di lan, all'interno di ambienti cluster attraverso le wan. La complessità di implementare un tale sistema non va sottovalutata, soprattutto considerando che il dfs deve essere indipendente dal sistema operativo per permetterne un'adozione diffusa e deve fornire disponibilità e buone prestazioni anche in presenza di lunghe distanze, di guasti hardware, di reti talvolta fragili e di utenti e carichi di lavoro sempre in crescita.

19.10 Sommario

- Un sistema distribuito è un insieme di unità d'elaborazione che non condividono memoria o un clock. Al contrario, ciascuna unità d'elaborazione ha la propria memoria locale e la comunicazione avviene attraverso varie linee di comunicazione, come bus ad alta velocità o Internet. Le dimensioni e le funzioni delle unità d'elaborazione di un sistema distribuito sono diverse.
- Un sistema distribuito fornisce all'utente l'accesso a tutte le risorse offerte dal sistema stesso. L'accesso a una risorsa condivisa può essere fornito tramite la migrazione dei dati, la migrazione delle computazioni o la migrazione dei processi. L'accesso può essere specificato dall'utente o implicitamente fornito dal sistema operativo e dalle applicazioni.
- Le pile di protocolli, com'è specificato dai modelli di stratificazione delle reti, aggiungono informazioni ai messaggi per assicurare che essi raggiungano la loro destinazione.
- Per tradurre il nome di un calcolatore nel corrispondente indirizzo di rete si deve usare un sistema di naming come il dns; un altro protocollo, come l'arp, può servire a tradurre l'indirizzo di rete in un indirizzo fisico del dispositivo, per esempio un indirizzo Ethernet.
- Se i sistemi risiedono su reti diverse sono necessari dei router per trasferire i pacchetti dalla rete di provenienza a quella di destinazione.
- I protocolli di trasporto udp e tcp indirizzano i pacchetti ai processi in attesa mediante l'uso di numeri di porta univoci a livello di sistema. Inoltre, il protocollo tcp consente al flusso di pacchetti di diventare un flusso di byte affidabile e orientato alla connessione.
- Per far funzionare correttamente un sistema distribuito vi sono diversi ostacoli da superare. Tra le varie problematiche da affrontare vi sono il naming dei nodi e dei processi nel sistema, la tolleranza ai guasti, il ripristino dopo un guasto e la scalabilità. Le problematiche relative alla scalabilità includono la gestione di un maggior carico, la tolleranza ai guasti e l'uso di schemi di archiviazione efficienti, che offrano la possibilità di compressione e/o deduplicazione.
- Un dfs è un servizio di file system i cui client, server e dispositivi di memorizzazione sono sparsi tra i siti di un sistema distribuito. Di conseguenza, l'attività di servizio si deve eseguire per mezzo della rete e invece di un unico sito di memorizzazione dei dati centralizzato vi sono più dispositivi di memorizzazione indipendenti.
- Esistono due principali modelli di dfs: il modello client-server e il modello basato su cluster. Il modello client-server consente la condivisione trasparente dei file tra uno o più client. Il modello basato su cluster distribuisce i file tra uno o più server di dati ed è progettato per l'elaborazione parallela di dati su larga scala.
- Idealmente, un dfs dovrebbe apparire ai propri client come un normale file system centralizzato (sebbene potrebbe non essere conforme alle tradizionali interfacce di file system come posix). La molteplicità e la dispersione dei suoi server e dei suoi dispositivi di archiviazione dovrebbero essere trasparenti. Un dfs trasparente facilita la mobilità dei client portando l'ambiente di lavoro del client nel sito in cui il client effettua l'accesso.
- Esistono parecchi metodi per il naming in un dfs. In quello più semplice i file si nominano attraverso una combinazione del loro nome di macchina e del loro nome locale, il che garantisce un nome unico per tutto il sistema. Un altro metodo, reso comune dall' nfs, fornisce mezzi per unire directory remote a directory locali, dando così l'impressione di un albero di directory coerente.
- Le richieste d'accesso a un file remoto sono generalmente gestite con due metodi complementari. Con il servizio remoto, le richieste d'accesso sono consegnate al server; la macchina server esegue gli accessi e i loro risultati sono riportati al client. Con l'uso di cache, se i dati necessari per soddisfare la richiesta d'accesso non sono ancora stati copiati nella cache, una copia di quei dati viene portata dal server al client. Gli accessi si eseguono sulla copia presente nella cache. Il problema di mantenere le copie nelle cache coerenti con il file principale costituisce il problema della coerenza della cache.

Esercizi di ripasso

19.1 Perché il passaggio di pacchetti broadcast tra le reti sarebbe una pessima scelta per i gateway? Quali sarebbero i vantaggi di tale comportamento?

19.2 Discutete vantaggi e svantaggi del salvataggio in una cache di traduzioni di nomi per computer localizzati in domini remoti.

19.3 Quali sono due difficili problemi che i progettisti devono risolvere nell'implementazione di un sistema di rete avente la qualità di essere trasparente?

19.4 Per costruire un sistema distribuito robusto occorre conoscere quali tipi di guasti possono verificarsi.

- a. Elencate tre tipi possibili di guasto in un sistema distribuito.
- b. Specificate quali di questi guasti si possono verificare anche su sistemi centralizzati.

19.5 È sempre di cruciale importanza sapere che un messaggio inviato è correttamente giunto a destinazione? Se la vostra risposta è sì, motivatela. Se la vostra risposta è no, fornite un esempio.

19.6 Considerate un sistema distribuito con due siti A e B. Valutate se il sito A può distinguere fra i seguenti problemi

- a. Cade B.
- b. Cade il collegamento tra A e B.
- c. B è estremamente sovraccarico e il suo tempo di risposta è di 100 volte superiore al normale.

Quali implicazioni ha la vostra risposta sul ripristino in sistemi distribuiti?

Esercizi

19.7 Qual è la differenza fra la migrazione della computazione e dei processi? Quale delle due è più facile da implementare e perché?

19.8 Anche se il modello di rete osi specifica sette strati di funzioni, nella maggior parte dei sistemi elaborativi si usano meno strati per realizzare una rete; spiegate perché e dite quali problemi può causare l'uso di un minor numero di strati.

19.9 Spiegate perché il raddoppio della velocità di un sistema in un segmento Ethernet può causare una diminuzione delle prestazioni della rete. Dite quali modifiche potrebbero migliorare la situazione.

19.10 Dite quali vantaggi porta l'uso di dispositivi specifici per le funzioni di router e gateway. Individuate gli svantaggi dell'impiego di calcolatori d'uso generale.

19.11 Dite in quali situazioni l'uso di un server dei nomi è più vantaggioso dell'uso di tabelle statiche nel calcolatore. Dite quali sono i problemi e le complicazioni connessi all'uso dei server dei nomi. Dite quali metodi si possono usare per ridurre il traffico generato dai server dei nomi per soddisfare le richieste di traduzione.

19.12 I server dei nomi sono organizzati in maniera gerarchica; a che scopo?

19.13 Gli strati inferiori del modello osi prevedono un servizio di datagrammi, senza garanzie di consegna dei messaggi. Lo strato di trasporto di protocolli come tcp/ip serve ad assicurare affidabilità. Discutete vantaggi e svantaggi del mettere a disposizione la consegna affidabile al più basso strato possibile.

19.14 Eseguite il programma della Figura 19.4 per determinare l'indirizzo ip dei seguenti nomi simbolici:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westmistercollege.edu
- www.ietf.org

19.15 Un nome di dominio come www.google.com si può mappare su più server. Tuttavia, se eseguiamo il programma mostrato nella Figura 19.4 otteniamo solamente un indirizzo ip. Modificate il programma in modo che visualizzi gli indirizzi ip di tutti i server invece di uno soltanto.

19.16 L'originario protocollo http impiegava il tcp/ip come protocollo sottostante: per ciascuna pagina, grafico o applet si costruiva, utilizzava, quindi chiudeva una distinta sessione tcp. Con questo metodo, a causa del sovraccarico dovuto alla costruzione e chiusura delle connessioni tcp/ip, si verificavano problemi di prestazioni. Dite se l'uso dell'udp al posto del tcp costituirebbe una buona alternativa e se è possibile fare altre modifiche che migliorino le prestazioni dell'http.

19.17 Elencate i vantaggi e gli svantaggi della trasparenza di una rete di calcolatori per l'utente.

19.18 Per ciascuno dei seguenti carichi applicativi dite se verrebbe gestito meglio da un modello di dfs basato su cluster o da un modello client-server. Motivate la vostra risposta.

- Ospitare i file degli studenti di un laboratorio universitario.
- Elaborare i dati inviati dal telescopio Hubble.
- Condividere i dati di un server domestico su più dispositivi.

19.19 Elencate i vantaggi di un dfs rispetto al file system di un sistema centralizzato.

19.20 Dite, motivando la risposta, se Openafs e nfs garantiscono: (a) trasparenza di locazione (b) indipendenza dalla locazione.

19.21 Dite in quali circostanze un client preferisce un dfs con trasparenza di locazione e in quali circostanze preferisce un dfs con indipendenza di locazione. Discutete i motivi di queste preferenze.

19.22 Dite quali aspetti di un sistema distribuito scegliereste per un sistema in esecuzione su una rete totalmente affidabile.

19.23 Confrontate le tecniche di copiatura in una cache locale dei blocchi dei dischi nel sistema client con le tecniche di caching sul server.

19.24 Quale dei due schemi, la deduplicazione a livello di file o a livello di blocco, porterebbe probabilmente a un maggior risparmio di spazio in un dfs multiutente? Motivate la vostra risposta.

19.25 Quali tipi di metadati aggiuntivi dovrebbero essere memorizzati in un dfs che utilizzi la deduplicazione?

CAPITOLO 20

Linux

Aggiornato da Robert Love

Questo capitolo presenta un'analisi approfondita del sistema operativo Linux. Esaminando un sistema reale nel dettaglio potremo vedere come i concetti che abbiamo discusso siano in relazione tra loro e vengono applicati praticamente.

Linux è una versione di unix che negli ultimi decenni ha avuto una crescente diffusione e viene ora utilizzato sia su piccoli dispositivi come i telefoni cellulari che su grandi supercomputer. In questo capitolo è trattata la storia e lo sviluppo di Linux, e sono esaminate le sue interfacce per il programmatore e l'utente, interfacce che devono molto alla tradizione di unix. Si analizzano anche i metodi adottati per il progetto e l'implementazione di queste interfacce. Linux è un sistema operativo in rapida evoluzione: il kernel descritto in questo capitolo è la versione 4.12, rilasciata nel 2017.

20.1 Storia di Linux

Il sistema operativo Linux assomiglia molto a un qualunque altro sistema unix, e in effetti la compatibilità con unix è stata uno degli obiettivi principali nella sua progettazione; tuttavia, il sistema Linux è molto più recente della maggior parte dei sistemi unix. Il suo sviluppo ha avuto inizio nel 1991 quando Linus Torvalds, uno studente finlandese, cominciò a sviluppare un kernel piccolo ma autosufficiente per la cpu 80386, la prima vera cpu a 32 bit della gamma Intel per i pc-compatibili.

Già fin dagli inizi del suo sviluppo il codice sorgente di Linux fu reso gratuitamente disponibile tramite la rete Internet, senza costi e con restrizioni minime sulle possibilità di distribuirlo: ne è risultata una storia ricca di contributi da parte di sviluppatori di tutto il mondo che comunicavano quasi esclusivamente tramite Internet. Da un kernel iniziale che realizzava parzialmente un ridotto sottoinsieme dei servizi di sistema di unix, il sistema Linux è cresciuto fino a includere tutte le funzionalità che ci si aspetta di trovare in un moderno sistema unix.

I primi stadi di sviluppo di Linux avevano a che fare in gran parte con il kernel del sistema operativo, il programma privilegiato che interagisce direttamente con l'hardware del calcolatore e gestisce tutte le risorse del sistema; chiaramente per ottenere un sistema operativo completo occorre molto più di questo kernel. È utile distinguere il kernel di Linux da un sistema Linux completo. Il **kernel di Linux** è un programma completamente originale sviluppato interamente dalla comunità Linux; il **sistema Linux**, nella sua forma odierna, incorpora una moltitudine di componenti, alcuni scritti *ex novo*, altri presi in prestito da diversi progetti di sviluppo e altri ancora creati in collaborazione con altri gruppi di programmatore.

Il sistema Linux di base è un ambiente standard per le applicazioni e per il codice scritto dagli utenti, ma non impone convenzioni rigide sulla gestione complessiva delle funzionalità. Un ulteriore livello organizzativo si è reso necessario con la progressiva maturazione di Linux: tale necessità è stata soddisfatta da varie distribuzioni. Una **distribuzione Linux** include tutti i componenti ordinari del sistema operativo più una serie di strumenti di gestione che semplifica l'installazione iniziale, gli aggiornamenti successivi, e l'installazione o la rimozione di altri pacchetti. Un moderno pacchetto di distribuzione, inoltre, fornisce di solito strumenti per la gestione dei file system, la creazione e gestione degli account utente, l'amministrazione dei servizi di rete, browser web, word processor e così via.

20.1.1 Kernel del sistema Linux

La prima versione del kernel di Linux resa disponibile al pubblico fu la 0.01, datata 14 maggio 1991; non forniva alcun servizio di rete, funzionava solo su pc con cpu compatibile con Intel 80386, e la sua gestione dei driver dei dispositivi era estremamente limitata. Anche il sottosistema per la memoria virtuale era abbastanza elementare, e non supportava i file memory-mapped; tuttavia, anche questa primissima versione permetteva la condivisione delle pagine con copiatura su scrittura (copy-on-write) e spazi degli indirizzi protetti. Il solo file system disponibile era quello di Minix, perché i primi kernel Linux furono sviluppati su piattaforme Minix.

La successiva versione principale, Linux 1.0, fu rilasciata il 14 marzo 1994; rappresentava il culmine di tre anni di rapido sviluppo del kernel. Forse la più importante nuova caratteristica riguardava la gestione dei servizi di rete: incorporava l'implementazione standard unix dei protocolli tcp/ip, e un'interfaccia socket compatibile con bsd unix per la programmazione di rete; grazie ai nuovi driver era anche possibile utilizzare il protocollo ip su una rete Ethernet o su linee seriali e modem usando i protocolli ppp o slip.

Il kernel 1.0 includeva anche un nuovo file system molto migliorato che non soffriva delle limitazioni del file system originario Minix; inoltre, supportava diversi tipi di controller scsi per l'accesso ai dischi ad alte prestazioni. Anche il sottosistema per la memoria virtuale era stato migliorato, e dava ora la possibilità della paginazione per effettuare lo swapping dei file e di effettuare il memory mapping di file arbitrari (anche se solo per operazioni di lettura).

Questa versione supportava inoltre una serie di ulteriori dispositivi che, seppur ancora limitati alla piattaforma Intel-pc, comprendevano floppy disk e cd-rom, schede audio, vari tipi di mouse e tastiere internazionali. Il kernel era anche in grado di simulare le operazioni in virgola mobile per i calcolatori basati sulla cpu 80386 che non erano dotate del coprocessore matematico 80387, e realizzava inoltre la **comunicazione tra processi** (ipc) nello stile di unix System V, compresa la memoria condivisa, i semafori e le code di messaggi.

A questo punto si cominciò a sviluppare la versione 1.1 del kernel, ma fu necessario distribuire in seguito numerose patch per la versione 1.0, nella quale erano stati riscontrati errori. Come convenzione di numerazione per le versioni del kernel fu adottato lo schema seguente: le versioni con un numero dispari dopo il punto, come la 1.1 o la 2.5, sono **kernel di sviluppo**; quelle con numero pari dopo il punto sono invece **kernel di produzione**. Gli aggiornamenti rispetto alle versioni stabili sono da intendersi solo come correzioni, mentre i kernel di sviluppo potrebbero includere nuovi servizi relativamente poco collaudati. Come vedremo, questo modello è rimasto in vigore fino alla versione 3.

Il kernel 1.2 fu rilasciato nel marzo 1995; non rappresentava un miglioramento paragonabile a quello della versione 1.0, ma in ogni modo permetteva la gestione di una ben più ampia varietà di hardware, compreso il nuovo bus pci. Gli sviluppatori inclusero anche la possibilità di usare la modalità virtuale 8086 della cpu 80386 – un'altra funzione specificamente dedicata ai pc – che permetteva di emulare il sistema operativo dos per pc. Fu inoltre aggiornata l'implementazione del protocollo IP, con l'aggiunta del supporto per accounting e firewall, e venne incluso un semplice supporto per i moduli del kernel caricabili e scaricabili dinamicamente.

Il kernel 1.2 fu anche l'ultima versione dedicata ai soli pc: i file sorgenti di Linux 1.2 comprendevano già un supporto parziale per le cpu sparc, Alpha e mips, ma la piena integrazione di queste altre architetture ebbe inizio solo dopo la diffusione della versione stabile 1.2.

Linux 1.2 si concentrava sul problema di gestire un maggiore varietà di hardware e di fornire implementazioni più complete delle funzioni esistenti. Molti nuovi servizi erano a quel tempo in via di sviluppo, ma l'integrazione del nuovo codice all'interno del kernel fu rimandata a dopo la distribuzione della versione stabile del kernel 1.2; di conseguenza, la versione 1.3 portò con sé una gran quantità di nuove funzionalità.

Questo materiale fu infine distribuito come Linux 2.0 nel giugno 1996. L'incremento del numero di versione principale era giustificato da due nuove caratteristiche di basilare importanza: il supporto di diverse architetture, tra cui il supporto nativo a 64 bit per Alpha, e il supporto al multiprocessing simmetrico (smp). Inoltre, il codice di gestione della memoria fu notevolmente migliorato per fornire una cache unificata per i dati del file system indipendente dalla cache dei dispositivi a blocchi. Grazie a questa modifica, il kernel poté offrire prestazioni notevolmente migliorate per quel che riguarda il file system e la memoria virtuale. Per la prima volta i meccanismi di caching del file system furono estesi ai file system di rete e vennero supportate le regioni scrivibili mappate in memoria. Tra le altre importanti novità citiamo i thread interni al kernel, la gestione delle dipendenze fra i moduli caricabili e il caricamento automatico dei moduli richiesti, il meccanismo delle quote per il file system e l'adozione delle classi di scheduling dei processi per l'elaborazione in tempo reale, compatibili con lo standard posix.

Nel 1999 fu rilasciato Linux 2.2 con ulteriori miglioramenti. È stato introdotto un porting su sistemi Ultrasparc. I servizi di rete sono stati affinati con un servizio di firewall più flessibile, una migliore gestione del traffico e dell'instradamento, e l'uso di finestre tcp ampie e di acknowledgement selettivi. Era diventato possibile leggere i dischi Acorn, Apple ed ntfs, ed era stato stato migliorato l' nfs con l'aggiunta di un demone, eseguito in modalità kernel. La gestione dei segnali, delle interruzioni, e di alcuni aspetti dell'i/o era eseguita impiegando un sistema di lock più selettivo per migliorare le prestazioni dell'smp.

Fra i miglioramenti delle versioni 2.4 e 2.6, citiamo i progressi al supporto dei sistemi smp, i journaling file system, oltre a migliorie alla gestione della memoria e dell'i/o a blocchi. Nella versione 2.6 lo scheduler dei processi è stato modificato in modo da usare un efficiente algoritmo di scheduling che esegue in tempo $O(1)$. Inoltre il kernel di Linux 2.6 applica la prelazione anche ai processi eseguiti in modalità kernel.

La versione 3.0 del kernel Linux è stata rilasciata nel luglio 2011. Il passaggio a un nuovo numero di versione principale è stato effettuato per ricordare il ventesimo anniversario di Linux. Le nuove caratteristiche includono un migliorato supporto della virtualizzazione, un nuovo meccanismo di write-back delle pagine, miglioramenti al sistema di gestione della memoria e un'ulteriore novità per quanto riguarda lo scheduler: il Completely Fair Scheduler (cfs).

Il kernel 4.0 di Linux è stato rilasciato nell'aprile 2015. Questa volta la modifica del numero principale della versione è stata del tutto arbitraria: gli sviluppatori del kernel di Linux erano stanchi di numeri di versioni secondarie sempre più grandi. Oggi le versioni dei kernel di Linux non rappresentano altro che l'ordine di rilascio. La serie 4.0 del kernel fornisce il supporto per nuove architetture, oltre a funzionalità mobili migliorate e a molti altri perfezionamenti. Nella parte restante di questo capitolo ci concentreremo su questo nuovo kernel.

20.1.2 Il sistema Linux

Come abbiamo già osservato, il kernel rappresenta per molti aspetti la parte centrale del progetto Linux, ma altri componenti concorrono a costituire un sistema operativo Linux completo. Mentre il kernel è composto di codice scritto *ex novo* specificamente per il progetto Linux, molti programmi aggiuntivi che completano il sistema non sono stati espressamente concepiti per il sistema Linux, ma sono condivisi da un certo numero di sistemi operativi ispirati a unix. In particolare il sistema Linux adotta molti strumenti sviluppati come parti del sistema operativo bsd di Berkeley, del sistema X Window del mit, e del progetto gnu della Free Software Foundation.

Questa condivisione ha funzionato in entrambe le direzioni: lo sviluppo delle principali librerie di sistema di Linux ha avuto origine dal progetto gnu, ma la comunità Linux le ha poi notevolmente migliorate affrontandone le carenze e inefficienze, ed eliminando gli errori. Altri componenti, per esempio il compilatore per il linguaggio C del progetto gnu, **gcc (gnu C compiler)**, erano già di qualità sufficientemente alta per essere direttamente usati in Linux. Gli strumenti di gestione dei servizi di rete sono stati derivati da codice originariamente scritto per bsd 4.3, ma discendenti più recenti del bsd come Freebsd hanno preso a loro volta in prestito codice dal sistema Linux. Tra gli esempi di questa condivisione vi sono la libreria di simulazione delle operazioni in virgola mobile per le cpu Intel e i driver dei dispositivi audio per pc.

La manutenzione complessiva del sistema Linux è curata da una rete alquanto lasca di sviluppatori che collaborano tramite la rete Internet, e la responsabilità dell'integrità di alcune specifiche componenti è assegnata a piccoli gruppi o individui. Un limitato numero di archivi ftp pubblici su Internet agiscono *de facto* come repository standard di questi componenti. Anche il documento **File System Hierarchy Standard** è curato dalla comunità Linux al fine di preservare la compatibilità fra le varie parti del sistema: esso specifica la struttura generale di un file system Linux, stabilendo sotto quali nomi di directory i file di configurazione, le librerie, i file eseguibili di sistema e i file di dati dovrebbero essere archiviati.

20.1.3 Distribuzioni Linux

In teoria chiunque potrebbe installare un sistema Linux compilando le ultime versioni del codice sorgente ottenibili dai siti ftp; una volta questo era esattamente ciò che ogni utente Linux doveva fare. A mano a mano che il sistema maturava, però, diversi individui o gruppi hanno cercato di rendere quest'inconvenienza meno pesante fornendo pacchetti precompilati standard di facile installazione.

Queste raccolte, o distribuzioni, comprendono molto di più del sistema Linux di base: generalmente forniscono strumenti aggiuntivi per l'installazione e la gestione del sistema, e anche pacchetti precompilati e pronti per l'installazione che forniscono molti comuni strumenti del sistema unix, per esempio server di news, browser web, text editor e persino giochi.

Le prime distribuzioni fornivano semplicemente un modo per estrarre e collocare i file nelle posizioni appropriate. Uno dei contributi importanti delle moderne versioni è la gestione avanzata dei pacchetti: esse comprendono un base dati di tracciatura dei pacchetti che permette di installare, aggiornare e rimuovere senza difficoltà i pacchetti desiderati.

Nei primi anni di vita di Linux la versione sls fu la prima raccolta di pacchetti Linux classificabile come una vera e propria distribuzione; tuttavia, anche se poteva essere installata in un'unica soluzione, non fornisce gli strumenti di gestione dei pacchetti che ci si può aspettare oggi. La versione **Slackware** rappresentò complessivamente un gran salto di qualità, nonostante la sua gestione dei pacchetti fosse piuttosto scadente, essa è ancora oggi assai diffusa tra la comunità Linux.

Dal rilascio di Slackware, sono divenute disponibili molte nuove distribuzioni commerciali e non commerciali. Due versioni di grande diffusione sono **Red Hat** e **Debian**, la prima prodotta da un'impresa commerciale, la seconda fornita dalla comunità Linux. Altre distribuzioni commerciali sono **Canonical** e **SUSE**. Ci sono molte altre versioni di Linux, ma sono troppe per poterle elencare tutte. Questa gran varietà non impedisce la compatibilità fra distribuzioni diverse. Il formato rpm per i file contenenti i pacchetti è adottato o almeno compreso dalla maggior parte delle distribuzioni, e le applicazioni commerciali distribuite in questo formato si possono installare ed eseguire con qualunque versione di Linux che accetti i file rpm.

20.1.4 Licenze Linux

Il kernel di Linux è distribuito in accordo alla versione 2.0 della gnu General Public License (gpl), i cui termini sono stati stabiliti dalla Free Software Foundation. Il sistema Linux non è software **public domain**: questa locuzione implica la rinuncia ai diritti d'autore, mentre i diritti sul codice di Linux sono tuttora detenuti dai vari autori. Tuttavia, il sistema Linux è *libero* nel senso che si può copiare, modificare e usare in qualunque modo si desideri, e si può far circolare (o vendere) le proprie copie.

La principale conseguenza dei termini della licenza di Linux è che chiunque lo usi o crei un prodotto da esso derivato (un'attività legittima) non può distribuire il suo prodotto senza il codice sorgente. I programmi distribuiti in accordo con la gpl non si possono ridistribuire in forma puramente binaria. Se rilasciate software che comprende una qualsiasi componente coperta da gpl dovete rendere il codice sorgente disponibile sotto gpl insieme a qualsiasi distribuzione del codice binario (si noti che questa limitazione non impedisce la realizzazione o persino la vendita di versioni puramente binarie, purché chiunque riceva i file eseguibili abbia la possibilità di ottenere il codice sorgente a un ragionevole prezzo).

20.2 Princìpi di progettazione

Dal punto di vista del progetto complessivo il sistema Linux assomiglia a qualunque altro sistema unix tradizionale non basato su microkernel. È un sistema multitutente multitasking con prelazione dotato di una completa serie di strumenti compatibili con unix; il suo file system aderisce alla tradizionale semantica unix, e anche il modello standard di comunicazione in rete di unix è pienamente realizzato. I dettagli interni della progettazione di Linux sono stati assai influenzati dalla storia del suo sviluppo.

Sebbene oggi il sistema Linux possa essere eseguito su un'ampia gamma di piattaforme, esso fu inizialmente concepito esclusivamente per l'architettura dei pc, e gran parte di questa prima fase di sviluppo fu portata a termine da appassionati e non da professionisti dotati di infrastrutture di ricerca e sviluppo ben finanziate: il risultato fu che, fin dall'inizio, Linux ottenne da risorse limitate il massimo possibile di capacità funzionali. Anche se oggi funziona perfettamente su calcolatori con più unità d'elaborazione con gigabyte di memoria centrale e terabyte di spazio di dischi, Linux può ancora funzionare con meno di 16 mb di ram.

A mano a mano che i pc divenivano più potenti e la memoria e i dischi più economici, l'originario kernel "minimalista" di Linux s'arricchiva di nuovi servizi tipici del sistema unix; la velocità e l'efficienza sono ancora oggi obiettivi importanti, ma molta ricerca recente e attualmente in corso si concentra su un terzo importante obiettivo: la standardizzazione. Uno dei prezzi che occorre pagare per la diversità delle versioni di unix disponibili è il fatto che non sia garantita la possibilità di trasferire codice sorgente fra due sue versioni: anche quando le chiamate di sistema sono le medesime nei due sistemi, non è detto che il loro comportamento sia identico. Gli standard posix comprendono un insieme di specifiche riguardanti diversi aspetti del comportamento di un sistema operativo, ed esistono documenti posix relativi ai servizi più comuni dei sistemi operativi e a estensioni quali i thread dei processi e le operazioni in tempo reale. Il sistema Linux è concepito al fine di conformarsi ai documenti posix, e almeno due sue distribuzioni hanno ottenuto la certificazione posix.

Presentando interfacce standard sia al programmatore sia all'utente, il sistema operativo Linux riserva poche sorprese a chiunque conosca già il sistema unix, queste interfacce non saranno trattate qui nei dettagli. Il contenuto dei paragrafi sull'interfaccia per il programmatore (Paragrafo C.3 Appendice C reperibile sulla piattaforma MyLab) e per l'utente del sistema bsd (Paragrafo C.4 Appendice C reperibile sulla piattaforma MyLab) vale anche per il sistema Linux. Si noti solo che ordinariamente l'interfaccia per il programmatore di Linux segue la semantica di unix svr4 e non quella del bsd; per ottenere la semantica del bsd, nei casi in cui vi sono differenze significative, è disponibile una raccolta di librerie.

Esistono molti altri standard nel mondo unix, ma la certificazione di Linux in conformità a essi è a volte rallentata dal fatto che di solito queste certificazioni sono disponibili solo a pagamento, e la certificazione di un sistema operativo rispetto alla maggior parte degli standard è un processo molto costoso. Ciononostante, poiché la capacità di eseguire un'ampia gamma di applicazioni è importante per qualunque sistema operativo, l'implementazione di standard è un obiettivo fondamentale nello sviluppo di Linux anche in assenza di certificazioni formali. Oltre allo standard di base posix, Linux attualmente comprende le estensioni posix per il threading (Pthreads) e un sottoinsieme delle estensioni posix per il controllo dei processi in tempo reale.

20.2.1 Componenti del sistema

Il sistema Linux è composto da tre porzioni principali di codice, in linea con le tradizionali versioni del sistema unix.

1. **Kernel.** Il kernel è responsabile di tutte le astrazioni importanti messe in atto dal sistema operativo, fra cui la memoria virtuale e i processi.
2. **Librerie di sistema.** Le librerie di sistema definiscono un insieme standard di funzioni grazie alle quali le applicazioni interagiscono col kernel, e realizzano la maggior parte dei servizi forniti dal sistema operativo che non necessitano dei pieni privilegi del kernel. La libreria di sistema più importante è la libreria C, conosciuta come libc. Oltre a fornire la libreria C standard, libc implementa la parte in modalità utente dell'interfaccia delle chiamate di sistema Linux e altre interfacce critiche a livello di sistema.
3. **Utilità di sistema.** Le utilità di sistema sono programmi che eseguono singoli compiti specializzati di gestione. Alcune di loro vengono chiamate solo una volta per inizializzare e configurare qualche aspetto del sistema; altre – note come **demoni** nella terminologia unix – rimangono permanentemente in esecuzione per eseguire compiti come la gestione delle richieste di nuove connessioni, la gestione delle richieste d'accesso dai terminali o l'aggiornamento dei file di log.

Nella Figura 20.1 sono illustrati i vari elementi che costituiscono un sistema Linux completo. La distinzione più importante in questo contesto è quella fra il kernel e tutto il resto. Tutto il codice del kernel è eseguito dalla cpu in modo privilegiato e con piena possibilità d'accesso alle risorse fisiche del calcolatore: nel contesto del sistema Linux ci si riferisce a questa modalità privilegiata come alla **modalità kernel**. Il kernel di Linux non contiene codice utente: tutto il codice di supporto al sistema operativo che non ha bisogno di essere eseguito in modalità kernel si trova nelle librerie di sistema e viene eseguito in modalità utente. A differenza della modalità kernel, la modalità utente ha accesso soltanto a un sottoinsieme controllato delle risorse di sistema.

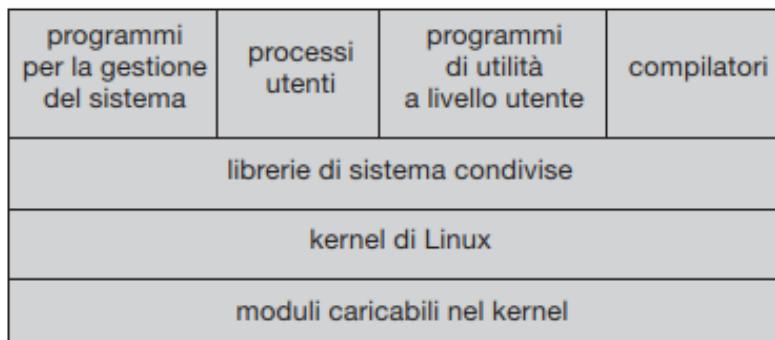


Figura 20.1 Componenti del sistema Linux.

Anche se parecchi sistemi operativi moderni hanno adottato per i loro kernel un'architettura basata sullo scambio di messaggi, Linux si conforma al modello storico dello unix: il kernel è un unico blocco monolitico di codice binario. La ragione principale di ciò sono le prestazioni. Poiché tutto il codice e le strutture dati del kernel sono contenute in un unico spazio d'indirizzi, non vi è necessità d'alcun cambio di contesto quando un processo invoca una funzione del sistema operativo o nel caso della gestione di un interrupt hardware. Inoltre il kernel può fare richieste e scambiare dati tra diversi sottosistemi invocando funzioni C relativamente economiche, rispetto alla più complicata comunicazione tra processi (IPC). Non è solo il codice di base relativo allo scheduling e alla gestione della memoria virtuale a occupare l'unico spazio d'indirizzi; *tutto* il codice del kernel, inclusi i driver dei dispositivi, il file system e il codice relativo al networking, si trova nello stesso spazio d'indirizzi.

Il fatto che tutto il codice del kernel sia messo nello stesso calderone non significa che non vi sia spazio per la modularità: così come le applicazioni possono caricare librerie condivise durante l'esecuzione per ottenere le parti di codice aggiuntivo di cui hanno bisogno, il kernel può caricare (o scaricare) dinamicamente moduli durante l'esecuzione. Non è necessario che il kernel sappia in anticipo quali moduli si dovranno caricare: i moduli sono componenti caricabili in modo veramente indipendente.

Il kernel costituisce la parte centrale del sistema operativo Linux: fornisce tutti i servizi necessari per eseguire i processi e gestire l'accesso alle risorse fisiche in modo controllato e protetto. In effetti, fa tutto ciò che un sistema operativo deve essere in grado di fare. Preso isolatamente, però, il sistema operativo fornito dal kernel non è un sistema unix completo: mancano molte delle caratteristiche di unix e anche quelle fornite non hanno il formato che le applicazioni si aspettano in ambiente unix. L'interfaccia del sistema operativo vista dalle applicazioni in esecuzione non fa propriamente parte del kernel: le applicazioni chiamano le librerie di sistema, che si servono a loro volta degli opportuni servizi del kernel.

Le librerie di sistema forniscono molti tipi di funzioni: a livello più semplice permettono alle applicazioni di effettuare chiamate di sistema al kernel. L'esecuzione di una chiamata di sistema richiede il passaggio del controllo dalla modalità utente a quella privilegiata del kernel; i dettagli di questo passaggio variano secondo l'architettura del sistema. Le librerie si occupano di raccogliere gli argomenti della chiamata di sistema e, se è necessario, organizzano gli argomenti nella forma richiesta dall'esecuzione della chiamata di sistema.

Le librerie possono anche fornire versioni più complesse delle chiamate di sistema di base: per esempio, le funzioni del linguaggio C di gestione dei file con buffer (*buffering*) sono tutte implementate nelle librerie di sistema; esse forniscono un più efficace controllo dell'i/o su file rispetto alle chiamate di sistema di base del kernel. Le librerie forniscono anche funzioni che non corrispondono per niente a chiamate di sistema, per esempio algoritmi di ordinamento, funzioni matematiche e procedure per la manipolazione di stringhe di caratteri. Tutte le funzioni necessarie per l'esecuzione di applicazioni unix o posix sono codificate nelle librerie di sistema.

Il sistema Linux include un'ampia varietà di programmi eseguibili in modalità utente: si tratta di utilità di sistema o di utente. Le utilità di sistema includono tutti i programmi necessari per inizializzare e poi amministrare il sistema, come quelli per configurare le interfacce di rete o per aggiungere e rimuovere utenti dal sistema. Anche gli strumenti per gli utenti sono necessari per il funzionamento di base del sistema, ma non sono richiesti privilegi elevati per eseguirli. Includono semplici programmi per la gestione dei file, come le utilità per copiare file, creare directory e modificare file di testo. Uno degli strumenti più importanti è la shell, l'interfaccia standard a riga di comando su sistemi unix. Linux supporta diverse shell: la più comune è la **shell Bourne-Again** (bash).

20.3 Moduli del kernel

Il kernel di Linux è in grado di caricare e scaricare su richiesta arbitrarie parti di codice: questi moduli caricabili sono eseguiti in modo privilegiato, e hanno quindi pieno accesso alle risorse fisiche del calcolatore. In teoria non c'è limite a ciò che un modulo del kernel può fare; può, per esempio, implementare un driver di dispositivo, un file system o un protocollo di comunicazione.

I moduli del kernel sono convenienti per diverse ragioni. Il codice sorgente di Linux è gratuito, e quindi chiunque volesse scrivere codice del kernel potrebbe compilare il kernel modificato e riavviare il sistema per caricare la nuova funzione, ma la ricompilazione, il collegamento e il ricaricamento dell'intero kernel è un processo laborioso che si dovrebbe ripetere molte volte durante la stesura di un driver. Usando i moduli del kernel non è necessario costituire un nuovo kernel per provare un nuovo driver: il driver si può compilare separatamente e può essere caricato dal kernel già in uso. Naturalmente, dopo che un nuovo driver è stato scritto, si può distribuire come modulo cosicché anche altri utenti potranno trarne beneficio senza dover ricompilare integralmente il kernel.

Quest'ultimo punto un'altra implicazione: essendo soggetto a licenza gpl, il kernel di Linux non può essere distribuito insieme con componenti proprietari, a meno che anch'essi non siano soggetti a licenza gpl e il loro codice sorgente non sia disponibile su richiesta. L'interfaccia a moduli del kernel permette a terzi di scrivere e distribuire alle loro condizioni driver di dispositivi o file system che non si potrebbero distribuire secondo le norme gpl.

I moduli del kernel permettono a un sistema Linux di essere attivato con un kernel minimale standard privo di driver aggiuntivi incorporati: i driver dei dispositivi necessari all'utente si possono caricare in modo esplicito nella fase d'avviamento del sistema, o possono essere caricati automaticamente dal sistema su richiesta e scaricati dopo l'uso. Un driver per un mouse, per esempio, può essere caricato nel momento in cui si inserisce la presa USB del mouse e scaricato quando la presa viene disinserita.

Gli elementi principali del supporto ai moduli di Linux sono i seguenti.

1. Il sistema di **gestione dei moduli** permette il caricamento dei moduli in memoria e la loro comunicazione con il resto del kernel.
2. Il **modulo di caricamento e scaricamento**, che è un'utilità in modalità utente, lavora con il sistema di gestione dei moduli e carica moduli in memoria.
3. Il sistema di **registrazione dei driver** permette di comunicare al resto del kernel la disponibilità di un nuovo driver.
4. La **risoluzione dei conflitti** è un meccanismo che permette a un driver di riservarsi l'uso di certe risorse fisiche e di proteggerle da un uso accidentale da parte di un altro driver.

20.3.1 Gestione dei moduli

Il caricamento di un modulo è un processo più complesso del semplice trasferimento in memoria del codice binario che lo costituisce: il sistema deve anche accertarsi del fatto che ogni riferimento fatto dal modulo a simboli o punti d'ingresso del kernel sia aggiornato per puntare alle corrette locazioni di memoria all'interno dello spazio d'indirizzi del kernel. Il sistema Linux affronta questo problema dividendo il caricamento in due parti: la gestione di sezioni di codice del modulo nella memoria del kernel, e quella dei simboli cui i moduli possono fare riferimento.

Il sistema Linux mantiene una tabella interna dei simboli nel kernel: essa non contiene l'intero insieme di simboli definiti durante la compilazione del kernel, ma solo i simboli esplicitamente esportati dal kernel. L'insieme di simboli esportati costituisce una ben definita interfaccia tramite la quale un modulo può interagire con il kernel.

Sebbene l'esportazione di un simbolo da una funzione del kernel presupponga la richiesta esplicita del programmatore, nessuno sforzo aggiuntivo è necessario per importare questi simboli in un modulo; chi scrive il modulo usa semplicemente il meccanismo standard di linking esterno del linguaggio C: tutti i simboli esterni cui il modulo fa riferimento ma che non sono dichiarati dal modulo stesso sono contrassegnati come irrisolti nel codice binario finale prodotto dal compilatore. Quando un modulo deve essere caricato in memoria, una utilità di sistema lo esamina per rilevare questi riferimenti irrisolti: il valore d'ogni simbolo esterno irrisolto viene cercato nella tabella dei simboli del kernel, e vengono sostituiti nel codice del modulo gli indirizzi corretti per quei simboli relativi al kernel attualmente in esecuzione. Solo a questo punto il modulo può essere caricato dal kernel: se l'utilità di sistema non riesce a risolvere un riferimento cercandone l'indirizzo nella tabella dei simboli del kernel, il modulo non è accettato.

Il caricamento di un modulo avviene in due fasi. In primo luogo, l'utilità di caricamento richiede al kernel di riservare per il modulo una regione continua di memoria virtuale del kernel; il kernel restituisce l'indirizzo dell'area di memoria allocata, e il caricatore può usare quest'indirizzo per rilocare il codice macchina del modulo rispetto all'indirizzo di caricamento corretto. Una seconda chiamata di sistema passa al kernel il nuovo modulo, insieme con l'eventuale tabella di simboli esterni che il modulo chiede di esportare: il modulo è copiato direttamente nello spazio di memoria precedentemente riservato, e si aggiorna la tabella dei simboli del kernel in base ai nuovi simboli per un eventuale uso da parte d'altri moduli non ancora caricati.

L'ultimo componente della gestione dei moduli è il meccanismo di richiesta dei moduli. Il kernel definisce un'interfaccia di comunicazione alla quale il programma di gestione dei moduli si può collegare; grazie a ciò il kernel potrà comunicare al programma di gestione se un processo richiede l'uso del driver di un dispositivo, di un file system o di un servizio di rete i cui moduli non sono ancora stati caricati, dando così al gestore la possibilità di caricare il modulo appropriato; le richieste di servizio saranno soddisfatte dopo il completamento di quest'operazione. Il programma di gestione interroga il kernel a intervalli regolari per appurare quali moduli dinamicamente caricati non sono più in uso, scaricando i moduli che non sono necessari al momento.

20.3.2 Registrazione dei driver

Un modulo che è stato caricato rimane una regione isolata della memoria fino a quando non comunica al kernel quali nuove funzionalità è in grado di fornire. Il kernel mantiene tabelle dinamiche di tutti i driver noti, e fornisce un insieme di procedure che permettono di aggiungere o rimuovere un driver da queste tabelle in qualunque momento. Il kernel assicura l'avviamento della

procedura d'inizializzazione d'ogni nuovo modulo caricato, e l'attivazione della procedura di chiusura prima che esso sia scaricato: queste procedure sono responsabili della registrazione delle funzionalità forniti dal modulo.

Un modulo può registrare molti tipi di funzionalità, senza limitarsi a un solo tipo. Un certo driver, per esempio, potrebbe voler registrare due meccanismi distinti per l'accesso a un dispositivo. Le tabelle di registrazione includono, tra gli altri, i seguenti elementi.

- **Driver dei dispositivi.** Questi driver comprendono i dispositivi a caratteri (come stampanti, terminali e mouse), i dispositivi a blocchi (fra i quali tutte le unità a disco), e i dispositivi d'interfacciamento alla rete.
- **File system.** Un file system è una qualsiasi entità che implementa le chiamate di sistema del file system virtuale di Linux: può trattarsi della implementazione di un formato per la scrittura di file nei dischi, oppure di un file system di rete come l'nfs, o ancora di un file system virtuale i cui contenuti sono generati su richiesta, come il file system `/proc`.
- **Protocolli di rete.** Un modulo può essere la implementazione di un intero protocollo di rete, come per esempio tcp, o più semplicemente un insieme di regole per il filtraggio dei pacchetti di dati per un firewall di rete.
- **Formato binario.** Questo formato specifica un modo per riconoscere, caricare ed eseguire un nuovo tipo di file eseguibile.

Inoltre, un modulo può registrare un nuovo insieme di elementi nelle tabelle `systcl` e `/proc`, in modo che lo stesso modulo si possa configurare dinamicamente (Paragrafo 20.7.4).

20.3.3 Risoluzione dei conflitti

Le versioni commerciali del sistema operativo unix sono di solito concepite per essere eseguite sui calcolatori di un singolo produttore; un vantaggio di questa situazione è che chi sviluppa i programmi ha un'idea piuttosto precisa delle possibili configurazioni dell'hardware. L'hardware di un pc, d'altro canto, è disponibile in un gran numero di configurazioni diverse, e con molti possibili driver per dispositivi come schede di rete e schede grafiche. Il problema di trattare le differenti configurazioni hardware si aggrava quando si adotta un criterio modulare di gestione dei driver dei dispositivi, perché l'insieme dei dispositivi attivi diviene variabile dinamicamente.

Il sistema Linux fornisce un meccanismo centrale di risoluzione dei conflitti che aiuta a regolare l'accesso a certe risorse hardware. I suoi obiettivi sono i seguenti:

- impedire che moduli diversi entrino in conflitto per l'accesso alle risorse hardware;
- impedire che una **procedura di autoverifica** (*autoprobes*) di un certo driver – cioè una procedura del driver che determina automaticamente la configurazione del dispositivo – interferisca con driver già presenti;
- risolvere i conflitti che possono nascere fra diversi driver che tentino di accedere alle stesse risorse fisiche; per esempio, sia il driver per la porta parallela di una stampante sia il driver di rete ip relativo a una porta parallela (*parallel-line ip*, plip) potrebbero cercare di comunicare con la porta parallela.

Per raggiungere questi scopi il kernel mantiene una lista delle risorse hardware assegnate. Il pc ha un numero limitato di possibili porte di i/o (indirizzi nel suo spazio d'indirizzi per i dispositivi di i/o), linee per le interruzioni e canali dma; da un driver di un dispositivo che voglia accedere a una di queste risorse ci si aspetta che prima di tutto prenoti la risorsa presso il database del kernel. Questo prerequisito, fra l'altro, permette all'amministratore del sistema di determinare esattamente quali risorse siano state riservate da quale driver a ogni dato istante.

Ci si attende che un modulo usi questo meccanismo per prenotare ogni risorsa fisica che intenda impiegare. Se la richiesta di prenotazione non è accettata, per esempio perché la risorsa non esiste o è già in uso, sta al modulo decidere quale tipo d'azione intraprendere: potrebbe dichiarare fallita la sua procedura d'inizializzazione e richiedere di essere scaricato, oppure continuare tentando di usare risorse hardware alternative.

20.4 Gestione dei processi

Un processo è il contesto fondamentale all'interno del quale tutte le attività richieste dagli utenti sono servite dal sistema operativo. Per essere compatibile con altri sistemi unix, il sistema operativo Linux deve adottare un modello dei processi simile a quello di altre versioni del sistema unix: tuttavia, Linux si comporta in modo diverso in alcuni punti chiave. In questo paragrafo si richiama il tradizionale modello unix dei processi (Appendice C, Paragrafo C.3.2, reperibile sulla piattaforma MyLab) e si presenta il modello di threading di Linux.

20.4.1 Modello fork()ed exec() dei processi

Il principio fondamentale della gestione dei processi in unix è di separare due operazioni che sono solitamente combinate: la creazione di un nuovo processo e l'esecuzione di un nuovo programma. La chiamata di sistema `fork()` assolve il primo compito, mentre l'esecuzione di un nuovo programma è avviata dalla chiamata di sistema `exec()`: queste due funzioni sono nettamente differenti: si può creare un nuovo processo tramite la `fork()` senza avviare l'esecuzione di un nuovo programma – il processo figlio continua a eseguire esattamente lo stesso programma, a partire dalla stessa posizione, del processo genitore. Similmente, l'esecuzione di un nuovo programma non richiede la creazione di un nuovo processo: un qualunque processo può eseguire la chiamata di sistema `exec()` in qualunque momento, nel qual caso un nuovo oggetto binario viene caricato nello spazio degli indirizzi del processo e il nuovo programma è eseguito nel contesto del processo esistente.

Il vantaggio di questo modello è la sua gran semplicità: non si deve specificare ogni dettaglio riguardante l'ambiente di un nuovo programma tramite la chiamata di sistema che ne avvia l'esecuzione: i nuovi programmi sono semplicemente eseguiti nell'ambiente esistente. Se un processo genitore desidera modificare l'ambiente d'esecuzione di un nuovo programma, può eseguire una `fork()` e poi, continuando a eseguire il programma originale nell'ambito del processo figlio, eseguire tutte le chiamate di sistema necessarie per modificare quel processo figlio prima di avviare infine il nuovo programma.

Nel sistema unix, quindi, un processo racchiude tutte le informazioni di cui il sistema operativo necessita per gestire il contesto di una singola esecuzione di un singolo programma; in Linux, questo contesto può essere suddiviso in un certo numero di sezioni specifiche. A grandi linee, le proprietà di un processo si dividono in tre gruppi: identità, ambiente e contesto del processo.

20.4.1.1 Identità dei processi

L'identità di un processo consiste principalmente dei seguenti elementi.

- **Identificatore del processo** (*process identifier, pid*). A ogni processo è associato un identificatore unico. Il pid si usa per specificare un certo processo al sistema operativo quando un'applicazione invoca una chiamata di sistema riferita a quel processo. Ulteriori identificatori associano un processo a un gruppo di processi (tipicamente un albero di processi generati da `fork()` a partire da un unico comando utente) e a una sessione di login.
- **Credenziali**. Ogni processo deve avere associati un identificatore utente e uno o più identificatori di gruppi di utenti che determinano i diritti d'accesso alle risorse del sistema e ai file (i gruppi di utenti sono discussi nel Paragrafo 13.4.2).
- **Personalità**. Il concetto di personalità di un processo non fa parte della tradizione di unix, ma nel sistema Linux ogni processo ha un identificatore di personalità associato che può modificare lievemente la semantica di certe chiamate di sistema: le personalità sono principalmente usate da librerie di emulazione per ottenere la compatibilità di certe chiamate di sistema con certe specifiche versioni di unix.
- **Spazio dei nomi**. Ogni processo è associato a una visione specifica della gerarchia del file system, chiamata il suo **spazio dei nomi** (namespace). La maggior parte dei processi condivide un namespace comune e opera quindi su una gerarchia di file system condivisa. I processi e i loro figli possono tuttavia avere namespace distinti, ognuno con un'unica gerarchia del file system, con la propria root directory e con il proprio insieme di file system montati.

Un processo ha un controllo limitato dei suoi stessi identificatori: gli identificatori del gruppo e della sessione di login possono essere cambiati se il processo desidera avviare un nuovo gruppo o una nuova sessione; le credenziali possono essere cambiate subordinatamente a certi controlli di sicurezza; il pid di un processo, invece, non è modificabile, e lo identifica in modo univoco fino alla sua terminazione.

20.4.1.2 Ambiente di un processo

L'ambiente di un processo è ereditato dal suo genitore, ed è composto di due vettori: il vettore degli argomenti e il vettore d'ambiente. Il **vettore degli argomenti** elenca semplicemente gli argomenti della riga di comando usata per invocare il programma in esecuzione, e comincia per convenzione con il nome del programma stesso. Il **vettore d'ambiente** è una lista di coppie della forma "NOME=VALORE" che associano ai nomi delle variabili d'ambiente valori testuali arbitrari. La descrizione dell'ambiente non è contenuta in memoria del kernel ma nello spazio d'indirizzi in modalità utente del processo, ed è il primo dato sullo stack del processo.

Questi due vettori non sono alterati a seguito della creazione di un nuovo processo: il processo figlio eredita l'ambiente del processo genitore. L'avvio di un nuovo programma, invece, implica la costituzione di un ambiente interamente nuovo: un processo che esegue la chiamata `exec()` deve anche specificare l'ambiente del nuovo programma tramite variabili d'ambiente che sono passate dal kernel al nuovo programma, e sostituiscono l'ambiente attuale del processo. A parte questa circostanza, i vettori d'ambiente e degli argomenti sono ignorati dal kernel; la loro interpretazione è lasciata interamente alle librerie in modalità utente e alle applicazioni.

Il passaggio delle variabili d'ambiente da un processo all'altro è il meccanismo di ereditarietà di queste variabili costituiscono un modo flessibile di scambiare informazioni ai componenti del sistema eseguiti in modalità utente. Alcune importanti variabili d'ambiente hanno un significato convenzionale correlato a certe parti del sistema: i valori della variabile `TERM`, per esempio, denotano il tipo di terminale usato durante una sessione di login, e molti programmi sfruttano questo fatto per stabilire come eseguire certe operazioni che hanno effetti sullo schermo dell'utente (spostamento del cursore, scorrimento di parti di testo, e così via). I programmi in grado di supportare più lingue usano la variabile `LANG` per stabilire in quale lingua mostrare i messaggi.

Il meccanismo delle variabili d'ambiente personalizza il sistema operativo, processo per processo: ogni utente può scegliere una lingua o un editor particolare indipendentemente dalle scelte altrui.

20.4.1.3 Contesto di un processo

L'identità di un processo e le proprietà dell'ambiente sono di solito stabilite al momento della creazione di un processo, e non mutano fino alla sua terminazione, anche se un processo potrebbe decidere di cambiare il proprio ambiente o alcuni aspetti della propria identità. Il contesto di un processo, invece, è lo stato del programma in esecuzione in ogni istante: esso cambia continuamente. Il contesto di un processo include le parti seguenti.

- **Contesto di scheduling.** La parte più importante del contesto di un processo è il suo contesto di scheduling: le informazioni di cui lo scheduler necessita per sospendere o riavviare il processo, comprese le copie di tutti i suoi registri. I registri in virgola mobile sono memorizzati separatamente, e ripristinati solo quando è necessario, per rendere più efficiente il salvataggio di stato dei processi che non usano l'aritmetica in virgola mobile. Il contesto di scheduling comprende anche informazioni sulla priorità di scheduling e su ogni segnale che attende d'essere recapitato al processo. Una parte cruciale del contesto di scheduling è lo stack di kernel del processo: un'area di memoria nello spazio d'indirizzi del kernel riservata esclusivamente all'uso di codice eseguito in modalità kernel; sia le chiamate di sistema sia la gestione delle interruzioni occorrono durante l'esecuzione del processo usano questo stack.
- **Accounting.** Il kernel mantiene informazioni sulle risorse correntemente impiegate da ciascun processo e anche sulla quantità totale di risorse impiegate durante la sua intera esistenza.
- **Tabella dei file.** È un array di puntatori alle strutture dati del kernel relative ai file aperti. Quando un processo esegue una chiamata di sistema di i/o relativa a un file, lo identifica tramite un intero, il suo descrittore di file (fd), che il kernel utilizza come indice in questa tabella.
- **Contesto del file-system.** Mentre la tabella dei file elenca i file aperti esistenti, il contesto del file system viene utilizzato nelle richieste di apertura di nuovi file. Il contesto del file system include la root directory del processo, la directory di lavoro corrente e lo spazio dei nomi.
- **Tabella dei gestori dei segnali.** I sistemi unix possono recapitare a un processo segnali asincroni in risposta a un evento esterno. Questa tabella definisce quali azioni si devono intraprendere quando si riceve un certo segnale. Tra le possibili azioni vi sono: ignorare il segnale, terminare il processo, invocare una procedura nello spazio degli indirizzi del processo.
- **Contesto della memoria virtuale.** Il contesto della memoria virtuale descrive completamente i contenuti dello spazio d'indirizzi di un processo, ed è trattato nel Paragrafo 20.6.

20.4.2 Processi e thread

Linux possiede la chiamata di sistema `fork()` per duplicare un processo senza caricare una nuova immagine eseguibile; esso offre anche un'altra opportunità per generare i thread, ovvero la chiamata di sistema `clone()`. Linux, però, non distingue tra processi e thread, impiegando generalmente il termine *task* – invece di *processo* o *thread* – in riferimento a un flusso di controllo nell'ambito di un programma. La chiamata `clone()` si comporta come la `fork()`, a eccezione del fatto che accetta come parametro un insieme di flag che stabiliscono quante e quali risorse del processo genitore debbano essere condivise dal processo figlio, mentre un processo creato mediante `fork()` non condivide risorse con il genitore. Alcuni di questi flag sono mostrati dallo schema seguente.

flag	significato
<code>CLONE_FS</code>	Le informazioni sul file system sono condivise
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dell'insieme dei file aperti

Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il processo genitore e il processo figlio condivideranno le medesime informazioni sul file system (come la directory di lavoro corrente), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Utilizzare `clone()` in questo modo equivale alla creazione di un thread in altri sistemi, dal momento che il processo genitore condivide la maggior parte delle proprie risorse con il processo figlio. Se invece nessuno dei flag è impostato, non si ha alcuna condivisione, e la funzionalità risultante è simile a quella fornita dalla chiamata di sistema `fork()`.

La mancanza di distinzione tra processi e thread è possibile perché Linux non memorizza l'intero contesto di un processo nella struttura dati principale del processo: esso è infatti distribuito in sottocontesti indipendenti. Pertanto, il contesto del file system di un processo, la tabella dei descrittori dei file, la tabella del gestore dei segnali e il contesto della memoria virtuale risiedono in strutture dati distinte. La struttura dati di un processo contiene semplicemente un puntatore per ogni struttura sopra citata. In tal modo, un

qualsiasi numero di processi può facilmente condividere un sottocontesto impostando il puntatore e incrementando un contatore di riferimenti al sottocontesto.

La chiamata di sistema `clone()` riceve argomenti che specificano quali sottocontesti copiare e quali condividere. Al nuovo processo è sempre attribuita una nuova identità e un nuovo contesto di scheduling (ciò è essenziale per un processo Linux); a seconda degli argomenti che riceve, tuttavia, il kernel può sia creare nuove strutture dati del sottocontesto, inizializzate come copie del processo genitore, sia stabilire che il nuovo processo utilizzi le stesse strutture dati del sottocontesto, usate dal processo genitore. La chiamata di sistema `fork()`, che copia tutti i sottocontesti senza condividerne alcuno, non è altro che un caso particolare di `clone()`.

20.5 Scheduling

Lo scheduling consiste nell'allocazione del tempo di cpu ai diversi task all'interno di un sistema operativo. Linux, come tutti i sistemi unix, supporta il multitasking con prelazione. In un tale sistema lo scheduler decide quale processo viene eseguito e quando. Prendere queste decisioni in modo da bilanciare equità e prestazioni sotto carichi di lavoro molto diversi è una delle sfide più complicate dei moderni sistemi operativi.

Di norma si pensa allo scheduling come all'esecuzione e sospensione dei processi utente, ma un altro aspetto dello scheduling importante per il sistema Linux è l'esecuzione dei vari task del kernel, che include sia i task richiesti da un processo in esecuzione, sia i task interni eseguiti per conto del kernel stesso, come i task generati dal sottosistema di i/o di Linux.

20.5.1 Scheduling dei processi

Il sistema Linux adotta due distinti algoritmi di scheduling: uno è un algoritmo time sharing con prelazione per l'equa condivisione del tempo di cpu da parte di più processi, e l'altro è concepito per elaborazioni in tempo reale dove le priorità assolute sono più importanti dell'equità.

L'algoritmo di scheduling adoperato per i task in time sharing ha beneficiato di forti innovazioni con il passaggio alla versione 2.6 del kernel. In precedenza, il kernel di Linux impiegava una variante del tradizionale algoritmo di scheduling di unix. Questo algoritmo non offre adeguato supporto ai sistemi smp, non scala bene al crescere del numero di task nel sistema e non gestisce equamente i task interattivi, in particolare su sistemi desktop e mobili. Lo scheduler dei processi è stato revisionato con la versione 2.5 del kernel, con l'implementazione di un algoritmo di scheduling che seleziona il task da eseguire in tempo costante, ossia $O(1)$, indipendentemente dal numero di task e processori nel sistema. Il nuovo scheduler fornisce un maggiore supporto per smp, includendo la predilezione del processore e il bilanciamento del carico. Questi cambiamenti, pur migliorando la scalabilità, non hanno migliorato le prestazioni per quanto riguarda i processi interattivi e l'equità, anzi, di fatto, hanno reso più problematiche certe situazioni sotto particolari carichi di lavoro. Lo scheduler dei processi è stato dunque revisionato per una seconda volta e con il kernel Linux versione 2.6 è stato inaugurato il **Completely Fair Scheduler** (cfs).

Lo scheduler di Linux è un algoritmo con prelazione basato su priorità, con due intervalli distinti di priorità: un intervallo **real-time**, con valori da 0 a 99, e un **nice value** con valori da -20 a 19. Valori di nice value minori indicano priorità più alta. Aumentando il nice value, si sta diminuendo la priorità e ci si sta comportando in modo "nice" (gentile) verso il resto del sistema.

cfs costituisce un allontanamento significativo dal tradizionale scheduler di unix, in cui le variabili fondamentali dell'algoritmo di scheduling sono priorità e porzioni di tempo (time slice). Il time slice è il tempo del processore che viene assegnato a un processo. I sistemi unix tradizionali danno ai processi un intervallo di tempo fissato, talvolta con una aggiunta o una penalità rispettivamente per i processi a priorità più alta e più bassa. Un processo può restare in esecuzione per tutta la durata della sua porzione di tempo e i processi con priorità maggiore vengono eseguiti prima dei processi a bassa priorità. Si tratta di un semplice algoritmo utilizzato da molti sistemi diversi da unix. Questo meccanismo molto semplice ha funzionato bene per i primi sistemi time-sharing, ma si è dimostrato incapace di fornire buone prestazioni in termini di equità e gestione di processi interattivi nei moderni sistemi desktop e nei dispositivi mobili.

cfs ha introdotto un nuovo algoritmo di scheduling, chiamato fair scheduling, che elimina i time slice intesi in senso tradizionale. Al posto delle porzioni di tempo, a tutti i processi viene assegnata una percentuale del tempo del processore. cfs calcola per quanto tempo un processo deve essere eseguito in funzione del numero totale di processi eseguibili. Per iniziare, cfs stabilisce che se ci sono N processi eseguibili a ognuno deve essere accordata una frazione 1/N del tempo del processore. A questo punto cfs calibra questa ripartizione ponderando l'assegnazione a ogni processo a seconda del suo nice value. Ai processi con nice value di default viene dato un peso pari a 1 (la loro priorità resta invariata), ai processi con un nice value minore (priorità superiore) viene dato un peso maggiore, mentre i processi con un nice value maggiore (priorità inferiore) ricevono un peso inferiore. cfs esegue poi ogni processo per un intervallo di tempo proporzionale al peso del processo diviso per il peso totale di tutti i processi eseguibili.

Per calcolare il periodo di tempo per cui un processo viene effettivamente eseguito, cfs si basa su una variabile configurabile chiamata **latenza obiettivo** (*target latency*) che rappresenta l'intervallo di tempo nel quale ogni task eseguibile dovrebbe andare in esecuzione almeno una volta. Si supponga, per esempio, che la latenza obiettivo sia di 10 millisecondi e si supponga di avere due processi eseguibili con la stessa priorità. Questi processi hanno lo stesso peso e quindi ricevono una uguale percentuale del tempo del processore. In questo caso, con una latenza obiettivo di 10 millisecondi, il primo processo viene eseguito per 5 millisecondi, quindi l'altro processo viene eseguito per 5 millisecondi, quindi il primo processo viene eseguito di nuovo per 5 millisecondi, e così via. Se avessimo 10 processi eseguibili, cfs manderebbe in esecuzione ogni processo per un millesimo di secondo prima di ripetere il ciclo.

Che cosa succederebbe se avessimo, per esempio, 10.000 processi? Secondo il procedimento descritto ogni processo resterebbe in esecuzione per 1 microsecondo e, a causa dei costi per lo switch di contesto, la pianificazione per una durata così breve risulterebbe inefficiente. Per risolvere questa questione cfs si basa su una seconda variabile configurabile, la granularità minima, che rappresenta il periodo minimo di tempo del processore che può essere assegnato a ogni processo. Tutti i processi, indipendentemente dalla latenza obiettivo, resteranno in esecuzione per un tempo pari almeno alla granularità minima. In questo modo cfs assicura che i costi dell'avvicendamento dei processi non crescano in maniera inaccettabile quando il numero di processi eseguibili diventa troppo grande, anche se, nel fare ciò, viola i suoi vincoli di equità. Visto che, tipicamente, il numero di processi eseguibili rimane ragionevole, sia i costi di avvicendamento che l'equità ne risultano massimizzati.

Con il passaggio al fair scheduling, il comportamento di cfs si differenzia da quello dello scheduler tradizionale unix sotto diversi aspetti. In particolare, come abbiamo visto, cfs elimina il concetto di time slice fisso. I processi ricevono invece una percentuale del tempo del processore e l'ammontare di questo tempo dipende da quanti altri processi eseguibili ci sono. Questo approccio risolve diversi problemi relativi all'assegnazione delle priorità negli algoritmi di scheduling con prelazione basati sulle priorità. È possibile, naturalmente, risolvere questi problemi in altri modi senza abbandonare il tradizionale scheduler unix. Tuttavia cfs è in grado di

risolvere i problemi con un semplice algoritmo che si comporta bene su sistemi interattivi come i telefoni cellulari senza compromettere le prestazioni sui server di grandi dimensioni.

20.5.2 Scheduling real-time

L'algoritmo di scheduling real-time di Linux è molto più semplice rispetto all'algoritmo fair scheduling utilizzato per i processi time-sharing standard. Linux implementa le due classi di scheduling in tempo reale richieste da posix.1b: fcfs e rr (Paragrafo 5.3.1 e Paragrafo 5.3.3, rispettivamente). In entrambi i casi, ogni processo ha una priorità in aggiunta alla sua classe di scheduling. Lo scheduler esegue sempre il processo con la priorità più alta, oppure, a parità di priorità, il processo che attende da più tempo. L'unica differenza fra lo scheduling fcfs e quello rr è che un processo fcfs continua l'esecuzione fino alla terminazione o finché non si blocca, mentre un processo rr può essere sospeso allo scadere del suo quanto di tempo, nel qual caso è posto alla fine della coda di scheduling; la conseguenza di ciò è che fra i processi rr della stessa priorità si effettua il time sharing.

Lo scheduling real-time di Linux è in tempo reale debole (*soft*), non in tempo reale stretto (*hard*); lo scheduler offre rigide garanzie sulle priorità relative dei processi in tempo reale, ma il kernel non fornisce alcuna garanzia sulla rapidità con cui un processo in tempo reale pronto per l'esecuzione sarà effettivamente eseguito. Nei sistemi in tempo reale hard, invece, viene garantito un ritardo massimo entro il quale un processo pronto per l'esecuzione deve essere eseguito.

20.5.3 Sincronizzazione del kernel

Il modo in cui il kernel esegue lo scheduling delle sue stesse operazioni è essenzialmente diverso dal modo in cui esegue lo scheduling dei processi. La richiesta d'esecuzione di codice in modalità kernel può avvenire in due modi: un programma in esecuzione può richiedere un servizio del sistema operativo esplicitamente, tramite una chiamata di sistema, o implicitamente, per esempio nel caso di un'eccezione di pagina mancante; oppure, il driver di un dispositivo può generare un segnale d'interruzione che induce l'esecuzione di un gestore di tale interruzione presente nel kernel.

Il problema che il kernel si trova a dover affrontare è che questi task potrebbero tentare di accedere alle stesse strutture dati interne: se una procedura di gestione di un segnale d'interruzione interrompe un task del kernel che sta usando certe strutture dati, la procedura in questione non può accedere a quelle stesse strutture dati se non a rischio di creare incoerenze. Quest'osservazione è connessa all'idea delle sezioni critiche, cioè parti di codice che condividono certe strutture dati e che non si possono eseguire in modo concorrente. Quindi la sincronizzazione del kernel richiede più del semplice scheduling dei processi: è necessario un meccanismo che permette l'esecuzione dei task del kernel senza rischiare i violare l'integrità dei dati condivisi.

Prima della versione 2.6, Linux adoperava un kernel senza prelazione: non era possibile applicare la prelazione ai processi eseguiti in modalità kernel, neppure nel caso che un processo con priorità più alta fosse pronto per l'esecuzione. Con il passaggio alla versione 2.6, il kernel ha adottato la prelazione: ogni task attivo nel kernel può ora essere sottoposto a prelazione.

Il kernel di Linux si serve di semafori convenzionali e spinlock (nonché della variante lettore-scrittore di questi due meccanismi) per implementare il locking nel kernel. Sulle macchine smp, il lock fondamentale è lo spinlock; il kernel è realizzato in modo da applicare gli spinlock solo per brevi periodi di tempo. Gli spinlock sono inadatti alle macchine monoprocesso, per cui si abilita e disabilita il diritto di prelazione del kernel. Su tali macchine, in pratica, anziché applicare uno spinlock, il kernel disabilita la prelazione, e anziché rimuovere lo spinlock, esso abilita la prelazione. Lo schema è il seguente.

monoprocesso	multiprocessore
disabilita la prelazione del kernel	applica spinlock
abilita la prelazione del kernel	rimuove spinlock

Il modo impiegato da Linux per attivare e disattivare il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici interfacce, `preempt_disable()` e `preempt_enable()`. Inoltre, se un task attivo in modalità kernel possiede un lock non è possibile ricorrere alla prelazione nel kernel. Per implementare questa regola, ogni task del sistema possiede una struttura `thread_info`, il cui campo `preempt_count` indica il numero dei lock posseduti dal task. Quando il task applica un lock, `preempt_count` è incrementato; quando ne rimuove uno, il contatore è decrementato. Se per il task in esecuzione `preempt_count` ha valore strettamente maggiore di zero, la prelazione a livello del kernel non è sicura, perché il task sta usando un lock. Se il valore è uguale a zero, invece, il kernel può essere interrotto, purché non vi siano chiamate in corso a `preempt_disable()`.

Gli spinlock e il meccanismo di abilitazione e inibizione della prelazione sono usati dal kernel solo quando il lock deve essere applicato per un breve periodo. Quando vi sia necessità di tenere in funzione più a lungo un lock, si usano i semafori.

La seconda tecnica di protezione che Linux usa si applica alle sezioni critiche incluse nelle procedure di gestione degli interrupt. Lo strumento fondamentale in questo caso è l'hardware di controllo delle interruzioni della cpu: disabilitando le interruzioni durante l'esecuzione di una sezione critica, il kernel si assicura di poter procedere senza rischiare accessi concorrenti a strutture dati condivise.

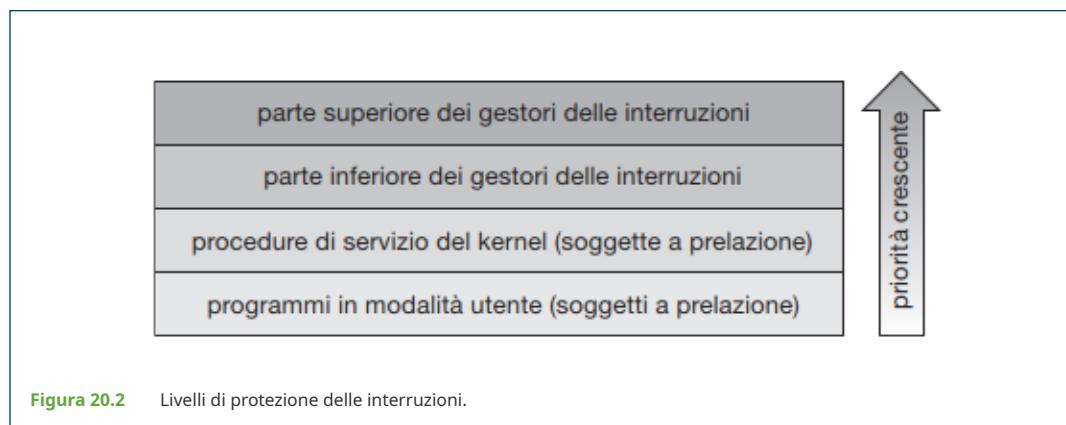
Disabilitare le interruzioni comporta tuttavia una penalizzazione: le istruzioni di abilitazione e disabilitazione delle interruzioni sono onerose nella maggior parte delle architetture hardware; inoltre ogni operazione di i/o è sospesa finché le interruzioni rimangono disabilitate, e tutti i dispositivi che attendono di essere serviti dovranno aspettare la riabilitazione delle interruzioni, cosa che ha effetti negativi sulle prestazioni. Per risolvere il problema il kernel di Linux usa un'architettura di sincronizzazione che permette l'esecuzione completa di lunghe regioni critiche senza richiedere la disabilitazione delle interruzioni. Questa possibilità torna particolarmente utile per il codice di rete: un interrupt relativo al driver di un dispositivo di rete può segnalare l'arrivo di un intero pacchetto di dati, e quindi implicare l'esecuzione di una gran quantità di codice per decodificare, instradare e inoltrare il pacchetto nell'ambito della procedura di gestione dell'interruzione.

Il sistema Linux realizza in pratica quest'architettura dividendo le procedure di gestione delle interruzioni in due parti: la parte superiore e quella inferiore. La **parte superiore** è un'ordinaria procedura di gestione delle interruzioni, ed è eseguita solo dopo la disabilitazione condizionale delle interruzioni (segnali d'interruzione dello stesso tipo o sulla stessa linea sono disabilitati, mentre altri possono essere serviti). La **parte inferiore** della procedura di gestione, invece, è eseguita, con tutte le interruzioni abilitate, da un minischeduler in grado di assicurare che le parti inferiori non s'interrompano mai fra loro; il minischeduler è automaticamente chiamato ogni volta che termina una procedura di gestione di interrupt.

Questa divisione permette al kernel di completare l'esecuzione di complesse elaborazioni come risposta a un'interruzione senza che il kernel debba preoccuparsi di essere interrotto. Se un'interruzione si verifica durante l'esecuzione di una parte inferiore, anche se l'interruzione può richiedere l'esecuzione della stessa parte inferiore, essa sarà differita fino alla terminazione di quella correntemente in esecuzione. L'esecuzione di una parte inferiore può essere interrotta da una parte superiore, ma non può mai essere interrotta da una parte inferiore simile.

L'architettura a parti è completata da un meccanismo di disabilitazione selettiva delle parti inferiori durante l'esecuzione di normale codice di kernel. Usando questo sistema il kernel può codificare facilmente le sezioni critiche. Gli interrupt handler pongono le proprie sezioni critiche nelle parti inferiori, e quando il codice di kernel ordinario vuole eseguire una sua sezione critica, può disabilitare ogni parte inferiore rilevante per evitare di essere interrotto da altre sezioni critiche. Alla fine della sezione critica il kernel riabilita le parti inferiori ed esegue quelle che, nel corso della sezione critica, sono state accodate a causa dell'esecuzione delle rispettive parti superiori.

Nella Figura 20.2 sono riassunti i vari livelli di protezione delle interruzioni all'interno del kernel; ogni livello può essere interrotto da codice eseguito a un livello superiore, ma non sarà mai interrotto da codice eseguito allo stesso livello o a livelli inferiori. Fa eccezione il codice utente: un processo utente può sempre essere sospeso da un altro processo utente a causa dello scadere del quanto di tempo.



20.5.4 Multielaborazione simmetrica

La versione del *Kernel 2.0* fu il primo kernel stabile di Linux in grado di gestire hardware di **multiprocessing simmetrico (smp)**: processi distinti si possono eseguire in parallelo su unità d'elaborazione distinte; la prima realizzazione dell'smp in Linux stabiliva che una sola unità d'elaborazione alla volta potesse eseguire codice in modalità kernel.

Nella versione 2.2 del kernel venne introdotto un solo spinlock (talvolta denominato **bkl**, da *big kernel lock*), che permetteva a più processi, eseguiti da processori diversi, di essere attivi concorrentemente nel kernel. Il bkl, però, non permetteva che un controllo poco granulare della mutua esclusione, con la conseguenza di una scarsa scalabilità su macchine con molti processori e processi. Nelle ultime versioni del kernel si è resa più scalabile l'implementazione della smp, sostituendo l'unico spinlock del kernel con lock multipli, ognuno dei quali protegge solo una piccola parte delle strutture dati del kernel. Gli spinlock in questione sono descritti nel Paragrafo 20.5.3.

I kernel 3.0 e 4.0 hanno apportato ulteriori migliorie alla smp, tra cui meccanismi di lock più fini, l'affinità per il processore e gli algoritmi di bilanciamento del carico, oltre al supporto per centinaia o migliaia di processori fisici in un unico sistema.

20.6 Gestione della memoria

Il sistema di gestione della memoria del sistema Linux ha due componenti: il primo si occupa dell'allocazione e del rilascio della memoria fisica: pagine, gruppi di pagine e piccoli blocchi di ram; il secondo si occupa della gestione della memoria virtuale, la memoria indirizzabile dai processi in esecuzione. In questo paragrafo si descrivono entrambi questi meccanismi, e si esamina poi il modo in cui i componenti caricabili di un nuovo programma sono portati all'interno della memoria virtuale di un processo in conseguenza di una chiamata di sistema `exec()`.

20.6.1 Gestione della memoria fisica

A causa di specifiche caratteristiche dell'hardware, Linux divide la memoria fisica in quattro **zone**, che rappresentano altrettante regioni della memoria. Le zone sono note come:

- ZONE_DMA
- ZONE_DMA32
- ZONE_NORMAL
- ZONE_HIGHMEM

Queste zone sono specifiche dell'architettura. Per esempio, sull'architettura Intel 80x86 alcuni dispositivi isa (*industry standard architecture*) possono accedere soltanto ai 16 mb inferiori della memoria fisica, tramite dma. Su tali sistemi, i primi 16 mb della memoria fisica comprendono ZONE_DMA. Su altri sistemi, alcuni dispositivi possono accedere solo ai primi 4 GB di memoria fisica, nonostante il supporto a indirizzi di 64 bit. In tali sistemi, i primi 4 GB di memoria fisica comprendono ZONE_DMA32.

ZONE_HIGHMEM (che sta per “memoria alta”) si riferisce alla memoria fisica che non è associata allo spazio degli indirizzi del kernel. Nell'architettura Intel a 32 bit (con uno spazio degli indirizzi, quindi, da 4 gb = 2^{32} bit), il kernel è mappato nei primi 896 mb dello spazio degli indirizzi; la memoria restante, che assume appunto il nome di **memoria alta**, è allocata da ZONE_HIGHMEM. Infine, ZONE_NORMAL comprende tutto il resto (le normali pagine, regolarmente mappate). La presenza di una data zona in un'architettura dipende dai suoi vincoli. Una moderna architettura a 64 bit come Intel x86-64 ha una piccola ZONE_DMA di 16 MB (per i dispositivi legacy) e tutto il resto della memoria è in ZONE_NORMAL, senza “memoria alta”.

La relazione tra zone e indirizzi fisici sull'architettura Intel x86-32 è mostrata dalla Figura 20.3. Il kernel mantiene una lista delle pagine libere per ciascuna zona, e soddisfa la richiesta di memoria fisica usando la zona appropriata.

zona	memoria fisica
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 ... 896 MB
ZONE_HIGHMEM	> 896 MB

Figura 20.3 Relazione fra le zone e gli indirizzi fisici nell'architettura Intel x86-32.

Il principale strumento di gestione della memoria fisica nel kernel di Linux è l'**allocatore delle pagine**: ciascuna zona ha un suo proprio allocatore, responsabile di allocazione e rilascio delle pagine fisiche, ed è in grado di allocare su richiesta gruppi di pagine fisicamente contigue. L'allocatore usa un **sistema buddy** (Paragrafo 10.8.1) per tener traccia delle pagine fisiche disponibili; un allocatore di questo tipo associa coppie di unità adiacenti di memoria assegnabile: ogni regione di memoria assegnabile ha una gemella adiacente (*buddy*, appunto), e ogni volta che due regioni gemelle allocate si liberano entrambe, vengono combinate per formare una regione più ampia, un *buddy-heap*. Anche questa più ampia regione ha una gemella con la quale potenzialmente potrà formare una regione allocabile ancora più ampia. D'altro canto, se una richiesta di una piccola regione di memoria non può essere soddisfatta assegnando una regione libera esistente, una regione libera più ampia sarà suddivisa in due regioni gemelle al fine di soddisfare la richiesta. Per tener traccia delle regioni di memoria libere di ogni specifica dimensione permessa si usano distinte liste concatenate; nel sistema Linux la più piccola dimensione assegnabile tramite questo meccanismo è di una singola pagina fisica. Nella Figura 20.4 è mostrato un esempio di allocazione *buddy-heap*: si vuole allocare una regione di 4 kb, ma la più piccola regione disponibile è di 16 kb. La regione è allora ricorsivamente divisa fino a ottenere la dimensione richiesta.

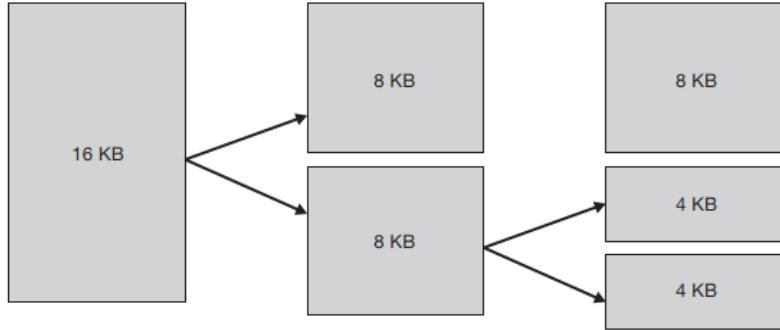
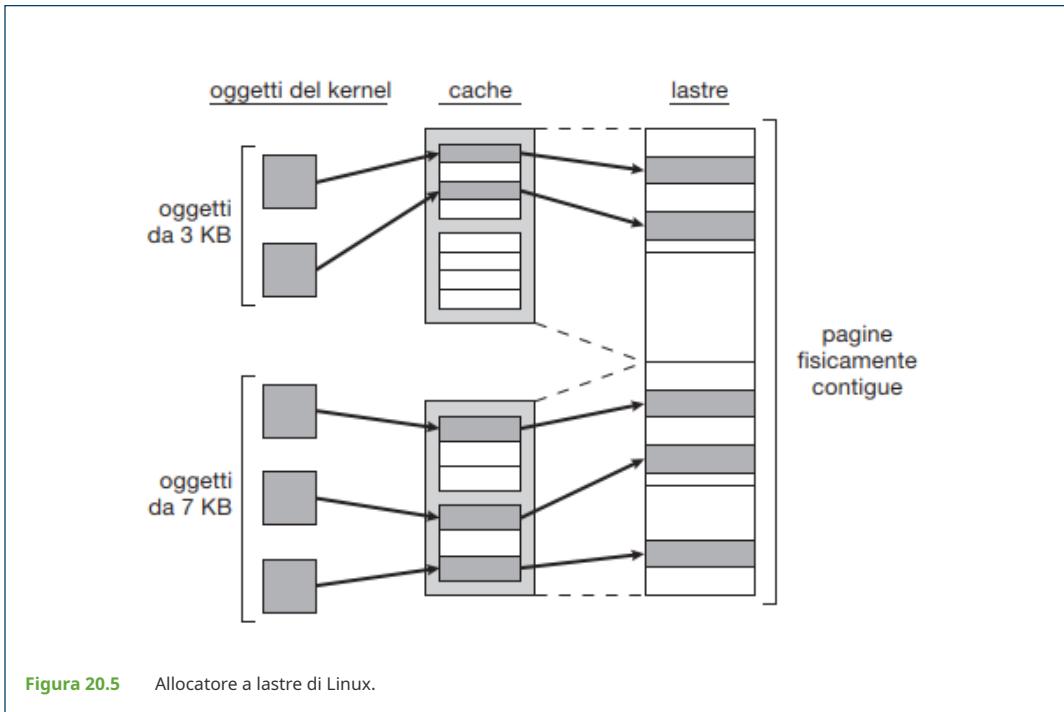


Figura 20.4 Divisione delle regioni di memoria secondo l'algoritmo *buddy-heap*.

In ultima analisi tutte le operazioni di allocazione di memoria nel kernel di Linux avvengono o staticamente, da driver che riservano aree di memoria contigua durante l'avvio del sistema, o dinamicamente tramite l'allocatore delle pagine; tuttavia le funzioni del kernel non devono necessariamente usare direttamente l'allocatore per riservare memoria: esistono molti sottosistemi specializzati di gestione della memoria che si appoggiano sull'allocatore delle pagine per gestire le loro proprie risorse di memoria. I più importanti sono quello della memoria virtuale, descritto nel Paragrafo 20.6.2, l'allocatore a lunghezza variabile `kmalloc()`, l'allocatore delle lastre, che alloggia la memoria necessaria per le strutture dati del kernel, e la cache delle pagine, che funge da cache per le pagine appartenenti ai file.

Molti componenti del sistema operativo Linux hanno bisogno di allocare su richiesta intere pagine, ma spesso c'è anche la necessità di blocchi di memoria più piccoli: il kernel fornisce un allocatore aggiuntivo per le richieste di dimensione arbitraria, in cui la dimensione di una richiesta non è nota in anticipo e potrebbe anche essere solo di qualche byte. Questo servizio è offerto dalla funzione `kmalloc()`, analoga alla `malloc()` del linguaggio C, la quale assegna su richiesta intere pagine fisiche, ma le divide poi in blocchi più piccoli. Il kernel mantiene diverse liste delle pagine usate dalla funzione `kmalloc()`, e tutte le pagine su una data lista vengono suddivise in blocchi di una fissa dimensione. L'allocazione della memoria implica in questo caso l'individuazione della lista appropriata e il successivo uso del primo elemento libero nella lista, o l'allocazione di una nuova pagina e la sua suddivisione. Le regioni di memoria richieste attraverso `kmalloc()` sono allocate in modo permanente, finché non sono esplicitamente liberate con una chiamata `kfree()`; per soppiare a carenze di memoria `kmalloc()` non può riallocare queste regioni, né revocarne l'uso.

Un altro metodo adottato da Linux per la memoria del kernel è noto come allocazione a lastre (*slab allocation*). Una **lastra** consiste di una o più pagine fisicamente contigue, usate per contenere le strutture dati del kernel. Una **cache** consiste di una o più lastre. Vi è una sola cache per ciascun tipo di struttura dati del kernel – per esempio, una cache per la struttura dati che rappresenta i descrittori dei processi, una cache dedicata agli oggetti che rappresentano file, un'altra per gli inode, e via di seguito. Ogni cache è popolata da **oggetti**, cioè istanze della struttura dati del kernel che la cache rappresenta. La cache che rappresenta gli inode, per esempio, contiene istanze di strutture inode, quella che contiene i descrittori dei processi memorizza istanze di strutture per descrittore di processo, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 20.5, che mostra oggetti del kernel da 3 e 7 kb. Questi oggetti sono memorizzati nelle rispettive cache, contenenti ciascuna oggetti da 3 kb e 7 kb.



L'algoritmo di allocazione delle lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, alcuni oggetti, inizialmente dichiarati **liberi**, sono assegnati alla cache. Il numero di questi oggetti dipende dalla grandezza della lastra corrispondente. Per esempio, una lastra che misura 12 kb (formata da tre pagine contigue di 4 kb) potrebbe contenere sei oggetti da 2 kb. All'inizio, tutti gli oggetti nella cache sono dichiarati liberi. Quando una struttura dati del kernel ha bisogno di un nuovo oggetto, l'allocatore può selezionare dalla cache qualunque oggetto libero e assegnarlo alla richiesta, marcandolo come **usato**.

Consideriamo una situazione in cui il kernel richieda all'allocatore delle lastre la memoria per un oggetto che rappresenta un descrittore di processo. Nei sistemi Linux, il descrittore ha tipo `struct task_struct`, che richiede circa 1,7 kb di memoria. Quando il kernel di Linux genera un nuovo task, richiede alla cache relativa la memoria necessaria per l'oggetto `struct task_struct`. La cache risponderà alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e che risulti libero.

In Linux, ogni lastra può presentare uno degli stati seguenti.

1. **Piena.** Tutti gli oggetti della lastra sono dichiarati usati.
2. **Vuota.** Tutti gli oggetti della lastra sono dichiarati liberi.
3. **Parziale.** La lastra contiene sia oggetti usati, sia oggetti liberi.

L'allocatore delle lastre, per soddisfare una richiesta, tenta in primo luogo di utilizzare un oggetto libero in una lastra parziale. Se non ne esistono, assegna un oggetto libero da una lastra vuota. In mancanza di lastre vuote disponibili, una nuova lastra viene creata da pagine fisiche contigue e assegnata a una cache; da tale lastra si attinge la memoria che deve essere allocata all'oggetto.

Gli altri due sottosistemi principali di Linux che attuano una gestione autonoma delle pagine fisiche sono strettamente collegati fra loro: la **cache delle pagine** e il sistema per la memoria virtuale. La cache delle pagine è la cache principale del kernel per i file e costituisce il meccanismo primario attraverso cui effettuare le operazioni di i/o relative ai dispositivi a blocchi (Paragrafo 20.8.1). Qualunque tipologia di file system, inclusi i file system nativi di Linux, basati su unità a disco, e i file system di rete nfs, impiegano la cache delle pagine. Essa memorizza pagine intere di dati contenuti nei file e il suo uso non è limitato ai dispositivi a blocchi: può anche essere usata per memorizzare dati a cui si accede dalla rete. Il sistema per la memoria virtuale gestisce i contenuti dello spazio d'indirizzi virtuali di ciascun processo. Questi due sistemi interagiscono strettamente tra loro: il caricamento di una pagina di dati nella cache delle pagine richiede la mappatura delle pagine nella cache, tramite il sistema di memoria virtuale. I paragrafi successivi illustrano nei particolari il sistema di memoria virtuale.

20.6.2 Memoria virtuale

Il sistema di memoria virtuale di Linux si occupa dello spazio d'indirizzi accessibile a ogni processo. Esso crea pagine di memoria virtuale su richiesta e gestisce il loro caricamento da disco o il loro trasferimento nell'area d'avvicendamento su disco quando è richiesto. In Linux il gestore della memoria virtuale considera lo spazio d'indirizzi di un processo da due punti di vista diversi: come insieme di regioni distinte e come insieme di pagine.

Il primo punto di vista è di natura logica, e riflette le istruzioni che il sistema per la memoria virtuale ha ricevuto riguardo all'organizzazione dello spazio d'indirizzi: quest'ultimo è visto come un insieme di regioni non sovrapposte, e ogni regione rappresenta un sottoinsieme continuo e allineato alle pagine dello spazio d'indirizzi. Ogni regione è descritta internamente da un'unica struttura dati `vm_area_struct` che ne definisce le caratteristiche, compresi i permessi di lettura, scrittura ed esecuzione del

processo, e le informazioni riguardanti eventuali file a essa associati. Le regioni riguardanti un dato spazio d'indirizzi sono organizzate in una struttura ad albero binario bilanciato che permette una rapida ricerca della regione corrispondente a un indirizzo virtuale.

Il kernel utilizza anche un secondo punto di vista riguardante lo spazio d'indirizzi, questa volta di natura fisica, rappresentata nella page table hardware legata al processo. Gli elementi della page table determinano l'esatta posizione corrente di ogni pagina della memoria virtuale, sia che essa si trovi in un disco sia che risieda nella memoria fisica; la gestione della memoria dal punto di vista fisico è realizzata da un insieme di procedure invocate dai gestori dei segnali di eccezione del kernel, ognualvolta un processo tenta di accedere a una pagina che non è in quel momento presente nelle page table. Ogni struttura `vm_area_struct` nella descrizione dello spazio d'indirizzi contiene un campo che punta a una tabella di funzioni che realizzano i servizi chiave di gestione delle pagine per ogni data regione di memoria virtuale. Tutte le richieste di operazioni relative a una pagina non disponibile sono recapitate alla fine all'appropriato gestore nella tabella di funzioni della struttura `vm_area_struct`, cosicché le procedure di gestione della memoria centrale non devono essere a conoscenza dei dettagli riguardanti la gestione di ogni possibile tipo di regione di memoria.

20.6.2.1 Regioni di memoria virtuale

Il sistema Linux opera con diversi tipi di regioni di memoria virtuale. Una prima proprietà caratterizzante un tipo di memoria virtuale è la sua memoria ausiliaria, cioè la descrizione dell'origine delle pagine; nella maggior parte dei casi, si tratta di un file o non è presente. Una regione priva di memoria ausiliaria è il tipo più semplice di memoria virtuale: essa rappresenta **memoria a valori nulli**, nel senso che quando un processo tenta di leggere una pagina di questa regione ottiene come risposta semplicemente una pagina di memoria riempita di zeri.

Una regione la cui memoria ausiliaria sia un file agisce come una finestra sui contenuti di quel file: quando un processo tenta di accedere a una pagina della regione, nella page table è scritto l'indirizzo di una pagina della cache delle pagine del kernel corrispondente all'appropriato offset all'interno del file. La stessa pagina di memoria fisica è usata sia dalla cache delle pagine sia dalle page table del processo, cosicché ogni cambiamento apportato al file dal file system è immediatamente visibile per ogni processo che abbia quel file mappato nel proprio spazio d'indirizzi. Il numero di processi che possono mappare una stessa regione dello stesso file è arbitrario, e a questo scopo ciascuno di essi userà la stessa pagina di memoria fisica.

Un'altra caratteristica di una regione di memoria virtuale è la sua reazione alle operazioni di scrittura; il mapping di una regione di memoria dallo spazio d'indirizzi di un processo può infatti essere *privato* o *condiviso*. Nel primo caso, se il processo scrive in quella regione, la procedura di paginazione rileva la necessità di una copiatura su scrittura per garantire l'effetto locale dei cambiamenti; nel secondo caso, d'altra parte, l'operazione comporta l'aggiornamento dell'oggetto associato a quella regione, in modo che i cambiamenti siano immediatamente visibili a ogni altro processo che mappa lo stesso oggetto.

20.6.2.2 Durata di uno spazio d'indirizzi virtuale

Il kernel crea un nuovo spazio d'indirizzi virtuale in due situazioni: l'avviamento di un nuovo programma tramite la chiamata di sistema `exec()`; la creazione di un nuovo processo per mezzo della chiamata di sistema `fork()`. Nel primo caso l'operazione è semplice: si assegna al nuovo processo un nuovo spazio d'indirizzi completamente vuoto; è compito delle procedure che carcano il programma preparare lo spazio d'indirizzi con regioni di memoria virtuale.

Nel secondo caso, la creazione di un nuovo processo tramite la `fork()` comporta la creazione di una copia completa dello spazio d'indirizzi virtuali esistente del processo genitore. Il kernel copia i descrittori `vm_area_struct` del processo genitore, e crea poi un nuovo insieme di tabelle delle pagine per il figlio; le tabelle del genitore sono copiate direttamente in quelle del figlio, incrementando solo il conteggio dei riferimenti a ogni pagina coinvolta: ne segue che dopo la `fork()` i processi genitore e figlio condividono le stesse pagine di memoria fisica nei loro spazi d'indirizzi.

Un caso particolare si ha quando durante l'operazione di copiatura s'incontra una regione di memoria virtuale privata; tutte le pagine appartenenti a questa regione sulle quali il processo ha scritto qualcosa sono private, e gli eventuali successivi cambiamenti apportati dal genitore o dal figlio non devono ripercuotersi sulla pagina corrispondente nello spazio d'indirizzi dell'altro processo. Durante la copiatura delle tabelle si stabilisce che le pagine in questione siano soltanto leggibili e le si contrassegna per la copiatura su scrittura: fino a che nessuno dei due processi modifica queste pagine, i due processi condividono le stesse pagine di memoria fisica, ma quando un processo tenta di scrivere in una di esse, si controlla il conteggio dei riferimenti alla pagina, e se essa è ancora condivisa il processo ne copia i contenuti in una nuova pagina di memoria fisica, usando poi questa copia in luogo dell'originale. Si tratta di un meccanismo che assicura il più a lungo possibile la condivisione tra processi di pagine di dati privati; le copie si creano solo se è assolutamente necessario.

20.6.2.3 Paginazione e avvicendamento dei processi

Uno dei compiti importanti che il sistema di memoria virtuale deve assolvere è spostare pagine di memoria dalla memoria fisica al disco quando la memoria fisica in questione è richiesta per altri scopi. I primi sistemi unix raggiungevano lo scopo spostando nei dischi i contenuti di interi processi in un'unica soluzione, ma le versioni moderne sfruttano maggiormente la paginazione, cioè il trasferimento di singole pagine di memoria virtuale dalla memoria fisica al disco e viceversa. Il sistema Linux non usa l'avvicendamento d'interi processi, ma adotta esclusivamente i più recenti meccanismi di paginazione.

Il sistema di paginazione si può dividere in due parti: l'**algoritmo di scelta** (*policy algorithm*) decide quali pagine trasferire su disco, e quando farlo; il **meccanismo di paginazione** compie il trasferimento su disco ed esegue anche l'operazione inversa non appena ve ne sia bisogno.

L'algoritmo di scelta di Linux è una versione modificata dell'algoritmo dell'orologio (con seconda chance), descritto nel Paragrafo 10.4.5.2. Il sistema Linux adotta una scansione in più passi e ogni pagina ha un'età ritoccata a ogni passo. L'età, per la precisione, è una misura della "giovinezza" di una pagina, cioè di quanto la pagina è stata usata di recente: le pagine per le quali è stato frequentemente richiesto l'accesso avranno un valore maggiore, mentre il valore delle pagine meno richieste diminuirà a ogni passo. L'algoritmo di paginazione sceglie per il trasferimento le **pagine meno frequentemente usate** (*least frequently used*, lfu).

Il meccanismo di paginazione è in grado di paginare sia in specifici dispositivi e partizioni di swapping, sia in normali file, anche se quest'ultima operazione è assai più lenta a causa dei rallentamenti indotti dal file system. L'allocazione di blocchi che risiedono in dispositivi di swapping è eseguita attraverso una bitmap dei blocchi usati che si trova sempre nella memoria fisica. L'allocatore adotta un algoritmo *next-fit* che tenta di scrivere le pagine in successioni ininterrotte di blocchi del disco al fine di migliorare le prestazioni. L'allocatore registra che una pagina è stata trasferita nel disco usando una caratteristica delle page table delle moderne cpu: il bit di pagina assente dell'elemento della page table è posto a uno, il che permette al resto dell'elemento della tabella di contenere un indice che identifica il luogo in cui la pagina è stata trasferita.

20.6.2.4 Memoria virtuale del kernel

Il sistema Linux riserva per il proprio uso una regione costante e dipendente dall'architettura dello spazio d'indirizzi virtuali di ogni processo. Gli elementi della tabella delle pagine che si riferiscono a queste pagine del kernel sono contrassegnati come protetti, cosicché le pagine non sono né visibili né modificabili quando la cpu è in esecuzione in modalità utente. Quest'area di memoria virtuale del kernel è costituita da due regioni. La prima è statica e contiene riferimenti a ogni pagina di memoria fisica disponibile nel sistema, fornendo un semplice modo di tradurre indirizzi fisici in indirizzi virtuali durante l'esecuzione di codice di kernel. La parte centrale del kernel insieme con tutte le pagine assegnate dall'ordinario allocatore delle pagine risiedono in questa regione.

Il resto della parte riservata al kernel dello spazio d'indirizzi non è dedicato ad alcuno scopo specifico: gli elementi della tabella delle pagine che si riferiscono a questa regione possono essere modificati dal kernel in modo da puntare a qualunque altra area di memoria desiderata. Il kernel fornisce due funzioni che permettono al codice kernel di usare questa memoria virtuale: `vmalloc()` assegna un numero arbitrario di pagine di memoria fisica, non necessariamente contigue, e le associa a un'unica regione contigua della memoria virtuale del kernel. La funzione `vremap()` associa una sequenza d'indirizzi virtuali a un'area di memoria usata da un driver di dispositivo per compiere l'i/o memory mapped.

20.6.3 Caricamento ed esecuzione dei programmi utente

L'esecuzione dei programmi utente da parte del kernel di Linux si attiva per mezzo della chiamata di sistema `exec()`, la quale chiede al kernel di eseguire un nuovo programma all'interno del processo corrente in modo tale da sovrascrivere integralmente il contesto d'esecuzione attuale con il contesto iniziale del nuovo programma. Il primo compito di questo servizio di sistema è verificare che il processo chiamante abbia diritto d'esecuzione sul file interessato; risolta tale questione, il kernel invoca una procedura di caricamento per avviare l'esecuzione del programma. Il caricatore non trasferisce necessariamente i contenuti del file da eseguire nella memoria fisica, ma per lo meno mappa il programma in memoria virtuale.

In Linux non c'è una singola procedura per il caricamento dei programmi: il sistema operativo mantiene invece una tabella dei possibili caricatori, e dà a ognuno di essi l'opportunità di tentare di caricare il file in questione al momento dell'esecuzione di una `exec()`. Il motivo originario dell'esistenza di una tale tabella era che nel passaggio dalla versione 1.0 alla versione 1.2 del kernel il formato dei file binari di Linux è stato modificato: i primi kernel adottavano il formato a.out, che è relativamente semplice e comune ai vecchi sistemi unix; i più recenti sistemi Linux usano il più moderno formato elf, adottato adesso dalla maggior parte delle attuali versioni dello unix. Il formato elf offre un certo numero di vantaggi rispetto ad a.out, fra i quali si segnalano la flessibilità e l'estendibilità: si possono aggiungere nuove sezioni a un file binario elf (per esempio per fornire ulteriori informazioni utili al debugging), senza che le procedure di caricamento vadano incontro a problemi. Grazie alla registrazione di più procedure di caricamento il sistema Linux gestisce facilmente i formati binari elf e a.out all'interno di un unico sistema.

Nei Paragrafi 20.6.3.1 e 20.6.3.2 sono trattati esclusivamente il caricamento e l'esecuzione di programmi in formato elf: le procedure di caricamento concernenti il formato a.out sono simili anche se più semplici.

20.6.3.1 Mappatura dei programmi in memoria

Il caricamento di un file binario nella memoria fisica non è eseguito dal caricatore binario; le pagine del file binario sono invece associate a regioni della memoria virtuale: solo quando il programma tenterà di accedere a una data pagina, la conseguente eccezione di page fault causerà il caricamento di quella pagina nella memoria fisica per mezzo della paginazione su richiesta.

L'iniziale associazione del file a regioni di memoria virtuale è compito del caricatore binario del kernel. Un file in formato binario elf consiste in un'intestazione seguita da diverse sezioni allineate alle pagine: il caricatore elf lavora leggendo l'intestazione e mappando le sezioni del file in regioni distinte della memoria virtuale.

Nella Figura 20.6 è mostrata l'organizzazione tipica delle regioni di memoria preparate dal caricatore elf. Il kernel si trova nella regione riservata a un estremo dello spazio d'indirizzi; si tratta di una regione privilegiata di memoria virtuale inaccessibile agli ordinari programmi eseguiti in modalità utente. Il resto della memoria virtuale è disponibile per le applicazioni che possono usare le funzioni del kernel che creano regioni associate a porzioni di un file o disponibili per i dati delle applicazioni.

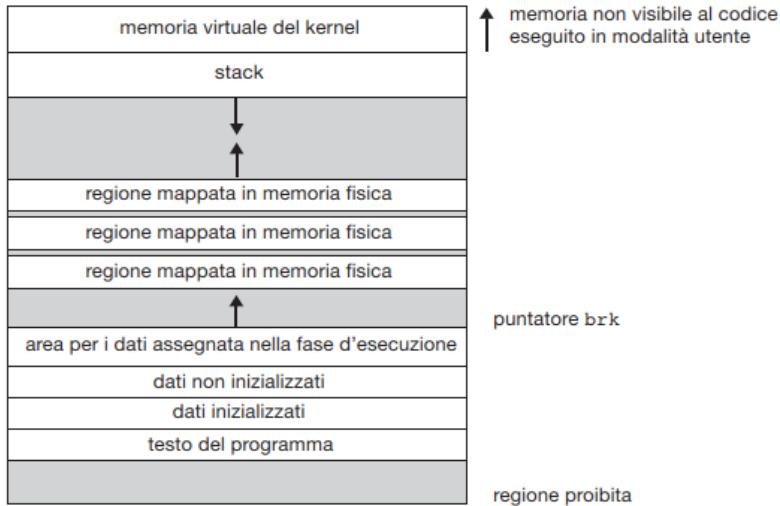


Figura 20.6 Organizzazione della memoria per i programmi elf.

Il compito del caricatore è di instaurare le associazioni iniziali per permettere l'avvio dell'esecuzione del programma: fra le regioni da inizializzare ci sono lo stack e i segmenti di testo e di dati del programma.

Lo stack è posto in cima alla memoria virtuale utente, e cresce verso il basso impiegando indirizzi via via inferiori. Comprende copie delle variabili d'ambiente che costituiscono gli argomenti passati al programma dalla chiamata di sistema `exec()`. Le altre regioni sono poste vicino all'estremo inferiore della memoria virtuale; le sezioni del file binario che contengono il testo del programma o dati solamente leggibili sono assegnate a regioni protette dalla scrittura. Sono poi mappati nella memoria virtuale i dati scrivibili inizializzati e infine i dati non inizializzati, che sono mappati in regioni private a valori nulli.

Appena sopra queste regioni di dimensione fissa si trova una regione a dimensione variabile che i programmi possono espandere secondo le necessità al fine di memorizzare dati allocati nella fase d'esecuzione. Ogni processo ha un puntatore `brk` che punta all'estensione attuale di questa regione di dati, e un processo può ampliare o contrarre la sua regione relativa a `brk` con una singola chiamata di sistema `sbrk()`.

Una volta completate queste operazioni, il caricatore inizializza il contatore di programma del processo secondo il punto d'inizio registrato nell'intestazione elf, e il processo è pronto per lo scheduling.

20.6.3.2 Collegamento statico e dinamico

Una volta che il programma è stato caricato e avviato, tutti i dati necessari contenuti nel file binario sono stati trasferiti nello spazio d'indirizzi virtuale del processo; tuttavia, la maggior parte dei programmi deve eseguire funzioni delle librerie di sistema, e anche queste devono essere caricate. Nel caso più semplice le funzioni di libreria necessarie sono direttamente incorporate nell'eseguibile binario del programma: si dice in questo caso che il programma è collegato staticamente alle librerie, e programmi di questo tipo possono cominciare l'esecuzione non appena caricati.

Lo svantaggio principale del collegamento statico è che ogni programma deve contenere le proprie copie delle stesse funzioni di libreria: sia in termini di memoria fisica sia di spazio di disco è più efficiente caricare in memoria le librerie di sistema una sola volta. Il collegamento dinamico permette di operare in questo modo.

Il sistema Linux realizza il collegamento dinamico in modalità utente grazie a una speciale libreria di collegamento. Ogni programma che utilizza il collegamento dinamico contiene una piccola funzione collegata staticamente che è chiamata all'avviamento del programma. Questa funzione non fa altro che mappare la libreria di collegamento nella memoria virtuale ed eseguire il codice contenuto nella funzione; la libreria di collegamento determina l'elenco delle librerie dinamiche richieste dal programma e i nomi delle variabili e delle funzioni di libreria che il programma richiede, leggendo alcune sezioni del file binario elf. La libreria di collegamento mappa poi le librerie richieste nella memoria virtuale, e risolve i riferimenti ai simboli contenuti in queste librerie. Non è importante l'esatto punto della memoria dove risiedono queste librerie condivise: si tratta di codice compilato in modo d'essere **codice indipendente dalla posizione** (*position independent code, pic*), che si può eseguire a partire da qualunque indirizzo di memoria.

20.7 File system

Linux adotta il modello standard di file system di unix. In unix un file non è necessariamente un oggetto memorizzato in un disco o prelevato da un file server di rete: un file è qualunque elemento sia in grado di gestire l'immissione o l'emissione di un flusso di dati. I driver dei dispositivi possono apparire come file agli occhi dell'utente, e così pure i canali di comunicazione fra processi o le connessioni di rete.

Il kernel di Linux gestisce tutti questi tipi di file nascondendone i dettagli relativi alla struttura interna sotto uno strato software: il file system virtuale (vfs). Di seguito ne diamo una presentazione, e in seguito analizziamo file system standard di Linux: *ext3*.

20.7.1 File system virtuale

Il vfs di Linux è concepito secondo principi di progettazione orientata agli oggetti. È costituito da due componenti: un insieme di definizioni che stabiliscono che cosa può essere un oggetto di tipo file, e uno strato software che serve a manipolare questi oggetti. Il vfs definisce quattro tipi principali di oggetti.

- L'**oggetto inode**, che rappresenta un file singolo.
- L'**oggetto file**, che rappresenta un file aperto.
- L'**oggetto superblocco**, che rappresenta un intero file system.
- L'**oggetto dentry**, che rappresenta una singola voce di una directory.

Per ciascuno dei quattro tipi di oggetti menzionati il vfs definisce un insieme di operazioni: ogni oggetto contiene un puntatore a una tabella di funzioni, e questa elenca gli indirizzi delle funzioni che realizzano le operazioni richieste per l'oggetto in questione. Per esempio, una api abbreviata per alcune operazioni relative agli oggetti file prevede.

- `int open(...)` – Apre un file.
- `ssize_t read(...)` – Legge da un file.
- `ssize_t write(...)` – Scrive su un file.
- `int mmap(...)` – Mappa un file in memoria.

La definizione completa dell'oggetto file è specificata nella `struct file_operations`, che si trova nel file `/usr/include/linux/fs.h`. Un'implementazione dell'oggetto file (relativa a un tipo di file specifico) deve realizzare ciascuna funzione specificata dalla definizione dell'oggetto file.

Lo strato di programmi del vfs può quindi eseguire un'operazione su uno di questi oggetti chiamando una funzione appropriata scelta dalla tabella delle funzioni dell'oggetto, senza dover sapere in anticipo esattamente con che tipo d'oggetto ha a che fare: il vfs non sa né ha bisogno di sapere se un *inode* rappresenta un file di rete, un file in un disco, una directory o una socket di rete; in ogni caso, la funzione che realizza l'operazione `read()` per quell'oggetto si trova in un punto ben determinato e invariabile della tabella delle funzioni dell'oggetto, e lo strato software del vfs richiamerà la funzione in questione senza interessarsi del modo in cui i dati saranno effettivamente letti.

Gli oggetti *inode* e gli oggetti file costituiscono il meccanismo d'accesso ai file. Un oggetto *inode* è una struttura dati contenente puntatori ai blocchi del disco dove si trova l'effettivo contenuto del file; un oggetto file rappresenta un punto d'accesso ai dati in un file aperto. Un processo non può quindi avere accesso ai dati contenuti in un file relativo a un *inode* se non ottiene prima un oggetto file corrispondente a tale *inode*. L'oggetto file tiene traccia del punto del file nel quale il processo sta correntemente leggendo o scrivendo, cosa che permette di gestire l'i/o sequenziale sui file; esso registra inoltre i permessi (ad esempio di lettura e scrittura) richiesti al momento dell'apertura del file e tiene traccia dell'attività del processo quando ciò è necessario per eseguire letture anticipate adattative (cioè il trasferimento anticipato di dati in memoria al fine di migliorare le prestazioni).

Di solito gli oggetti file appartengono a un solo processo, mentre gli oggetti *inode* sono condivisi. Esiste un oggetto file per ogni istanza di un file aperto, ma l'oggetto *inode* è sempre unico. Anche quando un certo file non è più usato da nessun processo il suo *inode* può essere mantenuto nella cache dal vfs per ottenere migliori prestazioni nell'ipotesi di un prossimo accesso al file relativo. Tutti i dati del file mantenuti nella cache sono organizzati in una lista all'interno dell'oggetto *inode* del file; l'*inode* registra anche alcune informazioni standard sul file, per esempio il proprietario, la dimensione e la data dell'ultima modifica.

Le directory sono trattate in modo leggermente diverso dagli altri file; l'interfaccia di programmazione unix definisce un certo numero di operazioni eseguibili sulle directory, come la creazione, rimozione o il cambiamento di nome di un file contenuto in una directory: al contrario delle operazioni di lettura e scrittura, che necessitano della preliminare apertura del file, le chiamate di sistema relative a queste operazioni non richiedono l'apertura da parte dell'utente dei file coinvolti. Il vfs perciò definisce queste operazioni sulle directory all'interno dell'oggetto *inode*, e non nell'oggetto file.

L'oggetto superblocco rappresenta un insieme di file correlati, che formano un file system completo e autosufficiente. Il kernel del sistema operativo mantiene un singolo oggetto superblocco per ogni unità a disco montata come file system e per ciascun file system di rete attualmente connesso. Il compito primario dell'oggetto superblocco è di fornire accesso agli *inode*. Il vfs identifica ciascun *inode* tramite una coppia univoca (file system/numero di *inode*), e rileva l'*inode* corrispondente a un dato numero di *inode* richiedendo all'oggetto superblocco la restituzione dell'*inode* associato a tale numero.

Infine, un oggetto *dentry* rappresenta una voce di directory che può includere il nome della directory nel percorso di un file (come `/usr`) oppure il file medesimo (per esempio `stdio.h`). Il file `/usr/include/stdio.h`, per esempio, contiene le voci di directory (1) `/`, (2) `usr`, (3) `include` e (4) `stdio.h`. Ciascuno di questi elementi è rappresentato da un oggetto *dentry* distinto.

Si consideri l'esempio seguente. Un processo intende aprire il file il cui percorso è `/usr/include/stdio.h`, servendosi di un editor. Poiché Linux tratta i nomi di directory alla stregua di file, la traduzione di questo percorso richiede che si ottenga, in via preliminare, l'*inode* associato alla radice, `/`. Il sistema operativo deve quindi leggere il file corrispondente per ottenere l'*inode* relativo al file `usr`, e deve ripetere tale processo fino a ottenere l'*inode* per il file `stdio.h`. Dal momento che la traduzione dei percorsi può richiedere

molto tempo, Linux mantiene una cache per gli oggetti *dentry*, da consultarsi durante la traduzione dei path name. Estrarre un *inode* dalla cache degli oggetti *dentry* è un processo decisamente più veloce della lettura di file su disco.

20.7.2 File system *ext3*

Il file system residente su dischi ordinariamente usato dal Linux si chiama *ext3* per ragioni storiche. Originariamente Linux adottava un file system compatibile con quello del sistema Minix allo scopo di facilitare lo scambio di dati con la piattaforma di sviluppo Minix; quel file system, tuttavia, soffriva di gravi limitazioni quali la massima lunghezza dei nomi dei file, pari a 14 caratteri, e la massima dimensione del file system, pari a 64 mb. Il file system del Minix fu sostituito da un nuovo file system chiamato *extfs* (*extended file system*); successive modifiche apportate al fine di migliorare le prestazioni e la scalabilità e aggiungere qualche servizio mancante portarono all'*ext2* (*second extended file system*). Ulteriori sviluppi hanno aggiunto la funzionalità di journaling e il sistema è stato ribattezzato come *ext3* (*third extended file system*). Gli sviluppatori del kernel Linux stanno lavorando su un'evoluzione di *ext3* con l'aggiunta di moderne caratteristiche, come le aree di memorizzazione contigue chiamate extent. Questo nuovo file system si chiama **ext4** (*fourth extended file system*). Nel resto di questo paragrafo verrà trattato *ext3*, perché è il file system di Linux più diffuso. La maggior parte delle nostre considerazioni vale anche per *ext4*.

Il file system *ext3* ha molto in comune con l'*FFS* (*fast file system*) del sistema bsd (Appendice C, Paragrafo C.7.7, reperibile sulla piattaforma MyLab); adotta un meccanismo simile per individuare i blocchi di dati appartenenti a un certo file, memorizzando puntatori a blocchi di dati in blocchi indiretti sparsi per il file system, impiegando fino a tre livelli di indirezione. Come nell'*FFS*, le directory sono memorizzate nei dischi esattamente come ogni altro file, anche se i loro contenuti sono interpretati diversamente: ogni blocco di una directory consiste di una lista concatenata d'elementi ognuno contenente la lunghezza dell'elemento stesso, il nome di un file e il numero dell'*inode* cui l'elemento si riferisce.

Le differenze principali tra l'*ext3* e l'*FFS* riguardano le strategie d'allocazione dello spazio dei dischi: nell'*FFS* lo spazio nei dischi si assegna ai file in blocchi di 8 kb, con blocchi che possono essere ulteriormente suddivisi in frammenti di 1 kb al fine di memorizzare piccoli file o blocchi riempiti solo parzialmente alla fine di un file; per contro, l'*ext3* non usa affatto i frammenti ma esegue tutte le assegnazioni secondo unità più piccole. La dimensione di default di un blocco nell'*ext3* varia in funzione della dimensione complessiva del file system. Le dimensioni supportate sono 1, 2, 4 e 8 kb.

Per ottenere buone prestazioni il sistema operativo deve tentare di eseguire trasferimenti di i/o in unità di grandi dimensioni, ognualvolta ciò sia possibile, raggruppando richieste di i/o fisicamente adiacenti. Il raggruppamento riduce l'overhead per ciascuna richiesta indotto dai driver dei dispositivi, dai dischi e dal controller del disco. Trasferimenti di i/o delle dimensioni di un blocco sono troppo piccoli per garantire buone prestazioni, quindi l'*ext3* adotta strategie d'allocazione concepite per collocare blocchi di un file logicamente adiacenti in blocchi fisicamente adiacenti nel disco, rendendo così possibile l'accorpamento di molte richieste relative a blocchi differenti in un'unica operazione di i/o.

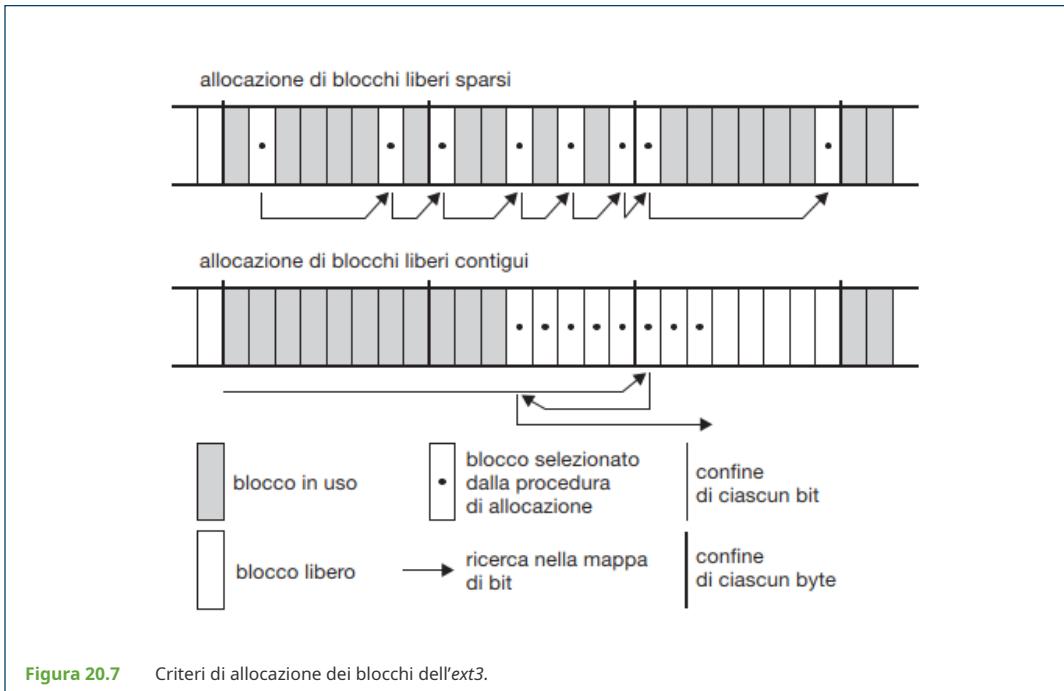
La politica di allocazione di *ext3* funziona come segue. Come nell'*FFS*, un file system *ext3* è suddiviso in segmenti chiamati **gruppi di blocchi**; l'*FFS* usa il simile concetto di **gruppi di cilindri**, dove ogni gruppo corrisponde a un singolo cilindro del disco fisico. Si noti che la tecnologia delle moderne unità a disco distribuisce i settori nei dischi con diverse densità, e quindi anche con diverse dimensioni dei cilindri, secondo la distanza dal centro del disco, perciò gruppi di cilindri di dimensione costante non corrispondono necessariamente alla geometria del disco.

Prima di allocare lo spazio a un file, l'*ext3* deve scegliere un gruppo di blocchi per quel file. Per quel che riguarda i blocchi di dati tenta di scegliere lo stesso gruppo di blocchi nel quale è stato allocato l'*inode* del file; per allocare un *inode* che non sia relativo a una directory, sceglie lo stesso gruppo di blocchi della directory cui il file appartiene. Le directory, invece di essere raggruppate, sono distribuite su tutti i blocchi disponibili. Queste strategie hanno l'obiettivo di mantenere informazioni correlate all'interno di uno stesso gruppo di blocchi, ma anche di distribuire il carico d'informazioni su tutti i gruppi di blocchi del disco, così da ridurre la frammentazione d'ogni area del disco stesso.

All'interno di un gruppo di blocchi, l'*ext3* tenta di compiere le assegnazioni in modo fisicamente contiguo se è possibile, riducendo quindi la frammentazione. Esso mantiene una bitmap di tutti i blocchi liberi di un gruppo di blocchi: durante l'allocazione dei primi blocchi relativi a un nuovo file, l'*ext3* comincia la ricerca di un blocco libero dall'inizio del gruppo di blocchi; quando invece si tratta di ampliare un file, continua la ricerca dall'ultimo blocco assegnato al file. La ricerca si svolge in due fasi: innanzi tutto si cerca un intero byte libero nella mappa di bit; se ciò non riesce, si cerca un bit libero. La ricerca di interi byte liberi ha lo scopo di tentare di allocare lo spazio nei dischi in porzioni di almeno 8 blocchi.

Una volta che si sia identificato un blocco libero, si estende la ricerca all'indietro fino a incontrare un blocco assegnato. Quando si trova un byte libero nella mappa di bit, quest'estensione a ritroso impedisce che l'*ext3* lasci un buco fra l'ultimo blocco assegnato nel precedente byte non nullo e il byte libero trovato; una volta individuato il blocco da allocare tramite una ricerca per byte o per bit, l'*ext3* estende l'allocazione in avanti fino a un massimo di otto blocchi **preassegnando** questo spazio supplementare al file. La preallocazione concorre a diminuire il grado di frammentazione nel corso di scritture intercalate in file diversi, e riduce anche il costo dell'allocazione in termini d'impegno della cpu grazie all'allocazione simultanea di più blocchi; i blocchi preassegnati sono nuovamente riportati sulla bitmap dei blocchi liberi al momento della chiusura del file.

Nella Figura 20.7 sono illustrati i criteri d'allocazione. Ogni riga rappresenta una sequenza di bit posti a uno o a zero nella bitmap di allocazione; questi bit contrassegnano i blocchi liberi e i blocchi in uso del disco. La prima possibilità è che si trovino blocchi liberi sufficientemente vicini al punto d'inizio della ricerca, in tal caso essi sono assegnati indipendentemente dal livello di frammentazione; infatti, dato che i blocchi sono vicini fra loro, è probabile che possano essere letti senza dover eseguire operazioni di seek nel disco, cosa che compensa parzialmente la frammentazione. Inoltre, allocare tutti questi blocchi a un solo file è una scelta che a lungo termine si rivela migliore dell'allocazione di blocchi isolati a file distinti, quando a lungo andare le regioni ampie di spazio libero su disco divengono rare. La seconda possibilità è che non si sia trovato un blocco libero nelle vicinanze del punto d'inizio della ricerca, quindi si procede nella ricerca di un intero byte libero nella bitmap; se si assegnasse questo byte senza prendere alcuna precauzione si creerebbe un frammento di spazio libero prima di esso, quindi è necessario tornare indietro per colmare lo spazio rimasto fra l'ultimo blocco assegnato precedentemente e il blocco in questione. Infine, per rispettare la dimensione predefinita di un'allocazione, che è di otto blocchi, si estende l'allocazione in avanti fin quando è necessario.



20.7.3 Annotazione delle modifiche (journaling)

Il file system ext3 supporta l'**annotazione delle modifiche (journaling)**, ovvero le modifiche effettuate nel file system sono trascritte, in modo sequenziale, in un log (detto *journal*). L'insieme di operazioni che esegue un compito specifico è detto **transazione**. Una volta trascritte le modifiche nel log, la transazione relativa si considera completata (*committed*). Una chiamata di sistema che modifica il file system (per esempio, `write()`) può dunque restituire il controllo al processo utente, permettendogli di proseguire con l'esecuzione. Nel frattempo, le voci del journal inerenti alla transazione sono replicate nelle strutture del file system reale: mentre sono eseguite queste modifiche, un puntatore viene aggiornato per indicare quali azioni sono complete e quali ancora da completare. Quando un'intera transazione è completata, viene rimossa dal journal. Quest'ultimo, che consiste in un buffer circolare, può risiedere in una sezione separata del file system, o anche in un disco dedicato. È più efficiente, ma più complesso, collocarlo sotto testine di lettura e di scrittura dedicate, diminuendo così le possibilità di contesa delle testine e il tempo di seek.

Se il sistema subisce un crash, nel journal rimarranno alcune transazioni. Le transazioni presenti non sono mai state completate nel file system; ma, poiché il sistema operativo le considera compiute con successo, esse devono essere completate. Le transazioni possono essere eseguite a partire dalla posizione corrente del puntatore sino al termine del lavoro, e le strutture del file system rimarranno coerenti. L'unico problema sorge nell'eventualità che una transazione fallisca: in altri termini, quando essa non è completata con successo prima del crash del sistema. Tutte le modifiche introdotte dalle transazioni fallite devono essere annullate, onde preservare la coerenza del sistema. Questo ripristino è tutto ciò che bisogna fare, in seguito a un crash, e ciò elimina i problemi di verifica della coerenza.

I file system con annotazioni delle modifiche possono anche essere più veloci di quelli che ne sono privi, visto che le modifiche risultano molto più veloci quando sono applicate al journal in memoria anziché direttamente alle strutture dati su disco. La ragione di questo miglioramento risiede nella maggior efficienza che l'esecuzione di i/o ad accesso sequenziale permette, rispetto agli accessi casuali. Le onerose operazioni di scrittura sincrone ad accesso casuale sul file system, sono convertite in operazioni molto meno dispendiose di scrittura sincrona sequenziale nel journal. I cambiamenti determinati da tali operazioni si riportano successivamente in modo asincrono nelle strutture appropriate, attraverso operazioni di scrittura ad accesso casuale. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file. Per sfruttare questo miglioramento delle prestazioni, ext3 può essere configurato per annotare solo i metadati e non i file di dati.

20.7.4 Il process file system di Linux

Il file system `/proc` è sufficientemente flessibile da permettere l'implementazione di un file system che non archivia in modo permanente alcun dato, ma semplicemente funge da interfaccia per qualche altro servizio. Il **process file system** di Linux, noto come `/proc`, è un esempio di file system i cui contenuti non sono in realtà memorizzati in alcun luogo, ma sono calcolati on demand in seguito alle richieste di i/o degli utenti.

Un file system `/proc` non è una caratteristica esclusiva di Linux: unix svr4 adottava un file system di questo tipo come interfaccia efficiente per le funzioni di debugging dei processi del kernel; ogni sottodirectory del file system corrispondeva non a una directory in qualche disco, ma piuttosto a un processo attivo del sistema, cosicché un elenco dei contenuti del file system presentava una directory per processo, essendo il nome della directory la rappresentazione decimale, in formato ascii, dell'identificatore unico del processo (pid).

Il sistema Linux implementa un file system di questo tipo, ma lo amplia notevolmente aggiungendo un certo numero di ulteriori directory e file di testo sotto la directory radice del file system. Questi elementi corrispondono a diverse statistiche relative al kernel e ai driver caricati; il file system `/proc` fornisce ai programmi la possibilità di accedere a queste informazioni leggendo semplici file di testo che nell'ambiente utente standard di unix è possibile trattare con strumenti particolarmente efficaci. Per fare un esempio, il classico comando `ps` di unix, che elenca gli stati di tutti i processi in esecuzione, è stato in passato realizzato come un processo privilegiato che leggeva gli stati dei processi direttamente dalla memoria virtuale del kernel; in Linux, questo comando è realizzato da un programma del tutto ordinario che semplicemente esamina e compone in forma maggiormente leggibile le informazioni contenute nel `/proc`.

Il file system `/proc` deve implementare una struttura di directory e i contenuti dei file in esso residenti. Giacché un file system unix è per definizione un insieme di *inode* relativi a file e directory identificati dal loro numero di *inode*, il file system `/proc` deve definire in modo unico un numero di *inode* permanente per ogni directory e per i file in essa contenuti. Una volta instaurata questa corrispondenza si può usare il numero di *inode* per identificare esattamente l'operazione richiesta da un utente che tenti di leggere un certo file o di esaminare una certa directory. Quando si leggono dati da uno di questi file, il file system `/proc` estrae l'informazione appropriata, la compone in forma testuale e la colloca nel buffer di lettura del processo richiedente.

La corrispondenza fra un numero di *inode* e il tipo d'informazione divide il numero di *inode* in due parti: nel sistema Linux un pid è lungo 16 bit, ma un numero di *inode* è lungo 32 bit; i primi 16 bit del numero di *inode* sono interpretati come un pid, e i rimanenti definiscono il tipo d'informazione richiesta in merito al processo.

Un pid pari a zero non è ammesso, quindi un campo pid pari a zero nel numero di *inode* significa che questo *inode* contiene informazioni globali e non relative a uno specifico processo: `/proc` contiene file globali distinti che forniscono informazioni come la versione del kernel, la memoria libera, statistiche sulle prestazioni, e i driver attualmente attivi.

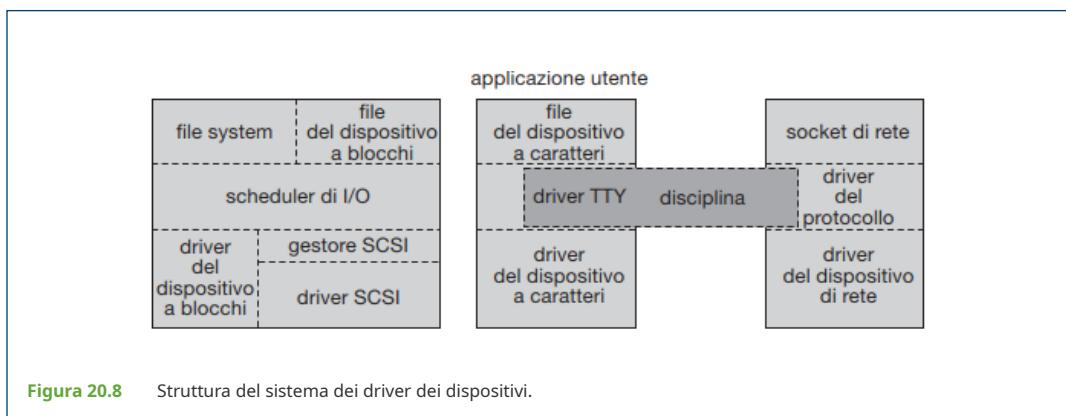
Non tutti i numeri di *inode* di questo tipo sono riservati; il kernel può allocare nuovi *inode* del `/proc` dinamicamente, mantenendo una mappa dei numeri di *inode* assegnati. Esso mantiene anche una struttura dati ad albero degli elementi globali del file system `/proc` che sono stati registrati: ogni elemento contiene il numero di *inode* del file, il nome del file, i permessi d'accesso e le funzioni speciali usate per generare i contenuti del file; i driver possono registrare o rimuovere elementi da quest'albero in qualunque momento, e una sezione speciale dell'albero che appare nella directory `/proc/sys` è riservata alle variabili del kernel. I file contenuti in quest'albero sono trattati per mezzo di un insieme di procedure di gestione comuni che permettono sia la lettura sia la scrittura di queste variabili, cosicché un amministratore del sistema può regolare i valori dei parametri del kernel semplicemente scrivendoli in ascii nel file appropriato.

Per permettere alle applicazioni di accedere efficientemente a queste variabili, il sottoalbero `/proc/sys` è reso disponibile tramite una speciale chiamata di sistema, `sysctl()`, che legge e scrive le stesse variabili in formato binario anziché sotto forma di testo, evitando i ritardi indotti dal file system; questa chiamata di sistema non fornisce una funzione aggiuntiva ma legge semplicemente l'albero dinamico degli elementi di `/proc` per decidere a quali variabili l'applicazione si riferisce.

20.8 Input e output

Per l'utente, il sistema per l'i/o di Linux appare molto simile a quello di un qualunque sistema unix: tutti i driver dei dispositivi, nei limiti del possibile, sono rappresentati come file ordinari. L'utente apre un canale d'accesso a un dispositivo nello stesso modo in cui apre un qualunque file: i dispositivi compaiono come oggetti del file system. L'amministratore del sistema può creare file speciali all'interno del file system che contengono riferimenti a uno specifico driver di dispositivo, e un utente che apra un tale file potrà leggere e scrivere nel dispositivo associato. Grazie all'ordinario sistema di protezione dei file, che stabilisce chi può accedere a quali file, l'amministratore può controllare l'accesso ai dispositivi.

Il sistema Linux suddivide i dispositivi in tre classi: dispositivi a blocchi, dispositivi a caratteri, e dispositivi di rete. Nella Figura 20.8 è illustrata la struttura globale del sistema dei driver dei dispositivi.



I **dispositivi a blocchi** comprendono tutti i dispositivi capaci di fornire l'accesso casuale a blocchi di dati di dimensione fissa completamente indipendenti; fra questi vi sono i dischi magneticici, i floppy disk, i cd-rom, i Blu-ray e le memorie flash. I dispositivi a blocchi sono generalmente usati per contenere interi file system, ma c'è anche la possibilità di accedere direttamente a questi dispositivi per permettere a programmi specializzati di creare e riparare i file system in essi contenuti. Anche le applicazioni possono accedere direttamente a questi dispositivi se lo desiderano; una base di dati, per esempio, potrebbe voler mappare i dati nei dischi in modo appropriato ai propri scopi specifici, invece di usare il file system generale.

I **dispositivi a caratteri** comprendono la maggior parte degli altri dispositivi, come mouse e tastiere. La differenza fondamentale tra i dispositivi a blocchi e quelli a caratteri è data dall'accesso casuale: ai primi, infatti, si può accedere in modo casuale, mentre i secondi sono ad accesso seriale. La ricerca di una certa posizione all'interno di un file, per esempio, potrebbe essere prevista per un dvd, ma non avrebbe senso nel caso di un mouse.

I **dispositivi di rete** sono trattati diversamente dai dispositivi delle altre due classi: gli utenti non possono trasferire direttamente dati ai dispositivi di rete, ma devono comunicare indirettamente tramite il sottosistema di rete del kernel. L'interfaccia ai dispositivi di rete è illustrata separatamente nel Paragrafo 20.10.

20.8.1 Dispositivi a blocchi

I dispositivi a blocchi costituiscono l'interfaccia principale a tutte le unità a disco del sistema. Le prestazioni sono una questione di particolare rilevanza per i dischi, e il sistema per i dispositivi a blocchi deve quindi fornire servizi che permettano di accedere ai dischi il più rapidamente possibile. Questa funzionalità si ottiene attraverso lo scheduling delle operazioni di i/o.

Per i dispositivi a blocchi, un **blocco** rappresenta l'unità sulla base della quale il kernel esegue le operazioni di i/o. Quando un blocco è caricato in memoria, è memorizzato in un **buffer**. Il **gestore delle richieste** è lo strato di software incaricato di gestire lettura e scrittura dei contenuti del buffer, da e verso il driver del dispositivo a blocchi.

Una lista distinta di richieste è mantenuta per ogni driver di un dispositivo a blocchi. Tradizionalmente lo scheduling di queste richieste avviene mediante l'algoritmo dell'ascensore (c-scan) che sfrutta l'ordine nel quale le richieste sono inserite o rimosse dalle liste. Le liste di richieste sono mantenute in ordine crescente di settore iniziale. Quando una richiesta è selezionata per essere elaborata da un driver, non è rimossa dalla lista; la rimozione avviene solo quando l'operazione di i/o è terminata: a questo punto il driver prosegue con la successiva richiesta nella lista anche se nuove richieste sono state inserite nella lista prima della richiesta attiva.

Il kernel Linux versione 2.6 ha introdotto un nuovo algoritmo di scheduling dell'i/o. Anche se rimane disponibile un semplice algoritmo dell'ascensore, lo scheduler dell'i/o di default è ora il Completely Fair Queuing scheduler (cfq). Lo scheduler cfq è fondamentalmente diverso dagli algoritmi dell'ascensore, perché invece di ordinare le richieste in una lista mantiene un insieme di liste, per default una per ogni processo. Le richieste provenienti da un processo finiscono nella lista di quel processo. Per esempio, se

due processi stanno emettendo richieste di i/o, cfq manterrà due liste separate di richieste, una per ogni processo. Le liste sono mantenute secondo l'algoritmo c-scan.

cfq serve le liste in maniera differenziata. Mentre un algoritmo tradizionale c-scan non è sensibile allo specifico processo, cfq serve la lista di ogni processo secondo una politica round-robin. cfq estrae un numero configurabile di richieste (4 per default) da ciascuna lista prima di passare alla lista successiva. Questo metodo garantisce equità a livello dei processi: ogni processo riceve una uguale frazione della banda del disco. Il risultato è particolarmente vantaggioso in presenza di carichi di lavoro interattivi, dove la latenza di i/o è importante. Nella pratica cfq si comporta bene con la maggior parte dei carichi di lavoro.

20.8.2 Dispositivi a caratteri

Un driver di un dispositivo a caratteri può essere qualsiasi driver che non offra l'accesso diretto a blocchi di dati di dimensioni fisse. Ogni driver di un dispositivo a caratteri registrato dal kernel di Linux deve anche registrare un insieme di funzioni che realizzino le operazioni di i/o su file gestite dal driver. Il kernel non esegue quasi alcuna elaborazione preliminare su una richiesta di lettura o scrittura su file relativa a un dispositivo a caratteri, ma passa semplicemente la richiesta al dispositivo in questione lasciandogli il compito di servirla.

L'eccezione principale a questa regola è costituita dai driver dei dispositivi a caratteri relativi ai terminali: il kernel mantiene un'interfaccia uniforme a questi driver per mezzo di un insieme di strutture `tty_struct`: ognuna di esse fornisce il controllo del flusso e la bufferizzazione dei dati provenienti dal terminale e passa questi dati a un **interprete** (*line discipline*).

Una **disciplina** è un interprete delle informazioni provenienti dal terminale. La più comune è la `tty`, che incolla il flusso dei dati del terminale ai flussi di i/o standard dei processi utente in esecuzione, permettendo a questi processi di comunicare direttamente con il terminale dell'utente. Questo compito è complicato dal fatto che ci può essere più di un processo di questo tipo contemporaneamente in esecuzione, e la disciplina `tty` è responsabile del collegamento e del distacco dell'i/o del terminale dai vari processi a esso collegati a mano a mano che questi processi sono sospesi o riattivati dall'utente.

Il sistema Linux dispone di altre discipline che non hanno nulla a che fare con l'i/o di un processo utente: i protocolli di rete ppp e slip rappresentano un modo di codificare un collegamento di rete su un dispositivo terminale come una linea seriale. Questi protocolli sono realizzati nel sistema Linux come driver che da un lato appaiono al sistema della gestione dei terminali come interpreti con una determinata disciplina, e dall'altro appaiono al sistema per la gestione della rete come driver di dispositivi di rete: dopo che una di queste discipline è stata attivata su un dispositivo terminale, ogni dato che appare su quel terminale sarà direttamente instradato all'appropriato driver del dispositivo di rete.

20.9 Comunicazione fra processi

Linux fornisce un ricco ambiente per la comunicazione fra processi. La comunicazione può limitarsi alla notifica del verificarsi di un evento, da parte di un processo a un altro processo, ma può anche coinvolgere il trasferimento di dati fra processi.

20.9.1 Sincronizzazione e segnali

Il meccanismo standard di unix usato per comunicare a un processo che un evento si è verificato è il **segnale**. I segnali si possono inviare da un processo a ogni altro processo, con alcune limitazioni che riguardano i segnali inviati a processi di proprietà di un altro utente. Tuttavia è disponibile solo un limitato numero di segnali, che per di più non veicolano al processo destinatario altra informazione se non il mero fatto che un segnale è stato generato. Non è necessario che un segnale sia generato da un altro processo: anche il kernel genera internamente dei segnali; per esempio, può inviare un segnale a un processo server quando arrivano dati in un canale di rete, a un processo genitore quando un figlio termina, o a un processo in attesa quando è trascorso un intervallo di tempo impostato in un timer.

Internamente il kernel di Linux non usa segnali per comunicare con i processi eseguiti in modalità kernel: se uno di essi attende il verificarsi di un evento, non userà i segnali per ricevere la comunicazione dell'evento. Infatti, le comunicazioni nel kernel riguardanti eventi asincroni sono realizzate per mezzo degli stati di scheduling e delle strutture `wait_queue`. Questo meccanismo permette ai processi eseguiti in modalità kernel di informarsi vicendevolmente sugli eventi rilevanti, e dà la possibilità ai driver e al sistema di networking di generare eventi. Ogniquando un processo desidera attendere la terminazione di un evento, si sposta alla struttura `wait_queue` associata a quell'evento e comunica allo scheduler di non poter essere posto in esecuzione; l'evento in questione riattiva tutti i processi nella `wait_queue` al momento della sua terminazione. Si tratta di una procedura che permette a più processi di attendere il verificarsi di un unico evento: per esempio, se diversi processi stanno tentando di leggere un file da un disco, essi saranno tutti riattivati quando i dati sono stati trasferiti correttamente in memoria.

Anche se i segnali sono sempre stati il principale meccanismo di comunicazione asincrona fra processi, il sistema Linux dispone anche del meccanismo basato sui semafori dello unix System V: un processo si pone in attesa a un semaforo con la stessa facilità con la quale aspetterebbe un segnale, ma i semafori hanno due vantaggi: poter essere condivisi in gran numero da diversi processi indipendenti, e l'esecuzione atomica delle operazioni su più semafori. Internamente il meccanismo standard delle wait queue di Linux sincronizza i processi che comunicano tramite semafori.

20.9.2 Passaggio di dati fra processi

Il sistema Linux mette a disposizione diversi metodi per il passaggio di dati fra processi. Il meccanismo standard delle *pipe* unix permette a un processo figlio di ereditare dal genitore un canale di comunicazione; i dati scritti a un'estremità di tale canale di comunicazione possono essere letti all'altra estremità. Nel sistema Linux le pipe sono semplicemente un tipo particolare di *inode* del file system virtuale, dotato di una coppia di strutture `wait_queue` per sincronizzare il lettore e lo scrittore. Il sistema unix definisce anche un insieme di servizi di rete che possono inviare flussi di dati a processi locali o remoti. I servizi di rete sono trattati nel Paragrafo 20.10.

Un altro metodo di comunicazione è la memoria condivisa che offre un modo estremamente rapido di comunicare quantità arbitrarie di dati: qualsiasi informazione scritta da un processo in una regione di memoria condivisa può essere immediatamente letta da ogni altro processo che abbia quella regione associata al suo spazio d'indirizzi. Il principale svantaggio della memoria condivisa è che di per sé non offre alcun metodo di sincronizzazione: un processo non può chiedere al sistema operativo se qualche dato sia stato scritto in una regione di memoria condivisa, né può sospendere l'esecuzione fino a che una tale operazione di scrittura si sia verificata. La memoria condivisa diviene uno strumento particolarmente potente quando è usata insieme con un altro meccanismo di comunicazione fra processi che fornisca la sincronizzazione mancante.

In Linux una regione di memoria condivisa è un oggetto permanente che può essere creato o cancellato dai processi; un tale oggetto è trattato come se fosse un piccolo spazio d'indirizzi indipendente: gli algoritmi di paginazione possono scegliere di trasferire nel disco alcune pagine di memoria condivisa allo stesso modo in cui possono scegliere di trasferire una pagina di dati di un processo. L'oggetto che rappresenta la memoria condivisa agisce come memoria ausiliaria della regione di memoria condivisa proprio come un file può essere la memoria ausiliaria di una regione di memoria virtuale. Quando un file è associato a una regione di memoria virtuale, il verificarsi di un page fault causa la mappatura in memoria virtuale dell'appropriata pagina del file; similmente le associazioni relative alla memoria condivisa fanno sì che dalle eccezioni di pagina mancante risulti il caricamento in memoria di pagine prelevate da un oggetto permanente di memoria condivisa. Sempre in analogia con i file, gli oggetti che rappresentano la memoria condivisa conservano i loro contenuti anche se al momento non sono associati alla memoria virtuale di alcun processo.

20.10 Struttura di rete

Il networking è uno dei punti di forza di Linux: questo sistema operativo non solo gestisce i protocolli standard Internet per la comunicazione fra sistemi unix, ma realizza anche altri protocolli originariamente sviluppati per sistemi operativi diversi da unix. Essendo stato originariamente concepito per i pc e non per grandi stazioni di lavoro o server, gestisce molti protocolli usati nelle reti di pc, per esempio l'AppleTalk e l'ipx.

Internamente il kernel di Linux realizza i servizi di rete per mezzo di tre strati software:

1. l'interfaccia socket;
2. i driver dei protocolli;
3. i driver dei dispositivi di rete.

Le applicazioni degli utenti richiedono tutti i servizi di rete tramite l'interfaccia socket; essa è progettata per somigliare allo strato di socket del 4.3bsd, cosicché i programmi scritti per far uso delle socket di Berkeley potranno essere eseguiti dal sistema Linux senza

che sia necessario apportare modifiche al codice sorgente. Questa interfaccia è descritta nell'Appendice C, Paragrafo C.9.1, reperibile sulla piattaforma MyLab. L'interfaccia socket bsd è sufficientemente generale da poter rappresentare gli indirizzi di rete di un'ampia gamma di protocolli; il sistema Linux usa questa sola interfaccia per accedere a tutti i protocolli supportati, e non solo quelli dei sistemi bsd standard.

Lo strato software seguente è quello dei protocolli, organizzato in maniera simile all'analogo livello del bsd. Si richiede che tutti i dati che arrivino a questo livello siano etichettati da un identificatore che specifichi a quale protocollo appartengono; ciò vale sia per i dati provenienti dalla socket di un'applicazione, sia per quelli provenienti dal driver di un dispositivo di rete. I protocolli possono eventualmente comunicare fra di loro: all'interno dell'insieme dei protocolli Internet, per esempio, protocolli distinti gestiscono l'instradamento, la comunicazione degli errori e la ritrasmissione affidabile dei dati persi.

Lo strato dei protocolli può riscrivere i pacchetti, crearne di nuovi, dividerli in frammenti o riassemblarli da frammenti, o anche semplicemente scartare i dati in arrivo; alla fine, quando ha terminato l'elaborazione di un gruppo di pacchetti, li passa all'interfaccia socket se la destinazione dei dati è locale, oppure a un driver di un dispositivo di rete, se i pacchetti devono essere inviati lungo la rete. Lo strato di protocolli decide a quale socket o dispositivo inviare il pacchetto.

Tutta la comunicazione che avviene fra gli strati che realizzano i servizi di rete è eseguita passando singole strutture dette `skbuff` (socket buffer); esse contengono un insieme di puntatori a un'unica regione contigua di memoria che funge da buffer all'interno del quale i pacchetti di rete possono essere assemblati. I dati validi puntati da una struttura `skbuff` non devono necessariamente trovarsi all'inizio del buffer, e non devono neanche estendersi fino alla sua fine: il codice di networking può aggiungere o togliere dati agli estremi del pacchetto, a patto che il risultato non superi le dimensioni del buffer. Questa possibilità è particolarmente importante per le moderne cpu, i cui miglioramenti in velocità hanno abbondantemente superato le prestazioni della memoria centrale: l'architettura basata sulle strutture `skbuff` rende possibile gestire le intestazioni e dei codici per il controllo degli errori dei pacchetti evitando duplicazioni non necessarie dei dati.

L'insieme di protocolli più importante in Linux è la suite di protocolli ip, composta di un certo numero di protocolli distinti. Il protocollo ip realizza l'instradamento da un calcolatore all'altro ovunque nella rete; su questo protocollo si appoggiano i protocolli udp, tcp e icmp. Il protocollo udp trasferisce singoli datagrammi fra i calcolatori; il protocollo tcp instaura connessioni affidabili fra sistemi con consegna garantita nell'ordine originario, e ritrasmissione automatica dei dati persi; il protocollo icmp si usa per la trasmissione di messaggi di stato e di vari tipi di messaggi d'errore.

Si assume che i pacchetti (`skbuff`) che raggiungano il software dei protocolli siano già etichettati da un identificatore interno che indica a quale protocollo è attinente il pacchetto. Differenti driver dei dispositivi di rete codificano il tipo di protocollo in modo differente, quindi l'identificazione del protocollo va eseguita all'interno dei driver stessi. I driver usano a questo scopo una tabella hash degli identificatori di protocollo noti, e passano poi il pacchetto al protocollo appropriato; è possibile aggiungere nuovi protocolli alla tabella hash sfruttando il meccanismo di caricamento dei moduli del kernel.

I pacchetti ip che giungono al sistema sono consegnati al driver ip, il cui compito è quello di eseguire l'instradamento: esso determina la destinazione del pacchetto e lo inoltra all'appropriato driver di protocollo interno per la consegna locale, oppure lo reinserisce nella coda del driver di un dispositivo di rete se deve essere inoltrato a un altro calcolatore. La decisione riguardante l'instradamento viene presa sulla base di due tabelle: la base permanente di informazioni sull'instradamento (*forwarding information base*, fib), e una cache delle più recenti decisioni di instradamento. La fib contiene informazioni sulle configurazioni d'instradamento e può specificare percorsi basati su singoli indirizzi di destinazione o su raggruppamenti di destinazioni; è organizzata come un insieme di tabelle hash indirizzate dagli indirizzi di destinazione, e la ricerca parte sempre dalle tabelle che contengono i percorsi più specifici. Quando una ricerca di questo tipo ha successo, il percorso individuato è posto nella cache dei percorsi, la quale contiene solo instradamenti per destinazioni specifiche e non per gruppi, in modo che le ricerche siano più rapide. Un elemento della cache dei percorsi viene eliminato dopo un certo tempo d'inutilizzo.

In diverse fasi il protocollo ip passa i pacchetti a una sezione distinta di codice per la **gestione del firewall**, cioè il filtraggio selettivo dei pacchetti secondo criteri arbitrari ma di solito relativi alle strategie di sicurezza. Il gestore del firewall mantiene un certo numero di **catene di firewall** distinte, e permette il confronto di una struttura `skbuff` con una qualunque di esse; catene distinte assolvono funzioni distinte: una si usa per inoltrare i pacchetti, una per la ricezione dei pacchetti, e una per i dati generati localmente. Ogni catena è costituita da un insieme ordinato di regole, ciascuna delle quali specifica una fra varie funzione di filtro, oltre a pattern di dati che devono trovare riscontro nei pacchetti.

Due altre funzioni eseguite dal driver ip sono la scomposizione e il riassemblaggio dei pacchetti di grandi dimensioni: un pacchetto da spedire troppo grande per essere posto nella coda di un dispositivo viene semplicemente diviso in **frammenti** più piccoli che possono essere posti in coda; il calcolatore ricevente si occuperà di riassemblare i frammenti. Il driver ip mantiene un oggetto `ipfrag` per ogni frammento che attende il riassemblaggio, e un oggetto `ipq` per ogni datagramma in corso d'assemblaggio. I frammenti in arrivo sono confrontati con ogni `ipq`, e nel caso di riscontro positivo il frammento è aggiunto all'oggetto; altrimenti, si crea un nuovo `ipq`. Quando l'ultimo frammento di un `ipq` è arrivato si costruisce una struttura `skbuff` interamente nuova per alloggiare il pacchetto, e si passa di nuovo il pacchetto al driver ip.

I pacchetti che l'ip identifica come destinati al calcolatore locale sono passati a uno degli altri driver di protocollo. I protocolli tcp e udp adottano lo stesso metodo per associare ogni pacchetto alle relative socket mittenti e destinatarie: ogni coppia di socket collegate è identificata in modo unico dagli indirizzi del mittente e del destinatario e dai corrispondenti numeri di porta. Gli elenchi delle socket sono riuniti in una tabella hash la cui chiave è costituita da questi quattro valori di indirizzo e porta e che può essere usata per individuare le socket relative ai pacchetti in arrivo. Il protocollo tcp deve anche occuparsi delle connessioni non affidabili, e a tal fine mantenere liste ordinate dei pacchetti trasmessi, ma non riscontrati, che saranno ritrasmessi dopo un certo tempo, e liste dei pacchetti ricevuti in modo disordinato, che saranno presentati alla socket una volta ricevuti i dati mancanti.

20.11 Sicurezza

Il modello di sicurezza di Linux è strettamente correlato ai modelli di sicurezza tipici di unix. I problemi relativi alla sicurezza sono classificabili in due gruppi.

1. **Autenticazione.** Assicurare che nessuno possa accedere al sistema senza prima dimostrare di averne diritto.
2. **Controllo dell'accesso.** Fornire un meccanismo che permetta di controllare se un utente abbia diritto d'accesso a un certo oggetto, e che impedisca l'accesso se l'esito del controllo è negativo.

20.11.1 Autenticazione

L'usuale meccanismo di autenticazione in unix è sempre stato basato su un file di password leggibile da tutti: la password di un utente è combinata con un valore casuale, e il risultato è codificato tramite una funzione di codifica non invertibile e infine registrato nel file delle password. L'uso di una funzione di codifica non invertibile significa che la password originale non si può ricavare dal file delle password se non per tentativi. Quando un utente presenta una password al sistema essa viene ricombinata con il valore casuale memorizzato nel file delle password; al risultato si applica la funzione di codifica non invertibile, e se ciò che si ottiene coincide con il contenuto del file delle password l'utente è ammesso al sistema.

Storicamente le implementazioni di unix di questo meccanismo hanno incontrato diversi problemi: le password erano spesso limitate alla lunghezza di otto caratteri, e il numero dei possibili valori casuali era così basso che si potevano facilmente combinare le password più comuni con ogni possibile valore casuale e avere buone possibilità d'individuare una o più password contenute nel file, ottenendo così l'accesso non autorizzato alle utenze corrispondenti. Sono state introdotte estensioni del meccanismo delle password al fine di mantenere segrete le password codificate memorizzandole in un file non accessibile al pubblico; altre estensioni permettono l'uso di password più lunghe o adottano metodi di codifica più sicuri. Sono stati introdotti altri meccanismi di autenticazione che limitano il tempo di collegamento al sistema. Esistono anche meccanismi per trasmettere informazioni di autenticazione a sistemi collegati tramite la rete.

Per affrontare i problemi di autenticazione un certo numero di produttori di sistemi unix ha sviluppato un nuovo meccanismo di sicurezza: il sistema **pam** (*pluggable authentication modules*), basato su una libreria condivisa che può essere usata da ogni componente del sistema che abbia bisogno di autenticare utenti. Una implementazione di questo sistema è disponibile per il sistema Linux. Il sistema pam permette di caricare su richiesta moduli di autenticazione secondo le indicazioni contenute in un file di configurazione valido per tutto il sistema. Un nuovo meccanismo di autenticazione aggiunto in seguito si può registrare nel file di configurazione in modo che tutti i componenti del sistema possano immediatamente usufruirne. I moduli pam sono in grado di specificare metodi di autenticazione, limitazioni relative agli account, funzioni di avvio di una sessione di lavoro o funzioni di cambio di password. In quest'ultimo caso, quando un utente cambia la propria password, tutti i necessari meccanismi di autenticazione possono essere aggiornati in un'unica soluzione.

20.11.2 Controllo degli accessi

Nei sistemi unix, e così anche nel sistema Linux, il controllo degli accessi è realizzato per mezzo d'identificatori numerici unici. Un **identificatore utente** (*uid*) individua un singolo utente o un singolo insieme di diritti d'accesso, un identificatore di gruppo (*gid*) è un identificatore aggiuntivo che si può usare per determinare i diritti di più di un utente.

Il controllo degli accessi si applica a vari oggetti del sistema: ogni file disponibile nel sistema è protetto dal meccanismo standard del controllo degli accessi, e ciò vale anche per altri oggetti condivisi come le regioni di memoria condivise e i semafori.

Ogni oggetto in un sistema unix sottoposto al controllo degli accessi di singoli utenti o gruppi dispone di un unico uid e un unico gid associati; anche i processi utenti hanno un unico uid, ma possono avere più di un gid. Se l'uid di un processo corrisponde all'uid di un oggetto, quel processo gode dei **diritti utente** o dei **diritti di proprietà** di quell'oggetto; se gli uid non corrispondono ma uno dei gid di un processo corrisponde al gid di un oggetto, il processo gode dei **diritti di gruppo** su quell'oggetto; altrimenti, il processo ha i **diritti generici** (*world rights*) sull'oggetto.

Il sistema Linux esegue il controllo degli accessi assegnando agli oggetti una **maschera di protezione** che specifica quali modi d'accesso – lettura, scrittura o esecuzione – si devono concedere ai processi con diritti d'accesso proprietari, di gruppo o generici. Il proprietario di un oggetto, per esempio, potrebbe avere accesso a un file per la sua lettura, scrittura ed esecuzione; gli utenti di un certo gruppo potrebbero avere accesso per la lettura ma non per la scrittura; e chiunque altro potrebbe non avere alcun diritto d'accesso.

L'unica eccezione a quanto detto è costituita dallo uid privilegiato **root**: un processo dotato di questo speciale uid gode automaticamente dei diritti d'accesso a qualunque oggetto del sistema, scavalcando i normali controlli: processi di questo tipo hanno anche il permesso di eseguire operazioni privilegiate come la lettura di qualunque regione della memoria fisica o aprire socket di rete riservate. Questo meccanismo permette al kernel di impedire agli utenti ordinari di accedere a determinate risorse del sistema: la maggior parte delle risorse chiave interne del kernel sono implicitamente di proprietà dell'uid **root**.

Il sistema Linux adotta il meccanismo standard **setuid** dello unix, descritto nell'Appendice C, Paragrafo C.3.2; esso permette a un programma di godere, durante una certa esecuzione, di privilegi diversi da quelli dell'utente che esegue il programma: per esempio, il programma **lpr** che pone un file in coda di stampa ha accesso alle code di stampa del sistema, anche se l'utente che ne richiede l'esecuzione non lo ha. L'implementazione in unix di **setuid** distingue fra lo uid **reale** e lo uid **effettivo** di un processo: il primo è quello dell'utente che richiede l'esecuzione del programma; il secondo è quello del proprietario del file.

In Linux questo meccanismo è esteso in due direzioni. In primo luogo implementa il meccanismo **saved user-id** secondo le specifiche **posix**; si tratta di permettere a un processo di perdere e riacquisire ripetutamente il suo uid effettivo. Per ragioni di sicurezza, infatti, un programma potrebbe voler eseguire la maggior parte delle sue operazioni in un modo sicuro, rinunciando ai privilegi conferitigli dal suo **setuid** status, ma potrebbe voler usufruire di tutti i suoi privilegi durante le esecuzioni di alcune

operazioni. Le implementazioni standard di unix riescono a fornire questo servizio solo scambiando gli uid reali ed effettivi; una volta che ciò è avvenuto, lo uid effettivo precedente viene ricordato, ma lo uid reale del programma non sempre corrisponde allo uid dell'utente che ne richiede l'esecuzione. Il meccanismo `saved user-id` permette a un processo di rendere il suo uid effettivo uguale al suo uid reale e di ritornare poi al valore precedente del suo uid effettivo senza dover mai modificare il valore dello uid reale.

Il secondo miglioramento apportato dal sistema Linux è l'aggiunta di una caratteristica dei processi che permette di usufruire di un sottoinsieme dei diritti conferiti dallo uid effettivo: le proprietà **fsuid** e **fsgid** di un processo sono usate quando è consentito l'accesso a un file, e sono attive ognqualvolta lo uid effettivo o il gid lo sono; tuttavia, fsuid e fsgid possono essere attivate indipendentemente dagli identificatori effettivi, cosa che permette a un processo di accedere ai file per conto di un altro utente senza dover assumere per altri aspetti l'identità di quell'utente. In particolare, i processi server possono impiegare questo meccanismo per fornire file a un certo utente senza divenire suscettibili di terminazione o sospensione da parte di quell'utente.

Linux offre un metodo per il passaggio flessibile dei diritti da un programma a un altro, divenuto comune nelle moderne versioni dello unix. Una volta che un collegamento fra due processi del sistema sia stato istituito per mezzo di una socket locale, ognuno dei due processi può inviare all'altro un descrittore di file relativo a uno dei suoi file aperti; l'altro processo riceve quindi un descrittore duplicato dello stesso file: ciò permette a un client di offrire a un processo server l'accesso selettivo a un solo file senza conferirgli alcun altro privilegio. Per esempio, non è più necessario che un server di stampa sia in grado di leggere tutti i file di un utente che chieda una stampa; l'utente, e cioè il client, potrebbe semplicemente comunicare al server i descrittori dei file dei quali richiede la stampa, negando al server la possibilità di accedere agli altri file di sua proprietà.

20.12 Sommario

- Linux è un moderno sistema operativo gratuito basato sugli standard unix. È stato progettato per essere eseguito efficientemente e in modo affidabile sui comuni pc, ma può anche essere eseguito su diverse altre piattaforme, come i telefoni mobili. Fornisce un'interfaccia per il programmatore e un'interfaccia per l'utente compatibili con i sistemi unix standard, e può eseguire moltissime applicazioni unix, compreso un crescente numero di applicazioni commerciali.
- Il sistema Linux non si è sviluppato dal nulla: nel suo complesso comprende molti componenti originariamente sviluppati in maniera indipendente. La parte centrale del kernel del sistema operativo è interamente originale, ma permette l'esecuzione di molti programmi gratuiti esistenti per il sistema unix; ciò rende il tutto un completo sistema operativo compatibile con unix e non soggetto a limitazioni legali imposte dagli autori del codice.
- Per ragioni legate alle prestazioni il kernel di Linux è realizzato secondo schemi tradizionali come un blocco monolitico di codice, ma è sufficientemente modulare da permettere alla maggior parte dei driver di essere caricati o rimossi dinamicamente.
- Il sistema Linux è un sistema multutente che fornisce meccanismi di protezione dei processi ed è in grado di eseguire più processi in time sharing. Un nuovo processo può condividere selettivamente parti del suo ambiente d'esecuzione con il processo genitore, permettendo di realizzare la programmazione multithread. La comunicazione fra processi sfrutta sia i meccanismi del System V – code di messaggi, semafori e memoria condivisa – sia l'interfaccia a socket del bsd. È possibile usufruire simultaneamente di più protocolli di rete differenti per mezzo dell'interfaccia a socket.
- Il sistema per la gestione della memoria usa la condivisione delle pagine e la copiatura su scrittura per minimizzare la duplicazione dei dati condivisi da processi diversi. Le pagine sono caricate quando si genera un riferimento a esse, e sono trasferite di nuovo in memoria ausiliaria secondo un algoritmo lfu quando è necessario riappropriarsi di regioni della memoria fisica.
- Dal punto di vista dell'utente, il file system è un albero di directory conforme alla semantica unix: internamente Linux usa uno strato d'astrazione per gestire molti file system differenti. Gestisce file system virtuali, di rete e orientati ai dispositivi. I file system orientati ai dispositivi accedono ai dischi attraverso una cache delle pagine unificata con il sistema per la memoria virtuale.

Esercizi di ripasso

20.1 I moduli del kernel caricabili dinamicamente offrono flessibilità quando si aggiungono driver al sistema. Hanno anche qualche svantaggio? In quali circostanze conviene compilare il kernel in un singolo file binario e in quali conviene invece mantenerlo suddiviso in moduli? Spiegate la vostra risposta.

20.2 Il multithreading è una tecnica di programmazione comune. Descrivete tre modi diversi per implementare i thread e confrontate questi tre metodi con il meccanismo `clone()` di Linux. Quando un meccanismo alternativo può essere preferibile all'impiego di cloni? Quando può valere il viceversa?

20.3 Il kernel di Linux non permette la paginazione della memoria del kernel. Come influisce questa restrizione sul progetto del kernel? Elencate due vantaggi e due svantaggi di questa scelta di progetto.

20.4 Discutete tre vantaggi del collegamento dinamico (condiviso) di librerie rispetto al collegamento statico. Descrivete due casi in cui sia preferibile il collegamento statico.

20.5 Confrontate l'utilizzo di socket di rete con l'utilizzo di memoria condivisa come meccanismi di comunicazione di dati tra processi in uno stesso computer. Quali sono i vantaggi di ciascun metodo? Quando uno è preferibile rispetto all'altro?

20.6 Una volta i sistemi unix utilizzavano ottimizzazioni dell'organizzazione del disco basate sulla posizione dei dati rispetto alla rotazione del disco. Le moderne implementazioni, incluso Linux, effettuano invece un'ottimizzazione mirata semplicemente all'accesso sequenziale. Perché è stata fatta questa scelta? Da quali caratteristiche hardware trae vantaggio l'accesso sequenziale? Perché l'ottimizzazione basata sulla rotazione non è più così utile?

Esercizi

20.7 Quali vantaggi e svantaggi presenta la scrittura di un sistema operativo con un linguaggio ad alto livello, quale il C?

20.8 In quali circostanze la successione di chiamate di sistema `fork()` ed `exec()` è da ritenersi la più appropriata? Quando è preferibile `vfork()`?

20.9 Quale tipo di socket dovrebbe essere usato per realizzare un programma per il trasferimento di file tra calcolatori? Quale tipo si dovrebbe scegliere per un programma che effettui un monitoraggio periodico al fine di verificare la presenza di altri elaboratori sulla rete? Motivate le vostre risposte.

20.10 Il sistema operativo Linux è eseguibile su diverse piattaforme. Dite che cosa devono fare gli sviluppatori di Linux per assicurare l'adattabilità del sistema a diverse cpu e diverse architetture per la gestione della memoria, e per minimizzare la quantità richiesta di codice del kernel specifico per le singole architetture.

20.11 Supponete di restringere l'accesso, da parte dei moduli caricabili del kernel, soltanto ad alcuni dei simboli definiti all'interno del kernel stesso. Quali i vantaggi e gli svantaggi di questa scelta?

20.12 Quali obiettivi fondamentali ha il meccanismo di risoluzione dei conflitti, utilizzato dal kernel di Linux per il caricamento dei moduli del kernel?

20.13 Discutete come l'operazione `clone()` di Linux sia usata per ottenere comportamenti equivalenti sia a quelli dei processi sia a quelli dei thread.

20.14 Attribuireste ai thread di Linux la qualifica di thread a livello utente, o piuttosto quella di thread a livello kernel? Fornite le opportune argomentazioni a sostegno della vostra risposta.

20.15 Dite quali sono i costi aggiuntivi implicati dalla creazione e dallo scheduling di un processo rispetto al costo di un thread ottenuto col meccanismo `clone`.

20.16 Come può il Completely Fair Scheduler (cfs) di Linux fornire una maggiore equità rispetto a uno scheduler tradizionale unix? Quando è garantita questa equità?

20.17 Dite quali sono le due variabili configurabili del Completely Fair Scheduler (cfs). Descrivete i pro e i contro di impostare ciascuna di esse a valori molto piccoli e molto grandi.

20.18 Lo scheduler di Linux implementa lo scheduling in tempo reale debole (*soft real-time scheduling*); dite quali funzioni, necessarie per certi compiti di programmazione in tempo reale, mancano? Come si potrebbero includere nel kernel e a che costo?

20.19 In quali circostanze un processo utente richiederebbe un'operazione il cui esito è l'allocazione di una regione di memoria a valori nulli?

20.20 Quali circostanze fanno sì che una pagina di memoria sia mappata nello spazio degli indirizzi di un programma utente con l'opzione di copiatura su scrittura attivata?

20.21 Nel sistema Linux le librerie condivise eseguono molte operazioni fondamentali per il sistema operativo. Dite qual è il vantaggio di scorporare dal kernel il codice relativo a questi servizi, ed elencate anche gli eventuali svantaggi. Motivate le risposte.

20.22 Quali sono i vantaggi di un file system con journaling come ext3 di Linux? Quali sono i costi? Perché ext3 fornisce la possibilità di effettuare il journaling solo per i metadati?

20.23 La struttura della directory nel sistema Linux potrebbe ricomprendere file appartenenti, in senso stretto, a file system diversi: uno di loro è il file system `/proc`. Quali conseguenze comporta la necessità di ospitare file system di tipo diverso nell'architettura del kernel di Linux?

20.24 Per quali aspetti la funzionalità `setuid` di Linux differisce dalla medesima funzionalità dei sistemi svr4?

20.25 Il codice sorgente di Linux è gratuitamente disponibile tramite la rete Internet o presso i rivenditori di cd-rom. Individuate tre possibili conseguenze riguardanti la sicurezza del sistema Linux.

CAPITOLO 21

Windows 10

Aggiornamento a cura di Alex Ionescu

Microsoft Windows 10 è un sistema operativo client multitasking con prelazione sviluppato per i microprocessori che implementano set di istruzioni delle architetture Intel ia-32, amd64, arm e arm64. Il corrispondente sistema operativo server di Microsoft, Windows Server 2016, si basa sullo stesso codice di Windows 10, ma supporta solo le architetture a 64 bit amd64.

Windows 10 è l'ultimo di una serie di sistemi operativi Microsoft basati sul codice nt, che ha sostituito i precedenti sistemi basati su Windows 95/98. In questo capitolo sono trattati gli obiettivi chiave del sistema, la sua architettura a strati che lo rende così facile da usare, il file system, le caratteristiche di networking e l'interfaccia di programmazione.

21.1 Storia

Nella metà degli anni '80 Microsoft e ibm cooperarono per sviluppare il sistema operativo os/2, scritto in linguaggio assembly per sistemi a singola cpu Intel 80286. Nel 1988 Microsoft decise di abbandonare la collaborazione con ibm e sviluppare un proprio sistema operativo facilmente adattabile dalla "nuova tecnologia" (nt), che includesse le interfacce per la programmazione delle applicazioni os/2 e posix. Nell'ottobre del 1988 Dave Cutler, progettista del sistema operativo dec vax/vms, fu assunto e incaricato della progettazione e realizzazione di questo nuovo sistema operativo.

Originariamente per il sistema Windows nt si sarebbe dovuta adottare l'api del sistema operativo os/2 come suo ambiente naturale, ma durante lo sviluppo si scelse di adottare una nuova api a 32 bit (Win32), basata sulla diffusa api a 16 bit di Windows 3.0. Le prime versioni del sistema Windows nt furono Windows nt 3.1 e Windows nt 3.1 Advanced Server (in quel periodo, la più recente versione dell'ambiente Windows a 16 bit era la 3.1). Nella versione 4.0 il sistema Windows nt adottò l'interfaccia utente di Windows 95 e incorporò il software del server Web e il browser; inoltre le procedure dell'interfaccia utente e il codice per la grafica furono spostati all'interno del kernel al fine di migliorare le prestazioni, con l'effetto collaterale di ridurre l'affidabilità del sistema.

Nonostante le prime versioni del sistema nt fossero state adattate per altre architetture, Windows 2000 (uscito nel febbraio 2000) ha limitato la portabilità ai soli processori Intel e compatibili, per ragioni di mercato. Rispetto a Windows nt, Windows 2000 vantava considerevoli innovazioni, quali: aggiunta della Active Directory (un servizio di directory basato sullo standard X.500), migliore gestione dei servizi di rete, supporto ai dispositivi *plug-and-play*, un file system distribuito, la possibilità di aumentare le unità di elaborazione ed espandere la memoria.

21.1.1 Windows XP, Vista e 7

Nell'ottobre 2001 l'uscita di Windows xp ha costituito l'evoluzione del sistema operativo desktop Windows 2000 e, nel contempo, come sostituzione di Windows 95/98. Nel 2002 venivano licenziate le versioni server di Windows xp, con il nome di Windows .net Server. Windows xp ha dotato l'interfaccia utente (gui) di un aspetto grafico rinnovato, a cui si aggiungono *nuove funzionalità di uso intuitivo*. Numerose funzionalità sono state introdotte per risolvere in modo automatico i problemi delle applicazioni, e del sistema operativo stesso. Windows xp può contare su una migliore gestione delle connessioni di rete e dei dispositivi (fra cui configurazione automatica delle connessioni wireless, un servizio di messaggistica istantanea, supporto multimediale e applicazioni digitali fotografiche e video), su un notevolissimo aumento delle prestazioni, sia per i pc desktop sia per sistemi multiprocessore più potenti, e su accresciute garanzie di affidabilità e sicurezza.

L'aggiornamento tanto atteso di Windows xp, chiamato Windows Vista, è stato rilasciato nel gennaio 2007, ma non è stato ben accolto. Anche se Windows Vista includeva molti miglioramenti che più tardi sono stati riproposti in Windows 7, questi sono stati offuscati dai problemi di lentezza e di compatibilità percepiti nell'utilizzo di Windows Vista. Microsoft ha risposto alle critiche al sistema Vista migliorando i propri processi di ingegnerizzazione e lavorando a stretto contatto con i produttori di hardware e applicazioni per Windows.

Il risultato è stato Windows 7, rilasciato nell'ottobre del 2009 insieme alle corrispondenti versioni server di Windows. Tra le modifiche ingegneristiche più significative vi è l'aumento dell'uso della tracciatura di esecuzione (*execution tracing*) per l'analisi del comportamento del sistema, al posto dell'utilizzo di contatori o profiling. Il tracing permane in esecuzione nel sistema, monitorando centinaia di scenari di esecuzione. Quando uno di questi scenari fallisce, o quando ha successo ma non ottiene buone prestazioni, è possibile analizzare i dati raccolti per determinare la causa del problema.

21.1.2 Windows 8

Tre anni dopo, nell'ottobre 2012, mentre l'industria si rivolgeva verso il mobile computing e il mondo delle **app**, Microsoft ha rilasciato Windows 8, il cambiamento più significativo del sistema operativo da Windows xp. Windows 8 ha introdotto una nuova interfaccia utente (denominata **Metro**) e una nuova api di programmazione (denominata **Winrt**), oltre a un nuovo modo di gestire le applicazioni, basato su un meccanismo di sandbox, realizzato attraverso un **sistema di pacchetti** che supporta esclusivamente il neonato **Windows Store**, concorrente di Apple App Store e di Android Store. Windows 8 ha inoltre incluso moltissimi miglioramenti in termini di sicurezza, gestione dell'avvio e prestazioni. Allo stesso tempo, il supporto per i "sottosistemi", un concetto che descriveremo più avanti nel capitolo, è stato rimosso.

Per muoversi verso il nuovo mondo mobile, Windows 8 è stato portato per la prima volta su architetture arm a 32 bit e ha introdotto diverse modifiche alle funzionalità di gestione dell'alimentazione e di estensibilità dell'hardware del kernel (trattate più avanti in questo capitolo). Microsoft ha commercializzato due versioni di questo porting: una versione, chiamata Windows rt, può eseguire sia applicazioni di Windows Store che alcune applicazioni "classiche" del marchio Microsoft, come Blocco Note, Internet Explorer e, soprattutto, Office; l'altra versione, chiamata Windows Phone, può eseguire esclusivamente le applicazioni dello Store.

Per la prima volta in assoluto, Microsoft ha rilasciato dei dispositivi mobili marchiati con il suo brand e denominati "Surface", tra cui il Surface rt, un tablet che esegue esclusivamente il sistema operativo Windows rt. Poco più tardi, Microsoft ha acquistato Nokia e ha iniziato la commercializzazione di smartphone con il proprio marchio che eseguivano il sistema Windows Phone.

Sfortunatamente, Windows 8 ha fallito i suoi obiettivi di mercato, per diversi motivi. Da un lato, Metro si concentra su un'interfaccia orientata ai tablet che obbliga gli utenti abituati ai vecchi sistemi operativi Windows a cambiare completamente il modo di lavorare sui loro computer desktop. Windows 8, per esempio, sostituisce il menu start con funzionalità per schermi touch e le icone del desktop con "tile" animate, e offre uno scarso supporto all'input da tastiera. Sul versante mobile, la scarsità di applicazioni nel Windows Store, l'unica fonte da cui ottenere app per telefoni e tablet di Microsoft, ha portato al fallimento di questi dispositivi, provocando l'uscita di produzione del dispositivo Surface rt e il write off dell'investimento nell'acquisizione di Nokia.

Microsoft ha cercato rapidamente di risolvere molti di questi problemi con il rilascio di Windows 8.1, nell'ottobre 2013. Questa versione elimina molti dei difetti di usabilità di Windows 8 su dispositivi non mobili, aumentando l'usabilità in presenza di tastiera e

mouse tradizionali e fornendo alternative all'utilizzo dell'interfaccia Metro basata su tile. Sono inoltre proseguite le modifiche finalizzate a migliorare sicurezza, prestazioni e affidabilità introdotte in Windows 8. Sebbene questa versione sia stata meglio accolta, la continua mancanza di applicazioni nel Windows Store ha costituito un notevole ostacolo alla penetrazione del sistema operativo nel mercato mobile, mentre sul versante desktop e server i programmati di applicazioni si sono sentiti abbandonati a causa della mancanza di miglioramenti nella loro area.

21.1.3 Windows 10

Con il rilascio di **Windows 10** nel luglio 2015 e della relativa versione server, Windows Server 2016, nell'ottobre 2016, Microsoft è passata a un modello "Windows-as-a-Service" (WaaS), con miglioramenti periodici delle funzionalità. Windows 10 riceve miglioramenti incrementali mensili denominati *feature rollup*, oltre a nuove versioni rilasciate ogni otto mesi e denominate "aggiornamenti" (update). Inoltre, ogni nuova release è resa disponibile al pubblico tramite il programma Windows Insider, o wip, che rilascia versioni su base settimanale. Analogamente a servizi cloud e siti web come Facebook e Google, il nuovo sistema operativo utilizza la telemetria in tempo reale (invia informazioni di debug a Microsoft) e il tracing per abilitare e disabilitare dinamicamente alcune funzionalità per effettuare il test a/b (ovvero confrontare il comportamento di un versione "A" rispetto a una versione "B" simile), provare nuove funzionalità verificando i problemi di compatibilità e aggiungere o rimuovere il supporto per hardware moderno o legacy. Queste funzionalità dinamiche di configurazione e di test sono ciò che rende tale release un'implementazione "as-a-service".

Windows 10 ha reintrodotto il menu di avvio, ha ripristinato il supporto per la tastiera e ha ridotto il ruolo delle applicazioni a schermo intero e delle tile live. Dal punto di vista dell'utente, queste modifiche hanno riportato la facilità d'uso che gli utenti si aspettavano dai sistemi operativi desktop basati su Windows. Inoltre, Metro (che è stato ribattezzato **Modern**) è stato ridisegnato in modo che le applicazioni del Windows Store possano essere eseguite sul desktop normale insieme alle applicazioni legacy. Infine, un nuovo meccanismo chiamato **Windows Desktop Bridge** ha reso possibile l'inserimento di applicazioni Win32 nel Windows Store, compensando la mancanza di applicazioni scritte appositamente per i nuovi sistemi. Nel frattempo, Microsoft ha aggiunto il supporto per C++11, C++14 e C++17 in Visual Studio e molte nuove api sono state aggiunte alla tradizionale api di programmazione Win32. Un'ulteriore novità di Windows 10 è il rilascio dell'architettura uwp (Unified Windows Platform), che consente alle applicazioni di essere scritte in modo da poter essere eseguite su Windows per desktop, Windows per IoT, xbox One, Windows Phone e Windows 10 Mixed Reality (precedentemente noto come Windows Holographic).

Windows 10 ha anche sostituito il concetto di sottosistemi multipli, rimosso in Windows 8 (come accennato in precedenza), con un nuovo meccanismo chiamato **Pico Provider**, che consente l'esecuzione nativa di file binari non modificati appartenenti a un diverso sistema operativo su Windows 10. Nell'Anniversary Update rilasciato nell'agosto 2016 questa funzionalità è stata utilizzata per fornire il sottosistema Windows per Linux, utilizzabile per eseguire binari elf di Linux in un ambiente user Ubuntu interamente non modificato.

In risposta alle crescenti pressioni della concorrenza nei settori della telefonia mobile e del cloud computing, Microsoft ha anche apportato miglioramenti in termini di potenza, prestazioni e scalabilità, consentendo l'esecuzione di Windows 10 su un numero maggiore di dispositivi. Per esempio, una versione denominata Windows 10 IoT Edition è specificamente progettata per ambienti come Raspberry Pi, mentre il supporto per tecnologie di cloud computing come i container è integrato in Docker per Windows. Windows 10 integra anche la tecnologia di virtualizzazione Microsoft Hyper-V, che fornisce ulteriore sicurezza e supporto nativo per l'esecuzione di macchine virtuali. È stata inoltre rilasciata la versione speciale di Windows Server denominata Windows Server Nano, con overhead estremamente basso e adatta per applicazioni containerizzate e altri usi di cloud computing.

Windows 10 è un sistema operativo multutente che supporta l'accesso simultaneo tramite servizi distribuiti o attraverso più istanze dell'interfaccia grafica tramite Windows Terminal Services. Le edizioni server di Windows 10 supportano sessioni di terminal server simultanee da sistemi desktop Windows. Le edizioni desktop del terminal server multiplano tastiera, mouse e monitor tra le sessioni del terminale virtuale per ciascun utente connesso. Questa funzionalità, chiamata **commutazione rapida dell'utente** (*fast user switching*), consente agli utenti la prelazione sulla console di un pc senza necessità di disconnettersi e riconnettersi.

Torniamo brevemente agli sviluppi sulla gui di Windows. Abbiamo osservato in precedenza che l'implementazione della gui è stata spostata in modalità kernel in Windows nt 4.0 per migliorare le prestazioni. Ulteriori miglioramenti delle prestazioni sono stati ottenuti in Windows Vista con la creazione di un nuovo componente in modalità utente, chiamato **Desktop Window Manager** (dwm), che fornisce l'aspetto dell'interfaccia Windows e si colloca al di sopra del software grafico DirectX. DirectX è ancora eseguito in modalità kernel, così come il codice (Win32k) che implementa la precedente interfaccia e i precedenti modelli grafici di Windows (User e gdi). Windows 7 ha apportato modifiche sostanziali al dwm, riducendo in modo significativo la sua occupazione di memoria e migliorandone le prestazioni, mentre Windows 10 ha apportato ulteriori miglioramenti, soprattutto in materia di prestazioni e sicurezza.

Inoltre, Windows DirectX 11 e 12 includono meccanismi gppu (calcolo general-purpose su hardware gpu) tramite Direct-Compute e molte componenti di Windows sono state aggiornate per sfruttare questo modello grafico ad alte prestazioni. Attraverso un nuovo livello di rendering chiamato Coreui, anche le applicazioni legacy possono ora trarre vantaggio dal rendering basato su DirectX (per la creazione del contenuto finale dello schermo).

Windows xp è stata la prima versione di Windows a 64 bit (dal 2001 per ia64 e dal 2005 per amd64). Il file system nativo di nt (ntfs) e molte delle api Win32 hanno sempre usato numeri interi a 64 bit, quando necessario; pertanto, l'estensione a 64 bit di Windows xp serviva principalmente per il supporto di indirizzi virtuali estesi. Le edizioni a 64 bit di Windows, inoltre, supportano anche memorie fisiche molto più grandi, con l'ultima versione di Windows Server 2016 che supporta fino a 24 tb di ram. Quando Windows 7 è stato rilasciato l'isa amd64 era diventato disponibile su quasi tutte le cpu, sia di Intel sia di amd. Inoltre, allo stesso tempo, le memorie fisiche sui sistemi client iniziarono a superare spesso il limite di 4 gb imposto dall'architettura ia-32. Per queste ragioni la versione a 64 bit di Windows 10 è ormai comunemente installata sui sistemi client, a parte i dispositivi IoT e mobili. Poiché l'architettura amd64 è fedelmente compatibile con ia-32 a livello dei singoli processi, in un unico sistema è possibile utilizzare liberamente applicazioni a 32 e 64 bit. È interessante notare che uno schema simile sta ora emergendo sui sistemi mobili. Apple ios è il primo sistema operativo mobile a supportare l'architettura arm64, che è l'estensione a 64 bit di arm (detta anche AArch64). Una futura versione di Windows 10 sarà inoltre ufficialmente fornita con un porting su arm64 progettato per una nuova classe di hardware, con compatibilità per le applicazioni ia-32 ottenuta tramite l'emulazione e la ricompilazione dinamica jit.

Nel resto della nostra descrizione di Windows 10 non faremo distinzione tra le versioni client e le corrispondenti versioni server. Le diverse versioni si basano sugli stessi componenti di base ed eseguono gli stessi file binari per il kernel e la maggior parte dei driver. Analogamente, anche se Microsoft distribuisce diverse edizioni di ogni release per essere presente sul mercato con diverse fasce di prezzo, sono poche le differenze che si riflettono sulla parte centrale del sistema. In questo capitolo ci concentreremo principalmente sulle componenti fondamentali di Windows 10.

21.2 Princìpi di progettazione

Gli obiettivi di progettazione dichiarati da Microsoft per il sistema operativo Windows includono sicurezza, affidabilità, compatibilità con le applicazioni Windows e posix, alte prestazioni e adattamento a livello internazionale. Alcuni ulteriori obiettivi quali l'efficienza energetica e il supporto dinamico dei dispositivi sono stati recentemente aggiunti a questa lista. Nel seguito discutiamo ciascuno di questi obiettivi e vediamo come essi vengano raggiunti in Windows 10.

21.2.1 Sicurezza

Gli obiettivi di sicurezza di Windows Vista e versioni successive richiedevano qualcosa in più della semplice uniformità agli standard progettuali che hanno permesso a Windows nt 4.0 di ricevere la classificazione di sicurezza C2 dal governo degli Stati Uniti d'America (la classificazione C2 indica un moderato livello di protezione nei confronti di programmi difettosi e di attacchi malevoli; Queste classificazioni sono state definite dal Dipartimento della Difesa nel "Trusted Computer System Evaluation Criteria", noto anche come Orange Book.). Un'approfondita revisione del codice e attività di testing sono state combinate con strumenti di analisi automatizzati per identificare e studiare potenziali difetti, dietro i quali potrebbero annidarsi falle di sicurezza. Inoltre, i programmi partecipativi **bug bounty** consentono a ricercatori e professionisti della sicurezza esterni di identificare e segnalare problemi di sicurezza precedentemente sconosciuti, in cambio di pagamenti e crediti nei rollup di sicurezza mensili, rilasciati da Microsoft per mantenere Windows 10 il più sicuro possibile.

Windows basa tradizionalmente la sicurezza sul controllo discrezionale degli accessi. Gli oggetti di sistema, inclusi i file, le impostazioni del registro di sistema e gli oggetti del kernel, sono protetti da **liste di controllo degli accessi (acl)** (si veda il Paragrafo 13.4.2). Le acl sono però vulnerabili a errori di utenti e programmatore e ai più comuni attacchi sui sistemi consumer, in cui l'utente viene indotto in maniera ingannevole a eseguire codice, spesso durante la navigazione sul Web. Windows Vista ha introdotto un meccanismo chiamato **livelli di integrità** che funge da rudimentale sistema di capability per il controllo degli accessi. Agli oggetti e ai processi viene assegnata un'integrità bassa, media o alta e il livello di integrità determina quali diritti avranno gli oggetti e i processi. Per esempio, Windows (in base alle sue politiche) non consente a un processo di modificare un oggetto con un livello di integrità più alto né di leggere la memoria di un processo di integrità superiore, indipendentemente dalle informazioni contenute nella acl.

Windows 10 ha ulteriormente rafforzato il modello di sicurezza introducendo una combinazione di controllo degli accessi basato su attributi (abac) e controllo degli accessi basato su attestazioni (cabc). Entrambe le funzionalità sono utilizzate per implementare il controllo dinamico degli accessi (dac) sulle edizioni server e per supportare il sistema basato sulle abilitazioni utilizzato dalle applicazioni Windows Store e dalle applicazioni Modern e pacchettizzate. Con attributi e attestazioni, gli amministratori di sistema non devono fare affidamento sul nome di un utente (o sul gruppo a cui appartiene) come unico strumento che il sistema di sicurezza può utilizzare per filtrare l'accesso a oggetti come i file, ma può anche considerare le proprietà dell'utente, per esempio, l'anzianità nell'organizzazione, lo stipendio e così via. Queste proprietà sono codificate come attributi, che sono abbinate a voci di controllo di accesso condizionale nell'acl, per esempio "Anzianità > = 10 anni".

Windows utilizza la crittografia all'interno dei comuni protocolli, per esempio dei protocolli utilizzati per comunicare in modo sicuro con i siti web. La crittografia è usata anche per proteggere da occhi indiscreti i file degli utenti memorizzati su disco. Windows 7 e le versioni successive consentono agli utenti di crittografare facilmente anche un intero disco, oltre ai dispositivi di memorizzazione rimovibili come le unità usb di memoria flash, per mezzo di una funzione chiamata BitLocker. Se viene rubato un computer con un disco criptato, i ladri avranno bisogno di utilizzare tecnologie molto sofisticate (come un microscopio elettronico) per ottenere l'accesso a qualsiasi file, e sarà impossibile ottenere un tale accesso se l'utente ha anche configurato un token esterno basato su usb (meno che anche il token usb non sia stato rubato).

Questi tipi di funzionalità di sicurezza si concentrano sulla sicurezza degli utenti e dei dati, ma sono vulnerabili a programmi altamente privilegiati che analizzano contenuti arbitrari e che possono essere attivati grazie a errori di programmazione che innescano l'esecuzione di codice dannoso. Per questa ragione, Windows adotta anche misure di sicurezza spesso chiamate "mitigazione degli exploit" (*exploit mitigation*), che comprendono strumenti di ampia portata come l'aslr (address-space layout randomization), il dep (Data Execution Prevention), il cfg (Control-Flow Guard) e l'acg (Arbitrary Code Guard), oltre a strumenti mirati specifici per varie tecniche di exploit (che non rientrano negli scopi di questo capitolo).

Dal 2001, i chip di Intel e amd consentono di contrassegnare le pagine di memoria in modo che non possano contenere codice eseguibile. La funzionalità dep di Windows contrassegna gli stack e gli heap di memoria (oltre a tutte le altre aree di soli dati) in modo che non possano essere utilizzati per eseguire codice. Questo impedisce attacchi in cui un bug del programma consente un buffer overflow e di conseguenza l'esecuzione del contenuto del buffer. Inoltre, a partire da Windows 8.1, tutte le allocazioni di memoria di soli dati del kernel vengono contrassegnate in questo modo.

Poiché dep impedisce l'esecuzione di dati controllati dagli hacker come codice, gli sviluppatori malintenzionati si sono mossi verso **attacchi di riutilizzo del codice** ("code reuse"), in cui il codice eseguibile esistente all'interno del programma viene riutilizzato in modi imprevisti (solo alcune parti del codice vengono eseguite e il flusso viene reindirizzato da un flusso di istruzioni a un altro). aslr ostacola molte forme di tali attacchi rendendo casuale la posizione delle regioni di memoria eseguibili (e di quelle contenenti dati), e quindi più difficile per gli attacchi di riutilizzo del codice sapere dove si trova il codice esistente. Questa protezione fa sì che spesso un sistema sotto attacco da parte di un utente remoto incorra in errori o si arresti in modo anomalo.

Nessuna mitigazione è perfetta, tuttavia, e aslr non fa eccezione. Per esempio, potrebbe essere inefficace contro gli attacchi locali (in cui alcune applicazioni sono indotte a caricare il contenuto dalla memoria secondaria, per esempio), così come contro i cosiddetti **attacchi di fuga di informazioni** ("information leak") in cui un programma viene spinto a rivelare parte del suo spazio di indirizzi. Per risolvere tali problemi, Windows 8.1 ha introdotto una tecnologia chiamata cfg, che è stata migliorata notevolmente in Windows 10. cfg lavora insieme al compilatore, al linker, al loader e al gestore della memoria per convalidare l'indirizzo di destinazione di qualsiasi salto indiretto (come una chiamata a funzione o un'istruzione di salto) mediante un elenco di prologhi di funzione validi. Se un programma viene ingannato e reindirizza il flusso di controllo da altre posizioni verso una tale istruzione, si blocca.

Se gli utenti malintenzionati non possono utilizzare dati eseguibili in un attacco, né riutilizzare il codice esistente, potrebbero tentare di indurre un programma ad allocare, da solo, codice eseguibile e scrivibile, che può quindi essere riempito dall'utente malintenzionato. In alternativa, gli autori di attacchi potrebbero modificare i dati scrivibili esistenti e contrassegnarli come dati eseguibili. La mitigazione acg di Windows 10 vieta queste operazioni. Una volta che il codice eseguibile è stato caricato, non potrà mai più essere modificato e, una volta caricati i dati, questi non potranno mai essere contrassegnati come eseguibili.

Windows 10 ha oltre trenta mitigazioni di sicurezza oltre a quelle qui descritte. Questo insieme di caratteristiche ha reso più difficili gli attacchi tradizionali, e forse ciò spiega in parte perché le applicazioni crimeware, come gli adware, le frodi di carte di credito e i ransomware siano diventate così prevalenti. Questi ultimi tipi di attacchi si basano sul fatto che gli utenti possono causare volontariamente e manualmente danni ai propri computer (per esempio facendo doppio clic sulle applicazioni nonostante gli avvisi o immettendo il proprio numero di carta di credito in una pagina bancaria fasulla), ma nessun sistema operativo può essere progettato per lottare contro la creduloneria e la curiosità degli esseri umani. Recentemente, Microsoft ha iniziato a lavorare direttamente con i produttori di chip, come Intel, per creare mitigazioni di sicurezza direttamente nell'isa. Uno di questi è, per esempio, il cet (**Control-flow Enforcement Technology**), un'implementazione hardware di cfg che protegge anche dagli attacchi rop (Return-Oriented-Programming) utilizzando uno stack shadow hardware, che contiene l'insieme di indirizzi di ritorno memorizzati quando viene chiamata una routine. Prima di eseguire il ritorno dalla chiamata, gli indirizzi vengono controllati e viene verificata la corrispondenza con gli indirizzi nello stack: una mancata corrispondenza significa che lo stack è stato compromesso e che è necessario intervenire.

Un altro aspetto importante della sicurezza è l'integrità. In questo ambito, Windows offre diverse funzionalità di **firma digitale** e utilizza le firme digitali per contrassegnare i file binari del sistema operativo in modo da poter verificare che i file siano stati prodotti da Microsoft o da un'altra società nota. Nelle versioni di Windows non ia-32, viene attivato all'avvio un modulo di integrità del codice per garantire che tutti i moduli caricati nel kernel abbiano firme valide, assicurando che questi non siano stati alterati. Inoltre, le versioni arm di Windows 8 estendono il modulo di integrità del codice con i controlli di integrità del codice in modalità utente, che verificano che tutti i programmi utente siano stati firmati da Microsoft o consegnati tramite il Windows Store. Una versione speciale di Windows 10 (Windows 10 S, destinata principalmente al mercato dell'istruzione) offre controlli di firma simili su tutti i sistemi ia-32 e amd64. Le firme digitali vengono anche utilizzate come parte di Code Integrity Guard, che consente alle applicazioni di difendersi dal caricamento di codice eseguibile che non sia stato firmato in modo appropriato dalla memoria secondaria. Per esempio, un utente malintenzionato potrebbe sostituire il file binario di terze parti con il proprio, ma la firma digitale non funzionerà e Code Integrity Guard non caricherà il file binario nello spazio dei processi.

Infine, le versioni aziendali di Windows 10 consentono di attivare una nuova funzionalità di sicurezza denominata **Device Guard**. Questo meccanismo consente alle organizzazioni di personalizzare i requisiti di firma digitale dei loro sistemi informatici, nonché inserire in blacklist e whitelist i signoli certificati o persino gli hash binari. Per esempio, un'organizzazione può scegliere di consentire solamente l'esecuzione di programmi in modalità utente firmati da Microsoft, Google o Adobe sui propri computer aziendali.

21.2.2 Affidabilità

Il sistema operativo Windows è cresciuto molto nei suoi primi dieci anni e questa maturazione ha portato al sistema Windows 2000. Allo stesso tempo è aumentata la sua affidabilità, grazie a fattori quali la maturità del codice sorgente, i numerosi test di funzionamento sotto stress, il miglioramento delle architetture delle cpu e il rilevamento automatico di molti gravi errori nei driver di Microsoft e di terze parti. Windows ha successivamente esteso gli strumenti per ottenere maggiore affidabilità includendo l'analisi automatica del codice sorgente per la rilevazione di errori, i test in grado di fornire parametri di ingresso non validi o imprevisti (noti come fuzzing) utili per rilevare gli errori di validazione dell'input e una versione applicativa del verificatore di driver che applica il controllo dinamico su una vasta gamma di errori comuni di programmazione in modalità utente. Ulteriori miglioramenti in termini di affidabilità sono dovuti allo spostamento di una quantità maggiore di codice dal kernel ai servizi in modalità utente. Windows fornisce un ampio supporto per lo sviluppo di driver in modalità utente e diversi servizi di sistema che, una volta, erano nel kernel, sono ora eseguiti in modalità utente: tra questi vi sono il dwm e gran parte del software per l'audio.

Uno dei miglioramenti più significativi introdotti grazie all'esperienza nei sistemi Windows consisteva nell'aggiunta dell'opzione di diagnostica della memoria all'avvio, particolarmente preziosa perché ben pochi pc consumer hanno una memoria con correzione d'errore. Quando una ram fallata inizia a perdere bit il risultato è un frustrante comportamento irregolare nel sistema. La disponibilità della diagnostica della memoria può avvisare gli utenti di un problema di ram. Windows 10 ha fatto ancora di più introducendo la diagnostica della memoria al tempo di esecuzione. Se una macchina incorre in un crash in modalità kernel più di cinque volte di seguito e non è possibile individuare la causa degli arresti anomali o il componente che li ha determinati, il kernel inizia a utilizzare i periodi di inattività per spostare il contenuto della memoria, svuotare le cache del sistema e scrivere informazioni di test ripetute in tutta la memoria, al fine di scoprire preventivamente se la ram è danneggiata. Gli utenti possono quindi essere informati di eventuali problemi senza la necessità di un riavvio per eseguire lo strumento di diagnostica della memoria.

Windows 7 ha introdotto una memoria heap con tolleranza agli errori. L'heap apprende dai crash di un'applicazione e inserisce in maniera automatica alcune forme di mitigazione nelle sue future esecuzioni, rendendo così l'applicazione più affidabile anche se contiene bug comuni come l'utilizzo di memoria già liberata o l'accesso alla memoria oltre il limite dell'allocazione. Poiché tali bug possono essere sfruttati per effettuare attacchi, Windows 7 include anche una mitigazione per gli sviluppatori che consente di bloccare questa funzionalità e chiudere immediatamente tutte le applicazioni con l'heap danneggiato. Questo esempio rappresenta un caso pratico della dicotomia che esiste tra le esigenze di sicurezza e le esigenze dell'esperienza utente.

Il raggiungimento di un'elevata affidabilità in Windows è particolarmente impegnativo, perché quasi due miliardi di computer eseguono questo sistema operativo. Persino i problemi di affidabilità che interessano solo una piccola percentuale di utenti possono impattare su un numero altissimo di persone. La complessità dell'ecosistema Windows aggiunge ulteriori ostacoli. Milioni di applicazioni diverse, driver e altri software sono costantemente scaricati ed eseguiti su sistemi Windows e naturalmente c'è anche un flusso costante di attacchi malware. Dato che Windows è diventato più difficile da attaccare direttamente, gli attacchi prendono sempre più di mira le applicazioni più diffuse.

Per far fronte a queste problematiche, Microsoft fa sempre più affidamento sulle comunicazioni provenienti dalle macchine dei clienti per raccogliere grandi quantità di dati provenienti da tutto l'ecosistema. I dati possono essere campionati per avere informazioni sulle prestazioni che le macchine ottengono, su quali software sono in esecuzione e sui problemi che si incontrano. Le macchine inviano automaticamente i dati a Microsoft in caso di crash o blocchi delle applicazioni, dei driver o del sistema. Viene inoltre misurata la

frequenza con cui vengono utilizzate le varie funzionalità, e i comportamenti di tipo legacy (vecchi metodi di cui Microsoft sconsiglia l'utilizzo) sono a volte disabilitati, e vengono visualizzati messaggi di avviso se si tenta di utilizzarli di nuovo. In questo modo Microsoft va costruendosi un'immagine sempre più chiara di quello che succede nell'ecosistema di Windows e ciò consente continui miglioramenti tramite aggiornamenti software e fornisce le informazioni che guidano lo sviluppo delle versioni future di Windows.

21.2.3 Compatibilità tra applicazioni in Windows

Come sottolineato in precedenza, Windows xp non è stato soltanto un aggiornamento di Windows 2000: è il sistema sostitutivo di Windows 95/98. Windows 2000 era studiato principalmente per la compatibilità con le applicazioni della clientela professionale, mentre Windows xp prevedeva una compatibilità molto più elevata con le applicazioni di largo consumo eseguibili in Windows 95/98. La compatibilità delle applicazioni è difficile da ottenere, in quanto molte applicazioni verificano preliminarmente una specifica versione di Windows, risentono delle peculiarità nelle implementazioni delle api, possono nascondere bachi latenti che non si manifestavano nel sistema precedente, e così via. Le applicazioni possono inoltre essere state compilate per un set di istruzioni differente. Windows 10 implementa diverse strategie per eseguire le applicazioni nonostante l'incompatibilità.

Come Windows xp, Windows 10 ha uno strato di compatibilità (*shim engine*) che si trova fra le applicazioni e le api di Win32: grazie a tale strato, Windows 10 appare compatibile con le precedenti versioni di Windows, addirittura quasi baco per baco (*bug-for-bug*). Windows 10 viene fornito con un database di shim di oltre 6.500 voci, che descrive particolari comportamenti e modifiche che devono essere apportate alle applicazioni meno recenti. Inoltre, tramite Application Compatibility Toolkit, gli utenti e gli amministratori possono creare i propri database shim. Il meccanismo SwitchBranch di Windows 10 consente agli sviluppatori di scegliere la versione Windows che vorrebbero far emulare all'app Win32, compresi tutti i difetti e/o bug dell'api precedente. La colonna "Contesto del sistema operativo" del Task Manager mostra in quale versione SwitchBranch del sistema operativo è in esecuzione ciascuna applicazione.

Windows xp, come le prime versioni di nt, mantiene il supporto per molte applicazioni a 16 bit usando uno strato di conversione (o *thunking*), chiamato wow32 (Windows-on-Windows 32), che traduce le chiamate delle api a 16 bit in chiamate equivalenti a 32 bit. Analogamente, la versione a 64 bit di Windows 10 fornisce uno strato che converte le chiamate delle api a 32 bit in chiamate native a 64 bit. Infine, la versione per arm64 fornisce un compilatore dinamico jit, che traduce il codice ia-32, chiamato wowa64.

Il modello a sottosistemi di Windows consente il supporto di personalità multiple del sistema operativo, purché le applicazioni vengano ricostruite come applicazioni pe (Portable Executable) con un compilatore Microsoft come Visual Studio e il codice sorgente sia disponibile. Come notato in precedenza, sebbene l'api progettata per Windows sia Win32api, alcune precedenti edizioni di Windows supportavano il sottosistema posix, posix è una specifica standard per unix che consente di ricompilare il software compatibile unix e di eseguirlo senza modifiche su qualsiasi sistema operativo compatibile con posix. Sfortunatamente, Linux si è allontanato sempre più dalla compatibilità posix e le applicazioni Linux fanno ora affidamento su chiamate di sistema specifiche di Linux e miglioramenti a glibc che non sono standardizzati. Inoltre, diventa poco pratico chiedere agli utenti (o persino alle imprese) di ricompilare con Visual Studio ogni singola applicazione Linux che vorrebbero utilizzare; inoltre, le differenze tra i compilatori gcc, CLang e il compilatore C/C++ di Microsoft rendono spesso l'operazione impossibile. Pertanto, anche se il modello a sottosistemi esiste ancora a livello di architettura, l'unico sottosistema che viene portato avanti è il sottosistema Win32 stesso, e la compatibilità con altri sistemi operativi è realizzata attraverso un nuovo modello che utilizza i Pico Provider.

Questo modello significativamente più potente estende il kernel tramite la capacità di inoltrare, o delegare, ogni chiamata di sistema, eccezione, errore, creazione e terminazione di thread e creazione di processi, insieme ad altre operazioni interne, a un driver esterno secondario (il Pico Provider). Questo driver secondario diventa ora il proprietario di tutte queste operazioni e, mentre utilizza ancora lo scheduler e il gestore di memoria di Windows 10 (in maniera simile a un microkernel), può eseguire in proprio l'abi, l'interfaccia alle chiamate di sistema, il parser del formato di file eseguibile, la gestione degli errori di pagina, il caching, il modello di i/o, il modello di sicurezza, e altro ancora.

Windows 10 include uno di questi Pico Provider, chiamato LxCore, che è una reimplementazione del kernel di Linux di molti megabyte (si noti che non è Linux e non ha codice in comune con Linux). Questo driver viene utilizzato dalla funzionalità "Windows Subsystem for Linux", che può essere utilizzata per caricare file binari elf non modificati di Linux senza la necessità del codice sorgente o la ricompilazione in binari pe. Gli utenti di Windows 10 possono eseguire un file system Ubuntu non modificato in modalità utente (e, più recentemente, anche Opensuse e Centos), utilizzando il comando di gestione pacchetti apt-get e eseguendo i pacchetti normalmente. Si noti che la reimplementazione del kernel non è completa: mancano molte chiamate di sistema, così come l'accesso alla maggior parte dei dispositivi, poiché i driver del kernel Linux non possono essere caricati. In particolare, mentre la rete è completamente supportata, così come i dispositivi seriali, non è possibile l'accesso framebuffer alla gui.

Come ulteriore strumento di compatibilità, Windows 8.1 e versioni successive includono anche la funzionalità **Hyper-V per client**, che consente alle applicazioni di ottenere la compatibilità bug-for-bug con Windows xp, Linux e anche dos eseguendo questi sistemi operativi all'interno di una macchina virtuale.

21.2.4 Prestazioni

Windows è stato progettato per fornire elevate prestazioni nei sistemi desktop (fortemente condizionati dalle prestazioni dell'i/o), nei sistemi server (dove la cpu è spesso il collo di bottiglia) e nei grandi ambienti multithread e multiprocessore (in cui la gestione dei lock e delle cache è uno snodo cruciale per la scalabilità). Per mantenere alto il rendimento delle prestazioni, nt utilizzava svariate tecniche, quali esecuzione asincrona dell'i/o, protocolli ottimizzati per le reti, grafica basata sul kernel e sofisticate tecniche di gestione della cache del file system. La progettazione degli algoritmi di gestione della memoria e di sincronizzazione è concepita tenendo presenti le prestazioni legate alle cache e ai multiprocessori.

Windows nt è stato progettato per il multiprocessing simmetrico (smp). In un computer multiprocessore, più thread possono contemporaneamente essere in esecuzione, anche nel kernel. Windows nt utilizza su ogni cpu uno scheduling dei thread basato su priorità con prelazione. A eccezione dei thread in esecuzione nel dispatcher del kernel o a livello di interrupt, i thread di qualsiasi processo in esecuzione su Windows possono subire la prelazione da thread a più alta priorità. Il sistema, quindi, è in grado di fornire risposte rapide (si veda il Capitolo 5).

Windows xp ha ulteriormente migliorato le prestazioni riducendo il percorso del codice nelle funzioni critiche e implementando protocolli di gestione dei lock scalabili, come le code di spinlock e di **pushlock** (versioni di spinlock ottimizzate con funzioni di lock read-write). I nuovi protocolli di locking contribuiscono a ridurre i cicli di bus del sistema; comprendono liste e code prive di lock, l'uso di operazioni atomiche read-modify-write (come l'incremento interallacciato – *interlocked increment*) e altre tecniche avanzate di sincronizzazione. Queste modifiche sono state necessarie perché Windows xp ha aggiunto il supporto per il multithreading simultaneo (smt), oltre a una tecnologia di pipeline massivamente parallela che Intel ha commercializzato con il nome **Hyper Threading**. A causa di questa nuova tecnologia, i normali pc domestici potevano apparire come dotati di due processori. Qualche anno dopo, l'introduzione dei sistemi multicore ha reso i sistemi multiprocessore la norma.

Successivamente, Windows Server 2003, destinato a server multiprocessore di grandi dimensioni, ha utilizzato algoritmi ancora migliori e si è mosso verso strutture dati, lock e cache a livello dei singoli processori; inoltre, ha utilizzato la colorazione delle pagine (un'ottimizzazione delle prestazioni per garantire che l'accesso alle pagine contigue di memoria virtuale ottimizzi l'uso della cache del processore) e il supporto delle macchine numa. Windows xp 64-bit Edition era basato sul kernel di Windows Server 2003, in modo che i primi utenti di sistemi a 64 bit potessero sfruttare questi miglioramenti.

Quando Windows 7 è stato sviluppato sono state approntate diverse modifiche sul versante della computazione. Il numero di cpu e la quantità di memoria fisica disponibile sui sistemi più grandi erano aumentati notevolmente: numerosi sforzi sono stati dunque rivolti al miglioramento della scalabilità del sistema operativo.

L'implementazione di smp in Windows nt utilizzava bitmask per rappresentare collezioni di processori e per individuare, per esempio, su quale insieme di processori poteva essere eseguito un particolare thread. Le maschere erano state progettate per occupare una singola parola di memoria, limitando a 64 il numero di processori supportati all'interno di un sistema a 64 bit, e a 32 quelli supportati in un sistema a 32 bit. Windows 7 ha quindi aggiunto il concetto di **gruppi di processori** per rappresentare un numero arbitrario di cpu fino a 64. Potevano essere creati diversi gruppi di processori, permettendo così di avere più di 64 processori in totale. Si noti che Windows chiama **processore logico** una porzione dell'unità di esecuzione di un processore su cui è possibile fare scheduling e che questo concetto è distinto da quello di processore fisico o core. Quando in questo capitolo ci riferiamo a un "processore", o a una "cpu", intendiamo esattamente un "processore logico" dal punto di vista di Windows. Windows 7 supportava fino a quattro gruppi di processori, per un totale di 256 processori logici, mentre Windows 10 supporta ora fino a 20 gruppi, per un totale di non più di 640 processori logici (quindi, non tutti i gruppi possono essere riempiti completamente).

Tutte queste cpu aggiuntive hanno portato a una maggior contesa per i lock utilizzati per lo scheduling di cpu e memoria. Windows 7 ha differenziato questi lock. Per esempio, prima di Windows 7 lo scheduler di Windows utilizzava un singolo lock per sincronizzare l'accesso alle code contenenti thread in attesa di eventi. In Windows 7 ogni oggetto ha un proprio lock, permettendo un accesso concorrente alle code. Allo stesso modo, il lock globale del gestore degli oggetti, il lock vacb del gestore della cache e il lock pfn del gestore della memoria sincronizzavano l'accesso a strutture dati globali di grandi dimensioni. Tutti sono stati scomposti in più lock su strutture dati più piccole. Inoltre, molti percorsi di esecuzione dello scheduler sono stati riscritti e resi privi di lock. Questo cambiamento ha portato a buone prestazioni in termini di scalabilità anche su sistemi con 256 processori logici.

Altre modifiche sono state dovute alla crescente importanza del supporto al calcolo parallelo. Per anni l'industria informatica è stata dominata dalla legge di Moore (si veda il Paragrafo 1.1.3) che ha portato a una maggiore densità dei transistor che a loro volta consentono frequenze di clock più alte per ogni cpu. La legge di Moore continua a valere, ma ormai sono stati raggiunti i limiti che impediscono alla frequenza di clock della cpu di crescere ulteriormente. I transistor vengono ora utilizzati per costruire sempre più cpu all'interno di ogni chip. Nuovi modelli di programmazione per realizzare il parallelismo, come Concurrency RunTime (Concr) e Parallel Processing Library (ppl) di Microsoft, oltre a Threading Building Blocks di Intel (tbb), vengono utilizzati per implementare il parallelismo nei programmi C++. Anche se la legge di Moore ha governato l'informatica per 40 anni, sembra ora che la legge di Amdahl, che disciplina il calcolo parallelo, sia destinata al futuro.

Infine, gli aspetti riguardanti la gestione energetica hanno complicato le decisioni di progettazione relative all'elaborazione ad alte prestazioni, specialmente nei sistemi mobili, dove la durata della batteria può diventare più importante delle esigenze di prestazioni, ma anche in ambienti cloud/server, dove il costo dell'elettricità può pesare di più rispetto alla necessità di avere le più alte velocità di computazione possibili. In accordo con queste considerazioni, Windows 10 supporta ora funzionalità che a volte possono sacrificare le prestazioni per una migliore efficienza energetica, come, per esempio, Core Parking, che mette un sistema inattivo in uno stato di sospensione e hmp (Heterogeneous Multi Processing), che assegna le attività in modo efficiente tra i core.

Per supportare il parallelismo a livello di task, i porting amd64 di Windows 7 e versioni successive forniscono una nuova forma di **scheduling in modalità utente** (ums) che permette ai programmi di essere scomposti in task poi pianificati sulle cpu disponibili da uno scheduler che opera in modalità utente anziché nel kernel.

L'utilizzo di più cpu sui piccoli computer è solo una parte del cambiamento che sta avvenendo nel calcolo parallelo. Le unità di elaborazione grafica (gpu) accelerano gli algoritmi di calcolo necessari per la grafica utilizzando architetture simd per eseguire una singola istruzione su più dati allo stesso tempo. Tale potenzialità ha condotto all'utilizzo delle gpu per calcoli generici e non solo per la grafica. Il supporto del sistema operativo a software come Opencl e cuda sta permettendo ai programmi di sfruttare le gpu. Windows supporta l'uso delle gpu attraverso il software presente nel suo supporto grafico DirectX. Questo software, chiamato DirectCompute, permette ai programmi di specificare **nuclei computazionali** utilizzando lo stesso modello di programmazione hlsl (*high-level shader language*) impiegato per programmare l'hardware simd per shader grafici. I nuclei computazionali vengono eseguiti molto velocemente sulla gpu e restituiscono i loro risultati alla computazione principale in esecuzione sulla cpu. In Windows 10, lo stack grafico nativo e molte nuove applicazioni Windows utilizzano DirectCompute e le nuove versioni di Task Manager monitorano l'utilizzo del processore e della memoria della gpu, mentre DirectX dispone ora di un proprio scheduler di thread gpu e di un gestore di memoria della gpu.

21.2.5 Estendibilità

L'**estendibilità** si riferisce alla capacità di un sistema operativo di tenere il passo con gli sviluppi della tecnologia informatica. Il sistema Windows ha una struttura stratificata che facilita eventuali cambiamenti nel corso del tempo. Il suo modulo "executive", che esegue in modalità kernel, fornisce i servizi di sistema fondamentali e le astrazioni a supporto di un uso condiviso del sistema; sopra l'executive operano molti sottosistemi server eseguiti in modalità utente, fra i quali vi erano i sottosistemi d'ambiente che simulavano

differenti sistemi operativi, oggi deprecati. Anche nel kernel, Windows utilizza un'architettura stratificata, con l'uso di driver caricabili nel sistema di i/o, in modo da poter aggiungere nuovi file system, nuovi tipi di dispositivi di i/o e nuovi tipi di networking mentre il sistema è in esecuzione. I driver non sono tuttavia limitati a fornire funzionalità di i/o. Come abbiamo visto, anche un Pico Provider è un tipo di driver caricabile (come la maggior parte dei driver anti-malware). Tramite Pico Provider e la struttura modulare del sistema è possibile aggiungere ulteriore supporto di sistema operativo senza influire sul modulo executive. La Figura 21.1 mostra l'architettura del kernel e dei sottosistemi di Windows 10.

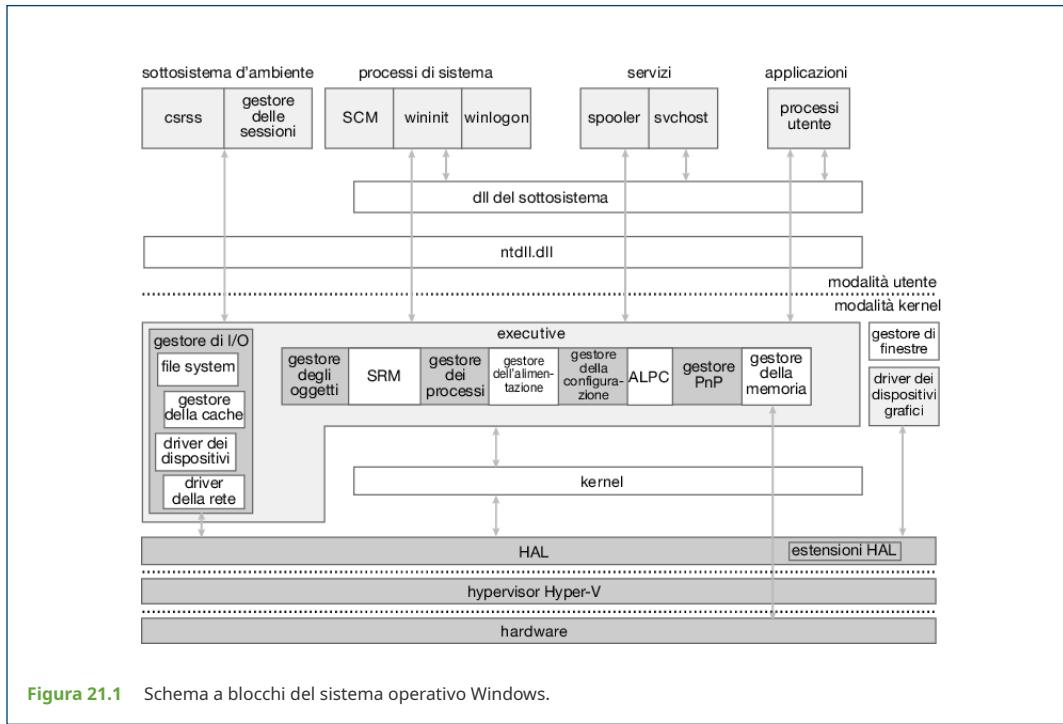


Figura 21.1 Schema a blocchi del sistema operativo Windows.

Il sistema Windows, come il Mach, adotta un modello client-server e gestisce l'elaborazione distribuita per mezzo di **chiamate di procedura remota** (rpc) secondo le specifiche dell'Open Software Foundation. Queste rpc si avvalgono di un componente dell'executive, denominato **chiamata di procedura locale avanzata** (alpc), che implementa comunicazioni altamente scalabili tra processi distinti su una macchina locale. Una combinazione di pacchetti tcp/ip e named pipe sul protocollo smb viene utilizzata per la comunicazione tra i processi in una rete. Basandosi su rpc, Windows implementa l'infrastruttura dcom (Distributed Common Object Model), il wmi (Windows Management Instrumentation) e il meccanismo Windows Remote Management (Winrm), utilizzabili per estendere rapidamente il sistema con nuovi servizi e funzionalità di gestione.

21.2.6 Portabilità

Un sistema operativo è **portabile** (*portable*) se, per poter essere eseguito in un'altra architettura, richieda un numero di modifiche relativamente piccolo; Windows è progettato per essere portabile. Così come per il sistema operativo unix, la maggior parte del sistema Windows è scritta in linguaggio C o in C++.

Il codice sorgente specifico per l'architettura è relativamente piccolo e si fa molto poco uso di codice assembly. Il porting di Windows su una nuova architettura riguarda soprattutto il kernel, dato che il codice in modalità utente viene scritto quasi esclusivamente per essere indipendente dall'architettura. Per il porting di Windows deve essere effettuato il porting del codice del kernel specifico per l'architettura e, a causa dei cambiamenti nelle principali strutture dati, come il formato della tabella delle pagine, è talvolta necessaria una compilazione condizionale in altre parti del kernel. L'intero sistema Windows deve poi essere ricompilato per il nuovo set di istruzioni della cpu.

I sistemi operativi sono sensibili non solo all'architettura della cpu, ma anche ai chip di supporto alla cpu e ai programmi di boot. L'insieme formato da cpu e chip di supporto è noto complessivamente come **chipset**. I chipset e il codice di avvio a essi associato determinano la modalità di emissione degli interrupt, descrivono le caratteristiche fisiche di ogni sistema e forniscono interfacce per gli aspetti più profondi dell'architettura della cpu, come la gestione degli errori e dell'alimentazione. Sarebbe davvero oneroso effettuare il porting di Windows su ciascun tipo di chip di supporto e su ogni architettura di cpu. Per ovviare a questo problema, Windows isola la maggior parte del codice dipendente dal chipset in una libreria dinamica (dll) che costituisce lo **strato di astrazione dell'hardware** (hal) e che viene caricata con il kernel.

Il kernel di Windows dipende dalle interfacce hal piuttosto che dai dettagli del chipset sottostante. Ciò permette di utilizzare il singolo insieme di file binari del kernel e dei driver per una particolare cpu con diversi chipset semplicemente caricando una versione diversa

dell'hal. Originariamente, per supportare le numerose architetture utilizzate da Windows e le numerose marche e tipologie di computer sul mercato, esistevano oltre 450 hal diversi. Nel corso del tempo, l'avvento di standard quali l'interfaccia avanzata di configurazione e alimentazione (acpi), la crescente similarità dei componenti disponibili sul mercato e la fusione dei produttori di computer ha portato a cambiamenti; oggi, il porting amd64 di Windows 10 viene fornito con un singolo hal. È interessante notare, tuttavia, che tali sviluppi non si sono ancora verificati nel mercato dei dispositivi mobili. Oggi, Windows supporta un numero limitato di chipset arm e deve avere il codice hal appropriato per ognuno di essi. Per evitare di tornare a un modello con hal multipli, Windows 8 ha introdotto il concetto di estensioni di hal, che sono dll caricate dinamicamente dall'hal in base ai componenti SoC (system on a chip) rilevati, come il controller delle interruzioni, il gestore del timer e il controller dma.

Nel corso degli anni Windows è stato portato su una serie di diverse architetture di cpu: cpu a 32 bit compatibili con ia-32 di Intel, cpu a 64 bit compatibili con ia64 o amd64, dec Alpha, mips e Powerpc. La maggior parte di queste architetture è scomparsa dal mercato e quando Windows 7 è stato rilasciato solo le architetture ia-32 e amd64 sono state supportate sui computer client e le architetture amd64 sui server. Con Windows 8 sono state aggiunte le architetture arm a 32 bit, mentre Windows 10 supporta anche arm64.

21.2.7 Funzionalità di adattamento internazionale

Windows è anche stato concepito per un'utenza internazionale. Si adatta alle lingue locali grazie all'api per la **gestione delle lingue nazionali** (*national language support, nls*), che fornisce procedure specializzate per trattare date, orari e valute in accordo agli usi dei diversi paesi. I confronti fra sequenze di caratteri tengono conto dei diversi alfabeti. Il codice dei caratteri proprio di Windows è unicode, nello specifico il suo formato di codifica utl-16E (diverso dallo standard utf-8 utilizzato da Linux e dal Web). I caratteri ansi sono convertiti in unicode (da 8 a 16 bit) prima della manipolazione.

Le stringhe contenenti informazioni di sistema si trovano in file sostituibili per adattare il sistema a lingue diverse. Prima di Windows Vista, Microsoft collocava queste tabelle delle risorse all'interno delle dll stesse: esistevano quindi file binari eseguibili distinti per ogni versione diversa di Windows ed era disponibile solo una lingua alla volta. Con il **supporto di più interfacce utente** (mui) di Windows Vista, possono essere utilizzate contemporaneamente diverse impostazioni locali, il che è importante per gli individui e le aziende multilingue. Ciò è stato ottenuto spostando tutte le tabelle delle risorse in file .mui separati che risiedono nella directory della lingua appropriata insieme al file .dll. Il loader offre il supporto per scegliere il file appropriato in base alla lingua correntemente selezionata.

21.2.8 Efficienza energetica

Aumentare l'efficienza energetica dei computer fa sì che le batterie di computer portatili e netbook durino più a lungo, permette di risparmiare notevolmente sui costi di esercizio e di raffreddamento dei data center e contribuisce a iniziative ecologiche volte a ridurre il consumo di energia da parte delle imprese e dei consumatori. Da qualche tempo Windows ha messo in atto diverse strategie per ridurre il consumo di energia. Quando possibile, le cpu vengono impostate in uno stato di potenza inferiore, per esempio abbassando la frequenza di clock. Inoltre, quando un computer non viene attivamente utilizzato Windows può mettere l'intero sistema in uno stato di basso consumo (sleep) o persino salvare tutta la memoria su disco e spegnere il computer (sospensione). Al ritorno dell'utente, il computer si accende e continua dal suo stato precedente, senza necessità di riavviare sistema e applicazioni.

Quanto più a lungo una cpu può rimanere inutilizzata, tanta più energia può essere risparmiata. Poiché i computer sono molto più veloci degli esseri umani, molta energia può essere risparmiata proprio mentre gli umani stanno pensando. Un problema è che molti programmi eseguono continue interrogazioni (polling) in attesa del completamento di un'attività e i timer software scadono spesso, non permettendo alla cpu di rimanere inattiva per un tempo sufficiente a risparmiare tanta energia quanta sarebbe possibile.

Windows 7 estende il tempo di inattività della cpu distribuendo gli interrupt del clock solo alla cpu logica 0 e alle altre cpu attive in quel momento (ignorando quelle inattive) e combinando i timer software idonei in un numero più piccolo di eventi. Sui sistemi server, alcune cpu possono restare completamente inutilizzate qualora i sistemi non fossero eccessivamente caricati. Inoltre, la scadenza dei timer non viene distribuita e una singola cpu è in genere responsabile della gestione di tutte le scadenze dei timer software. Un thread che era in esecuzione, per esempio, sulla cpu logica 3, non riattiva la cpu 3 per gestire la scadenza di un timer se questa è inattiva, quando un'altra cpu, non inattiva, può gestirla.

Sebbene queste misure abbiano aiutato, non sono state sufficienti per aumentare la durata della batteria nei sistemi mobili come i telefoni, dotati di una frazione della capacità della batteria dei computer portatili. Windows 8 ha quindi introdotto una collezione di funzioni per ottimizzare ulteriormente la durata della batteria. Innanzitutto, il modello di programmazione Winrt non consente timer precisi con un tempo di scadenza garantito. Tutti i timer registrati tramite la nuova api sono candidati a essere combinati, a differenza dei timer Win32, in cui questa opzione doveva essere attivata manualmente. In aggiunta, è stato introdotto il concetto di **tick dinamico**, in cui la cpu0 non è più proprietaria del clock, ma questa responsabilità è assunta dall'ultima cpu attiva.

Ancora più importante, l'intero modello di applicazione Metro/Modern/uwp fornito tramite Windows Store include una funzionalità, il **plm (Process Lifetime Manager)**, che sospende automaticamente tutti i thread in un processo che è rimasto inattivo per più di qualche secondo. Ciò non solo attenua i comportamenti di polling costante di molte applicazioni, ma rimuove anche la possibilità per le applicazioni uwp di eseguire il proprio lavoro in background (come l'interrogazione della posizione gps), costringendole a passare per un sistema di agenti in grado di combinare in modo efficiente audio, posizione, download e altre richieste e di memorizzare nella cache i dati mentre il processo resta sospeso.

Infine, utilizzando un nuovo componente chiamato **dam (Desktop Activity Moderator)**, Windows 8 e le versioni successive supportano un nuovo tipo di stato del sistema denominato **Sospensione Connessa** (Connected Standby). Quando si sospende un computer, dopo alcuni secondi, tutto si spegne e l'hardware diviene inattivo. Premendo un pulsante sulla tastiera si riattiva il computer e dopo alcuni secondi l'attività riprende. Su un telefono o su tablet, tuttavia, non è possibile che il dispositivo impieghi secondi per sospendersi: gli utenti vogliono che lo schermo si spenga immediatamente. Se Windows si limitasse a spegnere lo schermo, tuttavia, tutti i programmi continuerebbero a funzionare e le applicazioni legacy Win32, prive di plm e di possibilità di combinare i timer, continuerebbero a effettuare polling, magari risvegliando di nuovo lo schermo. In questo caso, la carica della batteria si consumerebbe in modo significativo.

La Sospensione Connessa risolve questo problema bloccando virtualmente il computer quando si preme il pulsante di alimentazione o quando si spegne lo schermo, senza mettere realmente il computer in stato di sospensione. Il clock hardware viene arrestato, tutti i processi e i servizi vengono sospesi e tutte le scadenze dei timer vengono ritardate di 30 minuti. L'effetto complessivo, anche se il computer è ancora in esecuzione, è che funziona in uno stato di inattività quasi totale e il processore e le periferiche possono effettivamente funzionare nel loro stato di più basso consumo energetico. Per supportare completamente questa modalità sono richiesti hardware e firmware speciali; per esempio, l'hardware del tablet Surface include questa funzionalità.

21.2.9 Supporto dinamico dei dispositivi

All'inizio della storia dell'industria dei pc le configurazioni di un computer erano abbastanza statiche; al massimo potevano essere occasionalmente collegati dei nuovi dispositivi alle porte seriali o alle porte per stampante o di gioco presenti sul retro del pc. I primi passi verso la configurazione dinamica dei pc furono i dock per computer portatili e le schede pcmcia: un pc poteva improvvisamente essere collegato o scollegato da tutta una serie di periferiche. I pc di oggi sono progettati per consentire agli utenti di collegare e scollegare frequentemente una grande quantità di periferiche.

Nel sistema Windows il supporto per la configurazione dinamica dei dispositivi è in continua evoluzione. Il sistema può riconoscere automaticamente i dispositivi quando sono collegati e può trovare, installare e caricare i driver appropriati, spesso senza l'intervento dell'utente. Quando i dispositivi vengono scollegati, i driver sono automaticamente scaricati e l'esecuzione del sistema continua senza interrompere altri software. Inoltre, Windows Update consente il download di driver di terze parti direttamente tramite Microsoft, evitando l'uso di dvd di installazione o l'accesso al sito web del produttore.

Oltre alle periferiche, Windows Server supporta anche l'aggiunta e la sostituzione dinamica a caldo di cpu e ram, nonché la rimozione dinamica della ram. Queste funzionalità consentono di aggiungere, sostituire o rimuovere componenti senza l'interruzione del sistema. Pur essendo di uso limitato nei server fisici, questa tecnologia è fondamentale per la scalabilità dinamica nel cloud computing, in particolare in IaaS (Infrastructure as-a-Service) e negli ambienti di cloud computing. In questi scenari, una macchina fisica può essere configurata per supportare un numero limitato di processori sulla base di un costo di servizio, che può quindi essere aggiornato dinamicamente, senza richiedere il riavvio, tramite un hypervisor compatibile come Hyper-V e con una semplice interazione sull'interfaccia utente del proprietario.

21.3 Componenti del sistema

L'architettura di Windows è un sistema stratificato di moduli che operano a specifici livelli di privilegio, come mostrato nella Figura 21.1. Per impostazione predefinita, questi livelli di privilegio vengono prima implementati dal processore (fornendo un isolamento "verticale" tra la modalità utente e la modalità kernel). Windows 10 può anche utilizzare il proprio hypervisor Hyper-V per fornire un modello di sicurezza ortogonale (logicamente indipendente) tramite i **vtl** (**Virtual Trust Level**). Quando gli utenti abilitano questa funzione, il sistema funziona in modalità vsm (Virtual Secure Mode), in cui il sistema stratificato ha ora due implementazioni, una chiamata **Normal World**, o **vtl 0**, e una chiamata **Secure World**, o **vtl 1**. In ognuno di questi mondi, troviamo una modalità utente e una modalità kernel.

Diamo uno sguardo a questa struttura in maniera più dettagliata.

- In Normal World, sono in modalità kernel (1) l'hal e le sue estensioni e (2) il kernel e il suo executive, che carcano i driver e le dipendenze dalle dll. Nella modalità utente sono presenti una raccolta di processi di sistema, il sottosistema Win32 e vari servizi.
- In Secure World, se vsm è abilitato, vi sono un kernel sicuro e il relativo executive (all'interno del quale è incorporato un micro-hal sicuro), mentre in modalità utente protetta viene eseguita una raccolta di **Trustlet** isolati (che tratteremo in seguito).
- Infine, il livello più basso in Secure World viene eseguito in una modalità di processore speciale (chiamata, per esempio, modalità vmx Root su processori Intel), che contiene il componente hypervisor Hyper-V, che utilizza la virtualizzazione dell'hardware per costruire il confine tra Normal World e Secure World (mentre quello tra kernel e utente è fornito dalla cpu in modo nativo).

Uno dei principali vantaggi di questo tipo di architettura è che le interazioni tra i moduli e tra i livelli di privilegio sono mantenute semplici e che le esigenze di isolamento e di sicurezza non sono necessariamente connesse ai privilegi. Per esempio, un componente sicuro e protetto che memorizza le password può essere di per sé non privilegiato. In passato, i progettisti dei sistemi operativi hanno scelto di soddisfare le esigenze di isolamento rendendo il componente sicuro altamente privilegiato, ma ciò si traduce in una perdita netta per la sicurezza del sistema quando tale componente viene compromesso.

Il resto di questo paragrafo descrive gli strati e i sottosistemi che abbiamo qui introdotto.

21.3.1 Hypervisor Hyper-V

L'hypervisor è il primo componente inizializzato su un sistema con vsm abilitato, e l'inizializzazione avviene non appena l'utente abilita il componente Hyper-V. L'hypervisor viene utilizzato sia per fornire funzionalità di virtualizzazione dell'hardware per l'esecuzione di macchine virtuali separate che per fornire la separazione vtl e il relativo accesso alla funzionalità slat (Second Level Address Translation) dell'hardware (discussa a breve). L'hypervisor utilizza un'estensione della virtualizzazione specifica della cpu, come Pacifica (svmx) di amd o Vanderpool (vt-x) di Intel, per intercettare qualsiasi interrupt, eccezione, accesso alla memoria, istruzione, porta o accesso al registro che si desideri e per rifiutare, modificare o reindirizzare l'effetto, l'origine o la destinazione dell'operazione. L'hypervisor fornisce anche un'interfaccia **hypercall**, che consente di comunicare con il kernel in vtl 0, il kernel sicuro in vtl 1 e tutti gli altri kernel e kernel sicuri di macchine virtuali in esecuzione.

21.3.2 Secure Kernel

Il kernel sicuro funge da ambiente in modalità kernel per applicazioni Trustlet isolate in modalità utente (vtl 1) (applicazioni che implementano parti del modello di sicurezza di Windows). Fornisce la stessa interfaccia di chiamata di sistema del kernel, in modo che tutti gli interrupt, le eccezioni e i tentativi di entrare in modalità kernel da un Trustlet vtl 1 portino a un ingresso nel kernel sicuro. Tuttavia, il kernel sicuro non è coinvolto nel cambio di contesto, nella schedulazione dei thread, nella gestione della memoria, nella comunicazione tra processi e in nessuno degli altri compiti del kernel standard. Inoltre, non sono presenti driver in modalità kernel in vtl 1. Nel tentativo di ridurre la superficie di attacco di Secure World, queste complesse implementazioni rimangono di competenza dei componenti di Normal World. Quindi, il kernel sicuro agisce come un "kernel proxy" che affida la gestione delle sue risorse, della paginazione, dello scheduling e di altro ai normali servizi del kernel in vtl 0. Questa implementazione rende il Secure World vulnerabile agli attacchi DoS, ma si tratta di un compromesso ragionevole nel progetto di sicurezza, che valorizza la privacy e l'integrità dei dati rispetto alle garanzie di servizio.

Oltre a inoltrare le chiamate di sistema, il kernel sicuro fornisce l'accesso alle chiavi segrete hardware, al modulo tpm (Trusted Platform Module) e alle politiche di integrità del codice attivate all'avvio. Con queste informazioni, i Trustlet possono crittografare e decrittografare i dati con chiavi che il Normal World non può ottenere e possono firmare e attestare rapporti (co-firmati da Microsoft) con token di integrità che non possono essere simulati o replicati al di fuori di Secure World. Usando una funzione della cpu chiamata slat (Second Level Address Translation), il kernel sicuro offre anche la capacità di allocare la memoria virtuale in modo tale che le pagine fisiche che lo supportano non possano essere viste dal Normal World. Windows 10 utilizza queste funzionalità per fornire una protezione aggiuntiva delle credenziali aziendali attraverso la Credential Guard.

Inoltre, quando Device Guard (menzionato in precedenza) è attivato, sfrutta le funzionalità di vtl 1 spostando tutto il controllo della firma digitale nel kernel sicuro. Ciò significa che, anche se viene attaccato attraverso una vulnerabilità del software, il kernel normale non può essere forzato a caricare driver non firmati, poiché il confine vtl 1 dovrebbe essere violato affinché ciò avvenga. Su un sistema protetto da Device Guard, affinché una pagina in modalità kernel in vtl 0 sia autorizzata all'esecuzione, il kernel deve prima chiedere l'autorizzazione al kernel sicuro, e solo il kernel sicuro può concedere l'accesso in esecuzione a questa pagina. Installazioni più sicure (nei sistemi embedded o ad alto rischio) possono richiedere questo livello di convalida della firma anche per le pagine in modalità utente. Inoltre, si sta lavorando per consentire a classi speciali di dispositivi hardware, come webcam usb e lettori di smartcard, di essere gestiti direttamente dai driver in modalità utente in esecuzione in vtl 1 (utilizzando la struttura umdf descritta in seguito), in modo da garantire che i dati biometrici siano catturati in sicurezza in vtl 1 senza che alcun componente del Normal World sia in grado di intercettarli. Attualmente, gli unici Trustlet autorizzati sono quelli che forniscono l'implementazione firmata da Microsoft di Credential Guard e del supporto Virtual-tpm. Le versioni più recenti di Windows 10 supporteranno anche le Enclaves vsm, che

consentiranno al codice di terze parti validato (ma non necessariamente firmato da Microsoft) di eseguire i propri calcoli crittografici. Le enclave software consentiranno alle normali applicazioni vti 0 di effettuare chiamate in un'enclave, che eseguirà il codice sui dati di input e restituirà i dati di output, presumibilmente crittografati.

Per maggiori informazioni sul kernel sicuro, consultare <https://blogs.technet.microsoft.com/ash/2016/03/02/windows-10-device-guard-and-credential-guard-demystified/>.

21.3.3 Strato di astrazione dell'hardware

Lo strato di astrazione dell'hardware (hal) è uno strato software che nasconde agli strati superiori le differenze presenti nel chipset, contribuendo alla portabilità del sistema operativo. Lo hal realizza una macchina virtuale usata come interfaccia dal dispatcher del kernel, dall'executive e dai driver dei dispositivi; è sufficiente una sola versione di ogni driver per ogni architettura della cpu, indipendentemente dai chip di supporto utilizzati. I driver mappano i dispositivi e vi accedono direttamente, ma i dettagli specifici del chipset legati alla mappatura in memoria, alla configurazione dei bus per l'i/o, al dma, alla gestione di funzionalità specifiche della scheda madre, sono tutti curati dalle interfacce hal.

21.3.4 Kernel

Il livello kernel di Windows ha quattro compiti principali: lo scheduling dei thread e il cambio di contesto, la sincronizzazione a basso livello del processore, la gestione delle interruzioni e delle eccezioni e il passaggio tra la modalità utente e la modalità kernel. Inoltre, il livello del kernel implementa il codice iniziale che prende il controllo dal boot loader, formalizzando la transizione verso il sistema operativo Windows, oltre a implementare il codice iniziale che blocca in modo sicuro il kernel in caso di eccezioni inaspettate o altre incoerenze impreviste. Il kernel è implementato in linguaggio C, con l'utilizzo del linguaggio assembly solo nei casi in cui sia assolutamente necessario per interfacciarsi con i livelli più bassi dell'architettura e per un accesso diretto ai registri.

21.3.4.1 Dispatcher del kernel

Il dispatcher del kernel costituisce il fondamento per l'executive e i sottosistemi. La gran parte del dispatcher non è mai rimossa dalla memoria centrale, e la sua esecuzione non è mai soggetta a prelazione. I suoi compiti principali sono lo scheduling dei thread, il cambio di contesto, l'implementazione delle primitive di sincronizzazione, la gestione del timer, delle interruzioni software (chiamate di procedura asincrone e differite), degli **interrupt tra processi (ipi)** e la consegna (*dispatching*) delle eccezioni. Il dispatcher gestisce inoltre le priorità degli interrupt hardware e software col sistema dei **livelli di richiesta di interrupt (irql)**

21.3.4.2 Comutazione tra thread in modalità utente e thread in modalità kernel

Quando un programmatore Windows pensa a un thread in realtà sta pensando a due thread: un **thread in modalità utente** (ut) e un **thread in modalità kernel** (kt). Ognuno di questi thread ha il proprio stack, i propri valori dei registri e il proprio contesto di esecuzione. Un ut richiede un servizio di sistema eseguendo un'istruzione che provoca una trap alla modalità kernel. Lo strato kernel esegue quindi un gestore di trap che passa dal thread ut al corrispondente thread kt. Quando un kt ha completato l'esecuzione nel kernel ed è pronto a tornare al corrispondente ut, viene invocato lo strato kernel per effettuare il passaggio all'ut, che continua la sua esecuzione in modalità utente. La commutazione a un thread kt avviene anche in caso di interrupt.

Windows 7 modifica il comportamento dello strato kernel per supportare lo scheduling in modalità utente dei thread ut. Lo scheduler in modalità utente di Windows 7 supporta lo scheduling cooperativo. Da un ut si può passare esplicitamente a un altro ut chiamando lo scheduler in modalità utente, senza intervento del kernel. Lo scheduling in modalità utente è descritto in dettaglio nel Paragrafo 21.7.3.7.

In Windows, il dispatcher non è un thread separato in esecuzione nel kernel. Piuttosto, il codice del dispatcher viene eseguito dal componente kt di un thread ut. Un thread entra nella modalità kernel nelle stesse circostanze in cui, in altri sistemi operativi, viene chiamato un thread del kernel. Queste stesse circostanze fanno sì che un kt esegua il codice del dispatcher dopo le sue altre operazioni, determinando quale thread eseguire successivamente sul core corrente.

21.3.4.3 Thread

Come molti altri moderni sistemi operativi, il sistema Windows associa le nozioni di processo e thread al codice eseguibile. Ogni processo ha uno o più thread, cioè unità d'esecuzione elaborate dal kernel; ogni thread possiede un suo stato, che comprende una priorità, dei gradi di affinità relativi alle unità d'elaborazione, e informazioni sull'uso della cpu.

Gli otto possibili stati di un thread sono: in inizializzazione, pronto, pronto differito, in standby, in esecuzione, in attesa, in transizione, e terminato. Un thread è pronto quando attende d'essere eseguito, mentre è pronto differito quando è stato selezionato per l'esecuzione, ma non è ancora stato schedulato. Un thread è in esecuzione quando un'unità d'elaborazione lo sta eseguendo; rimane in questo stato finché non viene interrotto da un thread con priorità più alta, o termina, o il suo quanto di tempo scade, o si blocca in attesa su un oggetto dispatcher – per esempio, un evento che segnala la terminazione di un'operazione di i/o. Se un thread esercita una prelazione su un altro thread su un processore diverso, viene messo nello stato di standby su quel processore e diventa il thread successivo da eseguire.

La prelazione avviene all'istante e il thread corrente non ha la possibilità di completare il suo quanto di tempo. Il processore invia quindi un interrupt software, in questo caso una **chiamata di procedura differita (dpc)**, per segnalare all'altro processore che un thread si trova in standby e deve essere immediatamente prelevato per l'esecuzione. È interessante notare che anche un thread nello stato di standby può subire una prelazione se un altro processore trova un thread con priorità ancora più elevata da eseguire in questo processore. A quel punto, il nuovo thread con priorità più alta andrà in standby e il thread precedente passerà allo stato pronto. Un thread è in attesa se attende la segnalazione di un oggetto dispatcher. Un thread è in stato di transizione quando attende le risorse necessarie all'esecuzione (per esempio, potrebbe essere in attesa che il suo stack venga caricato dal disco alla memoria). Un thread è terminato quando ha completato l'esecuzione. Un thread è nello stato di inizializzazione quando viene creato, prima di essere pronto per la prima volta.

Il dispatcher adotta uno schema con 32 livelli di priorità per determinare l'ordine d'esecuzione dei thread; le priorità si dividono in due classi: la classe a priorità variabile, contenente i thread le cui priorità sono comprese fra 1 e 15, e la classe statica, contenente i thread le cui priorità vanno da 16 a 31. Il dispatcher usa una lista concatenata per ogni livello di priorità; l'insieme di queste liste è il database del dispatcher. Il database utilizza una bitmap per indicare la presenza di almeno una voce nella lista associata alla priorità della posizione del bit. Pertanto, invece di dover attraversare l'insieme di liste dalla priorità più alta alla più bassa finché non viene trovato un thread pronto per essere eseguito, il dispatcher può semplicemente trovare la lista associata al bit più alto.

Prima di Windows Server 2003, il database del dispatcher era globale, con conseguenti pesanti contese su sistemi di grandi dimensioni. In Windows Server 2003 e versioni successive, il database globale è suddiviso in un database per processore, con lock per ogni processore. Con questo nuovo modello, un thread si trova soltanto nel database del suo **processore ideale**, in modo da garantire un'affinità del processore che include il processore sul cui database si trova il thread. Ora il dispatcher può semplicemente selezionare il primo thread nell'elenco associato al bit più alto impostato a 1 e non deve acquisire un lock globale. Il dispatching è quindi un'operazione a tempo costante, parallelizzabile tra tutte le cpu della macchina.

Su un sistema dotato di un singolo processore, se nessun thread è pronto, il dispatcher esegue un thread speciale detto *idle thread*, il cui compito è di iniziare la transizione a uno degli stati di sospensione iniziali della cpu. La classe di priorità 0 è riservata per l'idle thread. Su un sistema multiprocessore, prima di eseguire il thread idle, il dispatcher controlla i database del dispatcher di altri processori vicini, prendendo in considerazione le topologie di caching e le distanze dei nodi numa. Questa operazione richiede l'acquisizione dei lock di altri core al fine di ispezionare in modo sicuro le loro liste. Se nessun thread può essere rubato da un core vicino, il dispatcher controlla il core immediatamente successivo, e così via. Se nessun thread può essere rubato, il processore esegue il thread idle, per cui in un sistema multiprocessore ogni cpu avrà il proprio thread idle.

Collocando ogni thread esclusivamente sul database del dispatcher del suo processore ideale si provoca un problema di località. Si immagini una cpu che esegue un thread cpu-bound con priorità 2, mentre un'altra cpu sta eseguendo un thread con priorità 18, anch'esso cpu-bound. Se un thread con priorità 17 diventa pronto e il suo processore ideale è la prima cpu, il nuovo thread interrompe il thread corrente. Se invece il processore ideale è la seconda cpu, il thread passa alla coda di pronto, in attesa del suo turno di esecuzione (cosa che non succederà fino a quando il thread con priorità 18 non cederà la cpu terminando o passando in uno stato di attesa).

Windows 7 ha introdotto un algoritmo di bilanciamento del carico per affrontare questa situazione, ma si è trattato di un approccio piuttosto dirompente al problema della località. Windows 8 e versioni successive hanno risolto il problema in modo più sfumato. Invece di utilizzare un database globale come Windows xp e versioni precedenti, o un database per processore come in Windows Server 2003 e versioni successive, le nuove versioni di Windows combinano questi approcci per formare una coda ready condivisa all'interno di un gruppo di alcuni processori, ma non tutti. Il numero di cpu che formano un gruppo condiviso dipende dalla topologia del sistema e dal fatto che si tratti di un server o di un sistema client. Il numero viene scelto in modo da mantenere bassa la contesa su sistemi molto grandi, evitando al contempo problemi di località (e quindi latenza e contesa) su sistemi client più piccoli. Inoltre, le affinità con i processori sono ancora rispettate, di modo che a un processore in un dato gruppo venga garantito che tutti i thread nella coda condivisa siano appropriati, e che dunque non abbia mai bisogno di "saltare" un thread, mantenendo così l'algoritmo in tempo costante.

Windows ha un timer che scade ogni 15 millisecondi per creare un "tick" del clock per esaminare gli stati del sistema, aggiornare l'ora e fare altre operazioni di ordinaria amministrazione. Il tick viene ricevuto dal thread presente su ogni core non inattivo. Il gestore degli interrupt (eseguito dal thread, ora in modalità kt) determina se il quanto del thread sia scaduto e quando lo è l'interrupt del clock accoda al processore una dpc di fine del quanto. Questo accodamento porta a un interrupt software quando il processore ritorna alla normale priorità di interrupt. L'interrupt software fa in modo che il thread esegua il codice del dispatcher in modalità kt per riprogrammare il processore affinché esegua il successivo thread disponibile al livello di priorità del thread che subisce la prelazione, in modalità round-robin. Se nessun altro thread con il livello di priorità richiesto è pronto, non viene scelto un thread di livello inferiore, perché ne esiste già uno con priorità più alta: quello che ha appena esaurito il suo quanto. In questa situazione, il quanto di tempo viene semplicemente ripristinato al suo valore predefinito e lo stesso thread viene eseguito nuovamente. Di conseguenza, Windows esegue sempre il thread in stato di pronto con priorità più alta.

Quando un thread a priorità variabile viene risvegliato da un'operazione di attesa, il dispatcher può aumentarne la priorità. L'entità di questo aumento dipende dal tipo di attesa associato al thread. Se l'attesa era dovuta a i/o, l'aumento dipende dal dispositivo per il quale il thread era in attesa. Per esempio, un thread in attesa di i/o audio otterrebbe un aumento di priorità elevato, mentre un thread in attesa di un'operazione sul disco ne otterrebbe uno moderato. Questa strategia consente ai thread i/o-bound di mantenere occupati i dispositivi di i/o, consentendo allo stesso tempo ai thread cpu-bound di utilizzare cicli di cpu d'avanzo in background.

Un altro tipo di aumento di priorità è applicato ai thread che attendono su oggetti mutex, semafori o sincronizzazione di eventi. Questo aumento di solito è un valore codificato di un singolo livello di priorità, sebbene i driver del kernel abbiano la possibilità di apportare modifiche diverse (per esempio, il codice della gui in modalità kernel applica un aumento di due livelli di priorità a tutti i thread della gui che si attivano per elaborare i messaggi della finestra). Questa strategia viene utilizzata per ridurre la latenza tra quando viene segnalato un lock o un altro meccanismo di notifica e quando il successivo thread in attesa viene eseguito in risposta al cambiamento di stato.

Inoltre, il thread associato alla finestra della gui attiva dell'utente riceve un incremento di priorità di due quando si risveglia per qualsiasi motivo, in aggiunta a qualsiasi altro aumento esistente, per migliorare il tempo di risposta. Questa strategia, chiamata **aumento separato della priorità in primo piano**, tende a fornire buoni tempi di risposta ai thread interattivi.

Infine, Windows Server 2003 ha aggiunto un aumento lock-handoff per alcune classi di lock, come le sezioni critiche. Questo aumento è simile a quello di mutex, semafori ed eventi, tranne per il fatto che tiene traccia del possesso del lock e invece di potenziare il thread in fase di risveglio con un valore pari a un singolo livello di priorità, lo eleva a un livello di priorità superiore a quello del proprietario corrente del lock (quello che lo rilascia). Ciò è d'aiuto in situazioni in cui, per esempio, un thread con priorità 12 rilascia un mutex, ma il thread in attesa ha priorità 8. Se il thread in attesa riceve un aumento che lo porta ad avere priorità 9, non sarà in grado di esercitare la prelazione sul thread che rilascia il lock, ma se la sua priorità viene aumentata fino a 13 può esercitarla e acquisire immediatamente la sezione critica.

Poiché i thread possono essere eseguiti con priorità potenziate quando si risvegliano dalle attese, la priorità di un thread viene abbassata alla fine di ogni quanto, se il thread è sopra la sua priorità (iniziale) di base. Ciò avviene in base alla seguente regola: i thread di i/o e i thread potenziati in seguito a un risveglio causato da un evento, un mutex o un semaforo, perdono un livello di priorità alla fine del loro quanto; i thread potenziati a causa di un aumento lock-handoff o di un aumento separato della priorità in primo piano, perdono l'intero valore dell'aumento. I thread che hanno ricevuto aumenti di entrambi i tipi obbediranno a entrambe le regole (perdendo un livello secondo la prima, più quanto richiesto dalla seconda). Abbassando la priorità del thread si assicura che l'incremento sia applicato solo per la riduzione della latenza e per mantenere occupati i dispositivi di i/o senza dare una preferenza di esecuzione indebita ai thread cpu-bound.

21.3.4.4 Scheduling dei thread

Lo scheduling può avvenire quando un thread è pronto o è posto nello stato d'attesa, quando termina, o quando un'applicazione ne cambia il grado di affinità con un'unità d'elaborazione. Come già abbiamo detto in questo libro, un thread può diventare pronto in qualsiasi momento. Se un thread a più alta priorità diviene pronto durante l'esecuzione di un thread a priorità più bassa, quest'ultimo sarà interrotto; ciò fornisce al thread a priorità maggiore un accesso immediato alla cpu, senza necessità di attendere che il thread a priorità più bassa completi il suo quanto.

È lo stesso thread con priorità più bassa a lanciare un evento che lo porta a funzionare nel dispatcher, risvegliando il thread in attesa ed effettuando immediatamente il cambio di contesto, e posizionando se stesso nello stato pronto. Questo modello essenzialmente distribuisce la logica di scheduling in tutte le dozzine di funzioni del kernel di Windows e rende ciascun thread in esecuzione un'entità di scheduling. Al contrario, altri sistemi operativi si basano su un "thread di scheduling" esterno attivato periodicamente da un timer. Il vantaggio dell'approccio Windows è la riduzione della latenza, al costo di un overhead aggiuntivo per tutte le operazioni di i/o o di modifica dello stato, che fanno sì che il thread corrente esegua il lavoro dello scheduler.

Il sistema Windows, tuttavia, non è un sistema operativo per l'elaborazione in tempo reale stretto (*hard*) perché non garantisce che un thread, anche quello con priorità più alta, sia eseguito entro limiti di tempo fissati o abbia un tempo di esecuzione garantito. I thread sono bloccati a tempo indeterminato mentre le dpc e le **routine di servizio dell'interrupt (isr)** sono in esecuzione (come verrà spiegato più avanti), e possono essere soggetti a prelazione in qualsiasi momento da un thread con priorità più alta o essere costretti a un round-robin con un altro thread di uguale priorità alla fine del loro quanto di tempo.

Tradizionalmente gli scheduler del sistema operativo si servivano del campionamento per misurare l'utilizzo della cpu da parte dei thread. Il timer di sistema emetteva un segnale periodico e il gestore degli interrupt del timer prendeva nota di quale thread era in esecuzione al momento dell'interrupt e se era eseguito in modalità kernel o utente. Questa tecnica di campionamento era necessaria perché la cpu non aveva un clock ad alta risoluzione, oppure perché l'accesso al clock era troppo costoso o troppo inaffidabile per poter essere effettuato frequentemente. Anche se efficiente, il campionamento è impreciso e porta ad anomalie come attribuire l'intera durata del clock (15 millisecondi) al thread attualmente in esecuzione (o alle dpc, o alle isr). Il sistema finiva quindi per ignorare completamente alcuni millisecondi, diciamo 14,999, che avrebbero potuto essere spesi come inattivi, eseguendo altri thread, o altre dpc e isr, o una combinazione di tutte queste operazioni. Inoltre, poiché il quanto viene misurato in base ai tick del clock, è possibile che si abbia la prematura selezione round-robin di un nuovo thread, anche se il thread corrente è stato eseguito solo per una frazione del suo quanto di tempo.

A partire da Windows Vista l'uso del tempo di cpu è stato monitorato utilizzando il **contatore timestamp hardware (tsc)**, presente nei processori a partire dai Pentium Pro. L'utilizzo del tsc conferisce una maggiore accuratezza ai calcoli relativi all'utilizzo della cpu (per le applicazioni che lo utilizzano: si noti che il Task Manager non lo fa) e lo scheduler non interrompe più i thread prima che questi abbiano utilizzato completamente il loro quanto di tempo. Inoltre, Windows 7 e versioni successive tracciano mediante il tsc anche le isr e le dpc, ottenendo così misurazioni più accurate del tempo di interrupt (ancora, per gli strumenti che utilizzano questa nuova misurazione). Poiché ora viene contabilizzato tutto il tempo di esecuzione possibile, si può aggiungerlo al tempo di inattività (anch'esso monitorato utilizzando il tsc) e calcolare accuratamente il numero esatto di cicli della cpu utilizzati rispetto a tutti i possibili cicli della cpu in un determinato periodo (a causa del fatto che i processori moderni hanno frequenze che si adattano dinamicamente) ottenendo una misurazione accurata dell'uso dei cicli della cpu. Strumenti come SysInternals Process Explorer di Microsoft utilizzano questo meccanismo nella loro interfaccia utente.

21.3.4.5 Realizzazione delle primitive di sincronizzazione

Windows utilizza un certo numero di **oggetti dispatcher** per controllare il dispatching e la sincronizzazione nel sistema. Eccone alcuni esempi.

- L'oggetto **evento** è usato per registrare il verificarsi di un evento e per sincronizzarlo con una qualche azione. Gli eventi di notifica segnalano tutti i thread in attesa, mentre gli eventi di sincronizzazione segnalano un singolo thread in attesa.
- Il **mutex** fornisce la mutua esclusione in modalità kernel o utente, associata con la nozione di proprietà.
- Il **semaforo** opera come un contatore, per controllare il numero di thread che accede alla risorsa.
- Il **thread** è l'entità schedulata dal dispatcher del kernel ed è associata con un processo, che incapsula uno spazio di indirizzamento virtuale, un elenco di risorse aperte e altro ancora. Il thread riceve un segnale all'uscita del thread e il processo all'uscita del processo (ovvero quando tutti i suoi thread sono terminati).
- Il **timer** viene usato per tenere traccia del tempo e per segnalare timeout quando le operazioni impiegano troppo tempo e devono essere interrotte, oppure quando deve essere schedulata un'attività periodica. Proprio come gli eventi, i timer possono funzionare in modalità di notifica (signal all) o modalità di sincronizzazione (signal one).

È possibile accedere a tutti gli oggetti del dispatcher dalla modalità utente tramite un'operazione `open` che restituisce un handle. Il codice in modalità utente attende gli handle per la sincronizzazione con altri thread e con il sistema operativo (si veda il Paragrafo 21.7.1).

21.3.4.6 Livelli di richiesta di interrupt (IRQL)

Sia gli interrupt hardware che quelli software sono prioritizzati e vengono gestiti in ordine di priorità. Ci sono 16 livelli di richiesta di interrupt (irq) su tutte le ISA Windows tranne la legacy IA-32, che ne usa 32. Il livello più basso, irql 0, è chiamato `passive_level` ed è il livello predefinito a cui tutti i thread vengono eseguiti, sia nel kernel che in modalità utente. I livelli successivi sono i livelli di

interrupt software per apc e dpc. I livelli da 3 a 10 vengono utilizzati per rappresentare gli interrupt hardware in base alle selezioni effettuate dal gestore PnP con l'aiuto dello hal e dei driver del bus pci/acpi. Infine, i livelli più alti sono riservati all'interrupt del clock (utilizzato per la gestione dei quanti di tempo) e alla consegna degli ipi. L'ultimo livello, high_level, blocca tutti gli interrupt mascherabili e viene in genere utilizzato quando nel caso di un crash controllato del sistema.

Gli irql di Windows sono mostrati nella Figura 21.2.

Livelli di interruzione	Tipi di interruzione
31	controllo di macchina o errore sul bus
30	calo di tensione
29	comunicazione fra unità d'elaborazione (richiesta d'azione a un'altra unità d'elaborazione, ad esempio, avvio di un processo o aggiornamento dei TLB)
28	orologio
27	profile
3-26	ordinarie interruzioni IRQ dei PC
2	dispatch e chiamata di procedura differita (DPC) (kernel)
1	chiamata di procedura asincrona (APC)
0	passiva

Figura 21.2 Livelli dei segnali d'interruzione (irql) di un sistema Windows x86.

21.3.4.7 Interruzioni software: chiamate di procedura asincrona e differita

Il dispatcher implementa due tipi di interruzioni software: la **chiamata di procedura asincrona** (*asynchronous procedure call*, apc) e la **chiamata di procedura differita** (*deferred procedure call*, dpc). apc servono per sospendere o riattivare un thread esistente, per terminare i thread, per consegnare la notifica relativa a un'operazione di i/o asincrona che è stata completata e per estrarre il contenuto dei registri della cpu da un thread in esecuzione. Le apc sono accodate a specifici thread e permettono al sistema di eseguire sia il codice di sistema sia quello utente entro il contesto del processo. L'esecuzione di una apc in modalità utente non può avvenire in qualsiasi momento, ma solamente quando il thread è in attesa nel kernel ed è contrassegnato come *alertable* (allertabile). L'esecuzione in modalità kernel di una apc, al contrario, avviene instantaneamente nel contesto di un thread in esecuzione, poiché viene emessa come interrupt software in esecuzione su irql 1 (apc_level), che è superiore al predefinito irql 0 (passive_level). Inoltre, anche se un thread è in attesa in modalità kernel, l'attesa può essere interrotta dall'apc e ripresa una volta che l'apc completa l'esecuzione.

Le chiamate di procedura differite sono finalizzate a posporre l'elaborazione delle interruzioni. Dopo aver gestito tutti i processi urgenti bloccati da interruzioni dei dispositivi, la procedura di servizio delle interruzioni (*interrupt service routine*, isr) pianifica i lavori di elaborazione rimasti in sospeso accodando una dpc. L'interrupt software associato viene eseguito su irql 2 (dpc_level), che è inferiore rispetto a tutti gli altri livelli di interrupt hardware di i/o in modo che le dpc non bloccino altre isr. Oltre che per rinviare l'elaborazione delle interruzioni dei dispositivi, il dispatcher usa le dpc per elaborare le interruzioni generate dal timer e per interrompere l'esecuzione dei thread il cui quanto di tempo è scaduto.

Poiché irql 2 è più alto di 0 (passive) e di 1 (apc), l'esecuzione delle dpc impedisce ai thread di essere schedulati nel processore corrente, e alle apc di segnalare il completamento di un'operazione di i/o. È di conseguenza importante che le procedure dpc non impieghino quantità di tempo eccessive per essere completate. In alternativa, l'executive mantiene un gruppo di "thread di lavoro". Le dpc possono accodare i lavori da svolgere ai thread di lavoro e questi saranno eseguiti utilizzando il normale scheduling dei thread al livello irql 0. Poiché il dispatcher stesso funziona su irql 2, e poiché le operazioni di paginazione richiedono l'attesa per l'i/o (e ciò riguarda il dispatcher), le dpc sono limitate in modo che non possano incorrere in page fault, richiamare servizi del sistema o compiere qualsiasi altra azione che potrebbe sfociare nel tentativo di sospendere l'esecuzione in attesa di un oggetto dispatcher. A differenza delle apc, che sono legate a un thread, le routine dpc non fanno ipotesi in merito a quale contesto di processo il processore stia eseguendo, poiché eseguono nello stesso contesto del thread attualmente in esecuzione, che è stato interrotto.

21.3.4.8 Eccezioni, interruzioni e IPI

Il dispatcher del kernel fornisce anche la gestione delle eccezioni e delle interruzioni; Windows definisce diverse eccezioni indipendenti dall'architettura, tra cui:

- overflow in operazioni su interi o numeri in virgola mobile;
- divisone per zero, intera o in virgola mobile;
- istruzione illegale;
- dati non allineati;
- istruzione privilegiata;
- violazione d'accesso;
- quota di paginazione ecceduta;
- punto d'arresto (*breakpoint*) del debugger.

I trap handler gestiscono le eccezioni a livello hardware (chiamate **trap**) e richiamano il complesso codice di gestione delle eccezioni eseguito dal dispatcher delle eccezioni nel kernel. Il dispatcher delle eccezioni annota la causa dell'eccezione per individuarne

l'appropriato gestore.

Quando si verifica un'eccezione in modalità kernel, il dispatcher delle eccezioni invoca semplicemente una procedura che ne individua il gestore; nel caso in cui non se ne trovi alcuno avviene un errore di sistema fatale che lascia l'utente davanti al famigerato "schermo blu della morte", il quale sta a significare l'arresto totale del sistema. In Windows 10, la schermata blu è sostituita da una "faccina triste" più amichevole e da un codice qr, ma il colore blu rimane.

La gestione delle eccezioni è più complessa per i processi in modalità utente, poiché il servizio di segnalazione degli errori di Windows (wer) imposta una porta di errore alpc per ogni processo, appoggiandosi sul sottosistema di ambiente Win32 che imposta una porta delle eccezioni alpc per ogni processo che crea (per dettagli sulle porte, si veda il Paragrafo 21.3.5.4.). Inoltre, se un processo è in fase di debug, ottiene una porta debugger. Se una porta debugger è registrata, il gestore delle eccezioni invia l'eccezione alla porta. Se la porta non viene trovata o non gestisce quell'eccezione, il dispatcher tenta di trovare un gestore di eccezioni appropriato e, se non esiste, contatta il gestore di eccezioni non gestito predefinito, che informerà il wer del blocco del processo in modo che possa essere generato e inviato a Microsoft un dump di arresto anomalo. Se c'è un gestore, ma si rifiuta di gestire l'eccezione, il debugger viene chiamato di nuovo per rilevare l'errore per il debug. Se nessun debugger è in esecuzione, viene inviato un messaggio alla porta delle eccezioni del processo per dare al sottosistema d'ambiente la possibilità di reagire all'eccezione. Infine, viene inviato un messaggio al wer attraverso la porta di errore, nel caso in cui il gestore di eccezioni non gestite non abbia avuto la possibilità di farlo, e il kernel termina quindi il processo contenente il thread che ha causato l'eccezione.

Generalmente, il wer invierà le informazioni a Microsoft per ulteriori analisi, a meno che l'utente non abbia disattivato il servizio o stia utilizzando un server di segnalazione degli errori locale. In alcuni casi, l'analisi automatizzata di Microsoft potrebbe essere in grado di riconoscere immediatamente l'errore e suggerire una correzione o una soluzione alternativa.

Il dispatcher delle interruzioni del kernel gestisce le interruzioni richiamando una **procedura di servizio delle interruzioni** (*interrupt service routine*, isr) o una procedura interna del kernel. Un segnale d'interruzione è rappresentato da un **oggetto interruzione** contenente tutte le informazioni necessarie per gestire l'interruzione, il che rende facile associare procedure di servizio a un segnale d'interruzione senza dover accedere direttamente al dispositivo in questione.

Diverse architetture hanno tipi e numeri d'interruzione diversi; per garantire la portabilità il dispatcher delle interruzioni associa le interruzioni a un insieme convenzionale.

Il kernel usa una **tavella di dispatch delle interruzioni** (idt) per associare a ogni livello delle interruzioni una procedura di servizio; nel caso di un multiprocessore, il kernel di Windows mantiene idt separate per ogni processore, e l'irql di ogni processore si può regolare in modo indipendente per mascherare determinate interruzioni. Tutte le interruzioni che occorrono a un livello pari o inferiore dell'irql di un processore sono bloccate finché l'irql non si sia abbassato da un thread a livello del kernel o dal completamento di esecuzione di una isr. Il sistema Windows sfrutta questa caratteristica impiegando le eccezioni per eseguire funzioni di sistema: per esempio, il kernel usa eccezioni per avviare il dispatch di un thread, per gestire i timer e sincronizzare i thread con il completamento degli i/o.

21.3.5 Executive

L'executive del sistema Windows fornisce una serie di servizi utilizzabili da tutti i sottosistemi d'ambiente. Per fornire una buona panoramica di base, discuteremo qui i seguenti servizi: gestore degli oggetti, gestore della memoria virtuale, gestore dei processi, servizi relativi alle chiamate di procedura locali avanzate, gestore dell'i/o, della cache, monitor della sicurezza dei riferimenti, gestori del *plug-and-play* e dell'alimentazione, registri (*registry*) e avvio del sistema.

L'executive è organizzato secondo principi di progettazione orientata agli oggetti. Un **tipo di oggetto** in Windows è un tipo di dati definito dal sistema che ha una serie di attributi (valori dei dati) e un insieme di metodi (per esempio, funzioni o operazioni) che aiutano a definire il comportamento. Un **oggetto** è un'istanza di un tipo di oggetto. L'executive svolge il suo lavoro utilizzando un insieme di oggetti i cui attributi memorizzano i dati e i cui metodi eseguono le attività.

21.3.5.1 Gestore degli oggetti

Per la gestione di entità in modalità kernel, Windows usa un insieme di interfacce manipolate dai programmi in modalità utente. In Windows queste entità sono chiamate oggetti, mentre il componente di codice eseguibile che li gestisce è detto **gestore degli oggetti** (*object manager*). Esempi di oggetti sono: file, chiavi di registro, dispositivi, porte alpc, driver, mutex, eventi, processi e thread. Come abbiamo visto in precedenza, alcuni di questi, come i mutex e i processi, sono tutti *oggetti dispatcher*, il che significa che i thread possono bloccarsi nel dispatcher del kernel, nell'attesa che uno di questi oggetti sia segnalato. Inoltre, la maggior parte degli oggetti non dispatcher include un oggetto dispatcher interno, che viene segnalato dal servizio executive che lo controlla. Per esempio, gli oggetti file hanno un oggetto evento incorporato, che viene segnalato quando un file viene modificato.

Il codice in modalità utente o kernel accede a questi oggetti utilizzando un valore opaco chiamato **handle**, che viene restituito da molte api. Ogni processo ha una tabella degli handle contenente le voci che tengono traccia degli oggetti utilizzati dal processo. Esiste un "processo di sistema" (si veda il Paragrafo 21.3.5.11) che ha la sua tabella degli handle protetta dal codice utente. Le **tabelle degli handle** sono rappresentate in Windows da una struttura ad albero che può contenere da 1.024 fino a oltre 16 milioni di handle. Il codice in modalità kernel può accedere a un oggetto, oltre che utilizzando un handle, mediante un **puntatore di riferimento** (*referenced pointer*), che può essere ottenuto chiamando una speciale api. Dopo l'utilizzo, gli handle devono essere chiusi per evitare di mantenere un riferimento attivo sull'oggetto. Allo stesso modo, quando il codice del kernel usa un puntatore di riferimento, deve usare una speciale api per eliminare il riferimento.

Un processo ottiene un handle creando un oggetto, aprendo un oggetto esistente, ricevendo un handle duplicato da un altro processo o ereditando un handle dal processo padre. Per aggirare il problema che gli sviluppatori potrebbero dimenticare di chiudere i loro handle, tutti gli handle aperti di un processo vengono chiusi in modo implicito quando il processo viene chiuso o terminato. Tuttavia, poiché gli handle del kernel appartengono alla tabella degli handle di sistema, nel caso di un driver gli handle non vengono automaticamente chiusi e questo può portare a perdite di risorse sul sistema.

Dal momento che il gestore degli oggetti è l'unica entità che genera handle di oggetti, si tratta del luogo naturale per controllare la sicurezza, invocando l'srm (Security Reference Monitor, si veda il Paragrafo 21.3.5.7). Il gestore degli oggetti, richiamando l'srm, verifica se un processo ha il diritto di accedere a un oggetto quando tenta di aprirlo. Se il controllo dell'accesso ha esito positivo, i diritti risultanti (codificati come **maschera di accesso**) vengono memorizzati nella tabella degli handle. Pertanto, l'handle opaco rappresenta l'oggetto nel kernel e inoltre identifica l'accesso che è stato concesso all'oggetto. Questa importante ottimizzazione fa sì che ogni volta che un file viene scritto (cosa che potrebbe accadere centinaia di volte al secondo), i controlli di sicurezza vengono completamente ignorati, poiché l'handle è già codificato come un handle di "scrittura". Viceversa, se un handle è un handle di "lettura", i tentativi di scrittura sul file verrebbero immediatamente respinti, senza richiedere un controllo di sicurezza.

Il gestore degli oggetti applica inoltre delle quote, come la quantità massima di memoria che un processo può utilizzare, tenendo in conto della memoria occupata da tutti gli oggetti referenziati dal processo e rifiutando di allocare ulteriore memoria quando la quantità di memoria accumulata eccede la quota assegnata al processo. Poiché gli oggetti possono essere referenziati tramite handle dalla modalità utente e kernel e richiamati tramite i puntatori dalla modalità kernel, il gestore degli oggetti deve tenere traccia di due conteggi per ciascun oggetto: il numero di handle e il numero di puntatori. Il primo è il numero di handle che fanno riferimento all'oggetto in tutte le tabelle degli handle (inclusa quella di sistema). Il conteggio dei puntatori è la somma di tutti gli handle (che contano come puntatori) più tutti i puntatori utilizzati dai componenti in modalità kernel. Il conteggio viene incrementato ogni volta che il kernel o un driver abbiano bisogno di un nuovo puntatore e decrementato quando il componente abbia finito di utilizzare il puntatore. Lo scopo del conteggio di questi riferimenti è quello di garantire che un oggetto non venga liberato mentre ha ancora un riferimento, ma può ancora rilasciare alcuni dei suoi dati (come il nome e il descrittore di sicurezza) quando tutti gli handle sono chiusi (poiché i componenti in modalità kernel non hanno bisogno di queste informazioni).

Il gestore degli oggetti mantiene lo spazio dei nomi interno di Windows. A differenza di unix, che radica lo spazio dei nomi del sistema nel file system, Windows utilizza uno spazio di nomi del gestore degli oggetti che è visibile solo in memoria o tramite strumenti specializzati come il debugger. Invece che da una directory del file system, la gerarchia è gestita da un tipo speciale di oggetto chiamato oggetto directory che contiene un bucket hash di altri oggetti (inclusi altri oggetti directory). Si noti che alcuni oggetti non hanno nomi (come i thread), e anche per gli altri oggetti il possesso o meno di un nome è deciso da chi li ha creati. Per esempio, un processo dà un nome a un mutex solo se vuole che altri processi lo trovino, lo acquisiscono o si informino sul suo stato. Poiché processi e thread vengono creati senza nomi, essi vengono referenziati tramite un identificatore numerico separato, per esempio un id di processo (pid) o di thread (tid). Il gestore degli oggetti supporta anche collegamenti simbolici nello spazio dei nomi. Per esempio, le lettere di unità dos vengono implementate utilizzando collegamenti simbolici; \Global??\C: è un collegamento simbolico all'oggetto dispositivo \Device\HarddiskVolumeN che rappresenta un volume del file system montato nella directory \Device.

Ogni oggetto, come accennato in precedenza, è un'istanza di un **tipo di oggetto**. Il tipo di oggetto specifica come devono essere allocate le istanze, come devono essere definiti i campi di dati e come deve essere implementato il set standard di funzioni virtuali utilizzate per tutti gli oggetti. Le funzioni standard implementano operazioni come la mappatura dei nomi agli oggetti, la chiusura, l'eliminazione e l'applicazione di controlli di sicurezza. Le funzioni specifiche per un particolare tipo di oggetto sono implementate dai servizi di sistema progettati per funzionare su quel particolare tipo di oggetto, non da metodi specificati nel tipo di oggetto.

La funzione `parse()` è la più interessante tra le funzioni standard e consente all'implementazione di un oggetto di sovrscrivere il comportamento di denominazione predefinito del gestore degli oggetti (che è quello di utilizzare le directory degli oggetti virtuali). Questa capacità è utile per oggetti che hanno il proprio spazio dei nomi interno, specialmente quando è necessario mantenere lo spazio dei nomi attraverso i successivi boot di sistema. Il gestore di i/o (per gli oggetti file) e il gestore della configurazione (per gli oggetti chiave del registro di sistema) sono gli utenti più importanti delle funzioni di parse.

Tornando al nostro esempio di assegnazione dei nomi in Windows, gli oggetti dispositivo utilizzati per rappresentare i volumi del file system forniscono una funzione di parse per permettere a un nome come \Global??\C:\foo\bar.doc di essere interpretato come il file \foo\bar.doc del volume rappresentato dall'oggetto dispositivo HarddiskVolume2. Possiamo osservare come l'assegnazione dei nomi, le funzioni di parse, gli oggetti e gli handle lavorino insieme illustrando i passi che permettono di aprire un file in Windows.

1. Un'applicazione richiede l'apertura di un file denominato C:\foo\bar.doc.
2. Il gestore degli oggetti trova l'oggetto dispositivo HarddiskVolume2, cerca la procedura di parse `IopParseDevice` dal tipo di oggetto e la richiama con il nome del file relativo alla radice del file system.
3. `IopParseDevice()` cerca il file system che contiene il volume HarddiskVolume2 e poi chiama il file system, che cerca come accedere a \foo\bar.doc sul volume, eseguendo la propria analisi interna della directory foo per trovare il file bar.doc. Il file system alloca quindi un oggetto file e lo restituisce alla routine di parse del gestore di i/o.

Quando il file system ha terminato, `IopParseDevice()` assegna una voce nella tabella degli handle per l'oggetto file per il processo corrente e restituisce l'handle all'applicazione.

Se il file non può essere aperto correttamente, `IopParseDevice()` cancella l'oggetto file che ha allocato e restituisce all'applicazione una segnalazione di errore.

21.3.5.2 Gestore della memoria

Il componente dell'executive che gestisce lo spazio degli indirizzi virtuali, l'allocazione della memoria fisica e la paginazione è il **gestore della memoria** (*memory manager, mm*). Il gestore della mm è concepito assumendo che l'architettura sottostante sia in grado di associare indirizzi virtuali a indirizzi fisici, possieda un meccanismo di paginazione, realizzi trasparentemente la coerenza della cache nei sistemi multiprocessore e permetta l'associazione di più elementi della tabella delle pagine alla stessa pagina fisica.

Il gestore mm di Windows adotta uno schema di gestione basato su pagine delle dimensioni supportate dall'hardware (4kb, 2mb e 1gb). Le pagine di dati allocati a un processo, ma non residenti nella memoria fisica, sono memorizzate nel file di paginazione in un disco o mappate direttamente su un file ordinario di un file system locale o remoto. Le pagine possono anche essere marcate come *zero-fill-on-demand*, il che ha l'effetto di inizializzarle a zero prima dell'allocazione, cancellandone i contenuti precedenti.

Sui processori a 32 bit, come ia-32 e arm, i processi hanno 4 gb di memoria virtuale a testa. Per default, i 2 gb superiori sono quasi identici per tutti i processi, e servono a Windows per accedere in modalità kernel alle strutture dati e al codice del sistema operativo. Nel caso delle architetture a 64 bit come amd64 Windows fornisce per ogni processo 256 tb di spazio di indirizzamento virtuale, suddiviso in due regioni da 128 tb per la modalità utente e la modalità kernel (queste restrizioni sono basate su limitazioni hardware).

che verranno presto rimosse: Intel ha annunciato che i suoi futuri processori supporteranno fino a 128 pb di spazio di indirizzamento virtuale, dei 16 eb teoricamente disponibili).

La disponibilità del codice del kernel nello spazio di indirizzamento di ogni processo è importante e si trova comunemente anche in molti altri sistemi operativi. Generalmente, viene utilizzata la memoria virtuale per mappare il codice del kernel nello spazio di indirizzamento di ciascun processo in modo che, quando viene eseguita una chiamata di sistema o viene ricevuto un interrupt, il cambio di contesto per consentire al core corrente di eseguire quel codice è più leggero di quanto lo sarebbe senza questa mappatura. In particolare, non è necessario salvare e ripristinare i registri di gestione della memoria e la cache non viene invalidata. Il risultato finale è uno spostamento molto più rapido tra codice utente e codice del kernel, rispetto alle architetture più vecchie che mantengono la memoria del kernel separata e non disponibile all'interno dello spazio di indirizzamento del processo.

Per assegnare memoria virtuale il gestore mm procede in due passi: innanzitutto *riserva* una o più pagine di indirizzi virtuali dello spazio d'indirizzi virtuali del processo; quindi *impegna* l'assegnazione dello spazio per la memoria virtuale (spazio in memoria fisica o nel file di paginazione). Il sistema operativo Windows può limitare lo spazio del file di paginazione impiegato da un processo imponendo una quota massima d'assegnazione; un processo può rilasciare parti di memoria non più in uso a beneficio di altri processi. Le api che riservano gli indirizzi virtuali e impegnano la memoria virtuale ricevono come parametro l'handle di un oggetto processo. In questo modo, un dato processo è in grado di controllare la memoria virtuale di un altro processo.

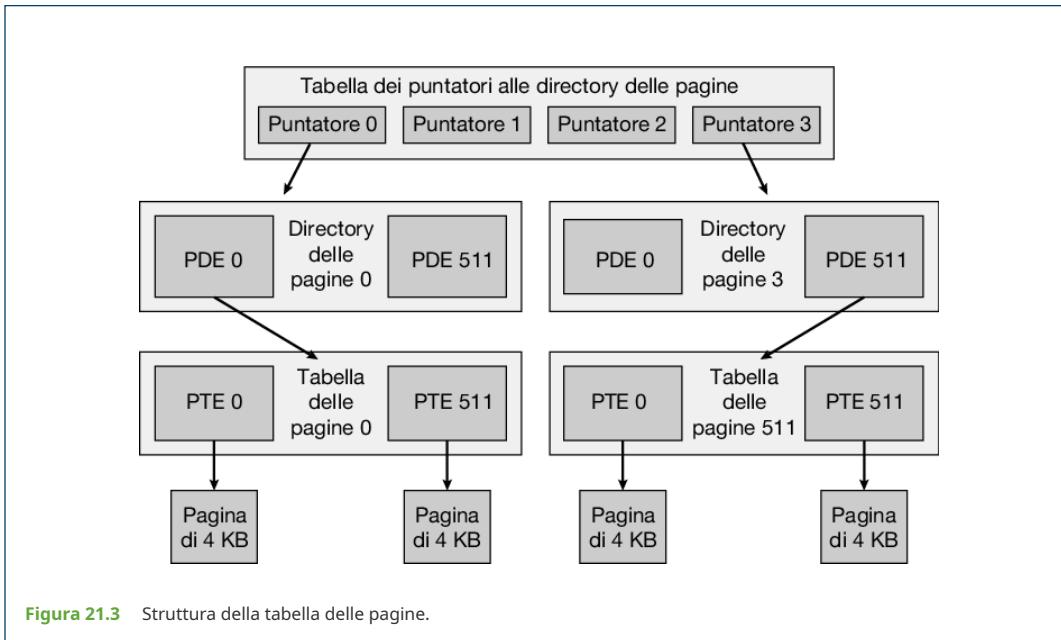
Windows implementa la memoria condivisa tramite gli **oggetti sezione**. Un processo, dopo aver ottenuto un handle dell'oggetto sezione, mappa la porzione di memoria della sezione su un intervallo di indirizzi, chiamato **vista** (*view*). Un processo può stabilire una vista dell'intera sezione o solo della parte di sezione di cui ha bisogno. Windows consente alle sezioni di essere mappate non solo nel processo corrente, ma in un qualunque processo per cui il chiamante possiede un handle.

Le sezioni sono utilizzabili in molti modi. Una sezione può utilizzare come memoria ausiliaria il file di paginazione di sistema o un file ordinario (un file mappato in memoria). Una sezione può essere *basata* (*based*), nel senso che appare allo stesso indirizzo virtuale a ogni processo che vi accede. Le sezioni possono anche rappresentare la memoria fisica, consentendo così a un processo a 32 bit di accedere a una maggior quantità di memoria fisica di quanta ne possa essere contenuta nel suo spazio di indirizzamento virtuale. Infine la protezione delle pagine della sezione si può impostare per la sola lettura, lettura e scrittura, sola esecuzione, non accessibile, e copiatura su scrittura.

Le ultime due impostazioni di protezione richiedono alcune spiegazioni.

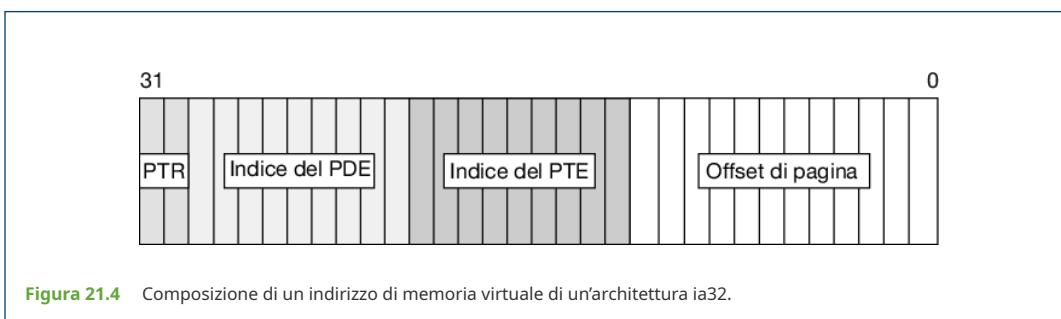
- Una *pagina non accessibile* solleva un'eccezione qualora si tenti di accedervi; l'eccezione si può usare, fra l'altro, per controllare se un programma difettoso esegue iterazioni che vanno oltre la dimensione di un array o semplicemente per rilevare se un programma tenta di accedere a indirizzi virtuali non presenti in memoria. Gli stack in modalità utente e kernel utilizzano le pagine non accessibili come **sentinelle** (*guard page*) per rilevare eventuali stack overflow. Un altro possibile uso è il **controllo del superamento delle dimensioni del buffer** (*buffer overrun*) nello heap. Sia l'allocatore della memoria utente sia l'allocatore del kernel usato dal verificatore del dispositivo possono essere configurati per mappare ogni allocazione richiesta fino alla fine di una pagina seguita da una pagina non accessibile; ciò permette di rilevare errori di programmazione che causano accessi oltre il limite dell'allocazione.
- Il meccanismo di copiatura su scrittura permette al gestore della mm di risparmiare memoria: quando due processi vogliono copie indipendenti dei dati dello stesso oggetto, il gestore pone solo una copia condivisa nella memoria fisica e attribuisce a quella sezione la proprietà di copiatura su scrittura; se uno dei processi tenta di scrivere dati in una delle pagine caratterizzate da tale proprietà, il gestore fornisce al processo una copia privata della pagina che può poi essere modificata.

La traduzione degli indirizzi virtuali in molti processori moderni sfrutta una tabella delle pagine a più livelli. Nel caso di processori ia-32 e amd64, ogni processo ha una directory delle pagine contenente 512 elementi della **directory delle pagine** (*page-directory entries*, pde), da 8 byte l'uno; ognuno di loro punta a una tabella delle pagine, che a sua volta contiene 512 elementi della **tabella delle pagine** (*page-table entries*, pte) di 8 byte, che infine puntano a **frame** da 4 kb nella memoria fisica. Per una serie di motivi, l'hardware richiede che le directory delle pagine o le tabelle dei pte a ogni livello di una tabella delle pagine multilivello occupino una sola pagina. In tal modo, il numero di pde o pte contenuti in una pagina determina quanti indirizzi virtuali sono tradotti da quella pagina. Per uno schema di questa struttura si veda la Figura 21.3.



La struttura fin qui descritta può essere utilizzata per rappresentare la traduzione di solo 1 gb di indirizzi virtuali. Nel caso di ia-32 è necessario un secondo livello di directory delle pagine contenente solo quattro voci, come mostrato nel diagramma. Su processori a 64 bit sono necessari più livelli. Per amd64 Windows utilizza in totale quattro livelli completi. La dimensione totale di tutte le pagine della tabella delle pagine necessarie per rappresentare pienamente anche solo uno spazio di indirizzamento virtuale a 32 bit per un processo è di 8 mb. Il gestore mm alloca pagine di pde e pte secondo le necessità e sposta le pagine della tabella su disco quando non sono utilizzate. Le pagine della tabella sono riportate in memoria quando vi si fa riferimento.

Descriviamo ora come gli indirizzi virtuali siano tradotti in indirizzi fisici su processori ia-32. Un valore di 2 bit può rappresentare i valori 0, 1, 2, 3, un valore di 9 bit può rappresentare valori nell'intervallo 0-511 e un valore di 12 bit valori da 0 a 4095. Un valore di 12 bit può pertanto selezionare qualsiasi byte all'interno di una pagina di 4 kb di memoria. Un valore di 9 bit può rappresentare uno qualsiasi dei 512 pde o pte in una directory delle pagine o in una pagina della tabella dei pte. Come mostrato nella Figura 21.4, la traduzione di un puntatore di indirizzo virtuale nell'indirizzo di un byte nella memoria fisica si effettua suddividendo il puntatore a 32 bit in quattro valori. A partire dai bit più significativi.



- Due bit vengono utilizzati per indicizzare i quattro pde al livello superiore della tabella delle pagine. Il pde selezionato conterrà il numero di pagina fisica per ciascuna delle quattro pagine della directory delle pagine che mappano 1 gb di spazio di indirizzi.
- Nove bit vengono utilizzati per selezionare un altro pde, questa volta da una directory delle pagine di secondo livello. Questo pde conterrà i numeri di pagina fisica di un massimo di 512 pagine della tabella dei pte.
- Nove bit vengono utilizzati per selezionare uno dei 512 pte dalla pagina selezionata della tabella dei pte. Il pte selezionato conterrà il numero di pagina fisica per il byte a cui stiamo accedendo.
- Dodici bit sono usati come offset all'interno della pagina. L'indirizzo fisico del byte a cui stiamo accedendo è costruito aggiungendo i 12 bit più bassi dell'indirizzo virtuale alla fine del numero di pagina fisica che abbiamo trovato nel pte selezionato.

Il numero di bit in un indirizzo fisico può essere diverso dal numero di bit in un indirizzo virtuale. Nell'architettura ia-32 originale, il pte e il pde erano strutture a 32 bit che facevano spazio a 20 bit di numero di pagina fisica: la dimensione dell'indirizzo fisico e la dimensione degli indirizzi virtuali erano quindi le stesse e tali sistemi potevano indirizzare solo 4 gb di memoria fisica. Più tardi, la ia-32 fu estesa con l'introduzione dei pte a 64 bit utilizzati ancora oggi e l'hardware iniziò a supportare numeri di pagina fisica a 24 bit. Questi sistemi sono in grado di gestire 64 gb di memoria ed erano utilizzati su sistemi server. Oggi tutti i server Windows si basano su amd64 o ia64 e supportano indirizzi fisici molto grandi, molto più grandi di quanto sia possibile utilizzare: ricordate il fatto che una volta 4 gb di memoria fisica sembravano ottimisticamente troppi.

Per migliorare le prestazioni il gestore mm mappa la directory delle pagine e le pagine della tabella dei pte nella stessa regione contigua di indirizzi virtuali in ogni processo. Questa auto-mappa (*self-map*) permette al gestore di utilizzare lo stesso puntatore per accedere al pde o al pte corrente, corrispondente a un particolare indirizzo virtuale, indipendentemente dal processo che è in esecuzione. In ia-32 l'auto-mappa utilizza una regione contigua da 8 mb dello spazio di indirizzamento virtuale del kernel, mentre in amd64 la mappa occupa 512 gb. Nonostante il significativo spazio di indirizzamento occupato, questa mappa non richiede alcuna pagina di memoria virtuale aggiuntiva; inoltre, consente alle pagine della tabella delle pagine di essere automaticamente spostate dentro e fuori dalla memoria fisica.

Nella creazione di un'auto-mappa, uno dei pde della directory delle pagine di livello superiore fa riferimento alla pagina stessa, formando un "ciclo" nella traduzione della tabella delle pagine. Quando non si percorre il ciclo si effettua un accesso alle pagine virtuali, quando il ciclo viene percorso una volta si accede alle pagine della tabella dei pte, quando il ciclo viene percorso due volte si accede alle pagine della directory delle pagine di livello più basso, e così via.

I livelli aggiuntivi delle directory delle pagine utilizzati per la memoria virtuale a 64 bit sono tradotti nello stesso modo, eccetto per il fatto che il puntatore all'indirizzo virtuale viene suddiviso in ancor più valori. Nel caso di amd64, Windows utilizza quattro livelli pieni, ognuno dei quali mappa 512 pagine, o $9 + 9 + 9 + 9 + 12 = 48$ bit di indirizzamento virtuale.

Per evitare i rallentamenti dovuti alla traduzione degli indirizzi virtuali tramite la ricerca dei pde e pte appropriati, i processori adottano uno speciale hardware detto **tlb** (*translation look-aside buffer*), contenente una memoria cache associativa per mappare pagine virtuali sui pte. Il tlb fa parte dell'**unità di gestione della memoria** (mmu) all'interno di ciascun processore. La mmu ha bisogno di percorrere la tabella delle pagine (le sue strutture dati) in memoria solo quando una traduzione è necessaria, perché manca nel tlb.

I pde e i pte non contengono solo i numeri di pagina fisica, ma hanno anche alcuni bit riservati per l'uso del sistema operativo e alcuni bit per controllare come l'hardware utilizzi la memoria, per esempio per controllare se la cache hardware debba essere utilizzata per ogni pagina. Inoltre le voci specificano il tipo di accesso consentito per le modalità kernel e utente.

Un pde può essere contrassegnato per segnalare che dovrebbe funzionare come pte piuttosto che come pde. Su ia-32 i primi 11 bit del puntatore di indirizzo virtuale selezionano un pde nei primi due livelli di traduzione. Se il pde selezionato è contrassegnato per fungere da pte i rimanenti 21 bit del puntatore sono usati come offset del byte. Ciò porta a una dimensione di pagina di 2 mb. Mischiare e far corrispondere pagine di 4 kb e 2 mb all'interno della tabella delle pagine è facile per il sistema operativo e può migliorare significativamente le prestazioni di alcuni programmi, riducendo la frequenza con cui la mmu ha bisogno di ricaricare le voci nel tlb, visto che un pde di 2 mb sostituisce 512 pte di 4 kb. Le più moderne architetture amd64 supportano anche pagine da 1gb, gestite in maniera simile.

Gestire la memoria fisica in modo che le pagine di 2 mb siano disponibili al bisogno è tuttavia difficile, in quanto le pagine possono continuamente essere suddivise in pagine di 4 kb, provocando frammentazione esterna della memoria. Inoltre, le pagine grandi possono causare una frammentazione interna piuttosto significativa. A causa di questi problemi, sono in genere esclusivamente lo stesso Windows e le applicazioni per server di grandi dimensioni a utilizzare pagine grandi per migliorare le prestazioni del tlb. Essi sono più adatti a farlo, perché il sistema operativo e le applicazioni server iniziano l'esecuzione all'avvio del sistema, prima che la memoria sia diventata frammentata.

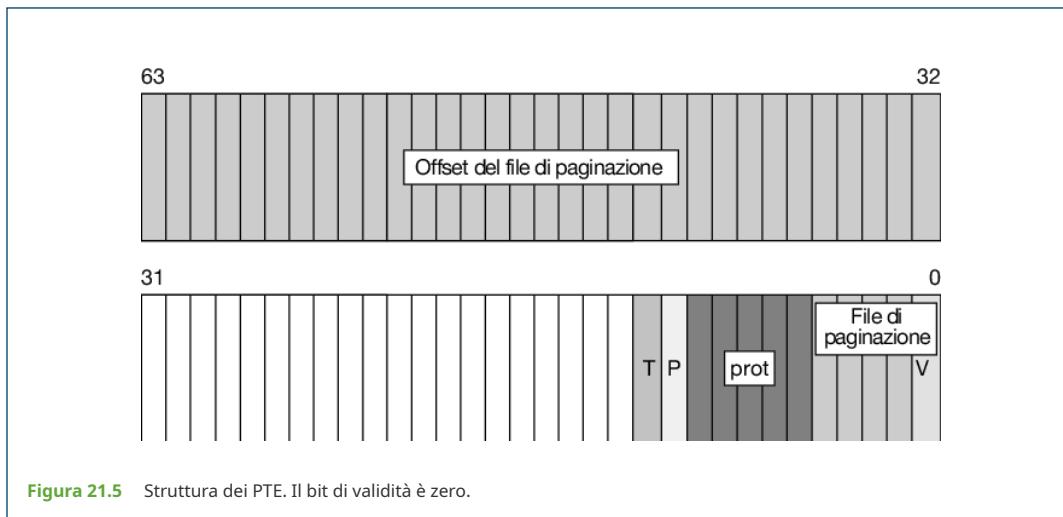
Windows gestisce la memoria fisica associando ciascuna pagina fisica a uno dei sette stati seguenti: libera, azzerata, modificata, in attesa, guasta, in transizione o valida.

- Una **pagina libera** non ha un contenuto particolare.
- Una **pagina azzerata** è libera, i suoi contenuti sono stati inizializzati a zero ed è disponibile per soddisfare richieste di pagine *zero-on-demand*.
- Una **pagina modificata** è stata modificata (tramite scrittura) da un processo e deve essere trasferita su disco prima di essere allocata a un altro processo.
- Una **pagina in attesa** è una copia d'informazioni già presenti su disco. Si può trattare di una pagina non ancora modificata, di una pagina già scritta su disco o di una pagina caricata anticipatamente in memoria perché ci si aspetta di utilizzarla al più presto.
- Una **pagina guasta** è inutilizzabile a causa di un errore hardware.
- Una **pagina in transizione** è in corso di trasferimento dal disco a un frame allocato nella memoria fisica.
- Una **pagina valida** è parte del working set di uno o più processi ed è contenuta all'interno delle tabelle delle pagine di questi processi.

Mentre le pagine valide sono contenute nelle tabelle delle pagine dei processi, le pagine in altri stati sono mantenute in liste distinte in base al tipo di stato. Le liste sono costruite associando le voci corrispondenti nel database pfn (page frame number, *numero di frame di pagina*), che contiene una voce per ogni pagina di memoria fisica. Le voci pfn includono anche informazioni come il numero di riferimenti, i lock e le informazioni numa. Si noti che il database pfn rappresenta pagine di memoria fisica, mentre il pte rappresenta pagine di memoria virtuale.

Se il bit di validità di un pte è a zero, l'hardware ignora gli altri bit e il gestore della mm può definirne il formato per il proprio uso. Le pagine non valide si possono trovare in un certo numero di stati rappresentati da bit nel pte. Le pagine dei file di paginazione mai caricati in seguito a un fault sono marcate *zero-on-demand*; le pagine mappate tramite oggetti sezione codificano un puntatore all'oggetto in questione; le pagine scritte sul file di paginazione contengono informazioni sufficienti per reperire la pagina sul disco, e così via. La struttura dei pte è mostrata nella Figura 21.5. Per questo tipo di pte i bit T, P e V sono tutti a zero. Il pte contiene 5 bit

dedicati alla protezione della pagina, 32 bit per l'offset all'interno del file di paginazione e 4 bit per selezionare il file di paginazione. Ci sono anche 20 bit riservati per ulteriori operazioni.



Windows utilizza una politica di sostituzione lru sui working set per prelevare le pagine dai processi in maniera appropriata. Quando si avvia un processo, gli viene assegnata una dimensione minima predefinita del working set. Il working set di ogni processo può crescere fino a quando la quantità residua di memoria fisica comincia a scarseggiare; a quel punto il gestore mm inizia a tracciare l'età delle pagine in ogni working set. Infine, se la memoria disponibile è quasi esaurita, il gestore mm taglia il working set rimuovendo le pagine più vecchie.

L'età di una pagina non dipende da quanto tempo è stata in memoria, ma dall'ultimo riferimento alla pagina, e viene determinata scandendo periodicamente il working set di ogni processo e incrementando l'età delle pagine che dall'ultima passata non sono state evidenziate nel pte come referenziate. Quando diventa necessario tagliare il working set, il gestore mm utilizza euristiche per decidere quanto tagliare da ogni processo e quindi rimuove le pagine a partire dalle meno recenti.

È possibile che il working set di un processo venga tagliato anche quando resta disponibile molta memoria. Ciò avviene se è stato impostato un limite rigido sulla quantità di memoria fisica che il processo può usare. In Windows 7 e versioni successive il gestore mm è in grado di tagliare anche i processi che stanno crescendo rapidamente, anche se la memoria è abbondante. Questa politica migliora significativamente la capacità di risposta del sistema sugli altri processi.

Windows tiene traccia dei working set non solo per i processi in modalità utente, ma anche per il processo di sistema, che comprende tutte le strutture dati paginabili e il codice che viene eseguito in modalità kernel. Windows 7 crea working set supplementari per il processo di sistema e li associa a particolari categorie di memoria del kernel. La cache dei file, l'heap del kernel e il codice del kernel hanno in Windows 7 i loro working set. La presenza di working set distinti consente al gestore mm di utilizzare diverse politiche per tagliare le diverse categorie di memoria del kernel.

A seguito di un errore per page fault, il gestore mm non carica solo la pagina immediatamente necessaria, perché vi è evidenza a supporto della tesi che i riferimenti in memoria seguano un **principio di località**: se una pagina è usata è probabile che lo siano anche le pagine adiacenti nel prossimo futuro. (Si pensi alle iterazioni per scandire un array, o al reperimento sequenziale delle istruzioni che costituiscono il codice di un thread). Per questi motivi, il gestore carica in memoria la pagina richiesta, insieme però a un certo numero di pagine adiacenti; ciò tende a ridurre il numero totale di page fault. Inoltre, le operazioni di lettura sono accorpate per migliorare le prestazioni.

Oltre a occuparsi della memoria impegnata, il gestore mm si prende anche cura della memoria riservata da ogni processo, ossia del suo spazio degli indirizzi virtuali. Ciascun processo ha un albero associato che specifica la gamma di indirizzi virtuali usati e la loro finalità. Su questa base, il gestore mm può caricare le pagine mancanti man mano che servono: se il pte non è inizializzato, il gestore cerca l'indirizzo interessato all'interno dell'albero dei **descrittori di indirizzi virtuali** (*virtual address descriptor*, vad), e usa l'informazione così reperita per creare il pte mancante e individuare la pagina. In qualche caso potrebbe non esistere la pagina della tabella dei pte, quindi il gestore dovrà allocarla e inizializzarla in maniera trasparente. In altri casi, la pagina potrebbe essere condivisa come parte di un oggetto sezione e il vad conterrà un puntatore a quell'oggetto sezione. L'oggetto sezione contiene informazioni su come trovare la pagina virtuale condivisa in modo che il pte possa essere inizializzato per puntare direttamente a questa.

A partire da Vista, Windows mm include un componente chiamato SuperFetch, che combina un servizio in modalità utente con un codice specializzato in modalità kernel, che include un filtro del file system, per monitorare tutte le operazioni di paginazione sul sistema. Ogni secondo, il servizio verifica tutte queste operazioni e utilizza diversi agenti per monitorare gli avvii di applicazioni, i cambi di utente rapidi, le operazioni di standby/sospensione/ibernazione e altro come mezzo per comprendere gli schemi di utilizzo del sistema. Con queste informazioni costruisce un modello statistico, utilizzando le catene di Markov, le cui applicazioni è probabile che l'utente lancerà e in che momento, in combinazione con quali altre applicazioni, e quali porzioni di queste applicazioni verranno utilizzate. Per esempio, SuperFetch può arrivare a comprendere che l'utente avvia Microsoft Outlook al mattino per lo più per leggere

e-mail, ma scriva e-mail più tardi, dopo pranzo. Può anche capire che una volta che Outlook è in background, è probabile che venga avviato Visual Studio, che l'editor di testo sarà molto richiesto, che il compilatore sarà richiesto un po' meno frequentemente, il linker ancora meno frequentemente e il codice della documentazione quasi mai. Con questi dati, SuperFetch popola preventivamente la lista di standby, effettuando letture di i/o a bassa priorità dalla memoria secondaria in momenti di inattività per caricare ciò che pensa che l'utente possa eseguire successivamente (o un altro utente, se sa che è probabile un cambio utente rapido). Inoltre, utilizzando le otto liste di attesa priorizzate da Windows, ciascuna di queste pagine precaricate può essere memorizzata nella cache a un livello che corrisponde alla probabilità statistica che sarà necessaria. In questo modo, le pagine di cui è improbabile una richiesta possono essere eliminate a basso costo e rapidamente in seguito a un imprevisto bisogno di memoria fisica, mentre le pagine che potrebbero essere presto richieste possono essere mantenute più a lungo. In effetti, SuperFetch può persino forzare il sistema a effettuare il trimming dei working set di altri processi prima di spostare tali pagine dalla memoria.

Il monitoraggio di SuperFetch crea un notevole sovraccarico del sistema. Sulle unità meccaniche, che hanno tempi di ricerca nell'ordine dei millisecondi, questo costo è bilanciato dal vantaggio di evitare latenze e ritardi di diversi secondi nei tempi di avvio dell'applicazione. Sui sistemi server, tuttavia, tale monitoraggio non è vantaggioso, dati i carichi di lavoro casuali e il fatto che il throughput è più importante della latenza. Inoltre, i miglioramenti combinati della latenza e della larghezza di banda su sistemi con memoria non volatile efficiente e rapida, come gli ssd, rendono il monitoraggio meno vantaggioso per tali sistemi. In tali situazioni, SuperFetch si disabilita da solo, liberando alcuni cicli di cpu.

Windows 10 offre un ulteriore grande miglioramento al mm introducendo un componente chiamato **gestore dell'archivio compresso**. Questo componente crea un **archivio compresso di pagine** nel working set del processo di compressione della memoria, che è un tipo di processo di sistema. Quando le pagine condivisibili si trovano nella lista di standby e la memoria disponibile è bassa (o vengono prese alcune decisioni dell'algoritmo interno), le pagine della lista vengono compresse anziché espulse. Lo stesso può accadere con pagine modificate selezionate per il trasferimento nella memoria secondaria, sia riducendo la pressione della memoria, magari evitando in primo luogo la scrittura, sia facendo sì che le pagine scritte vengano compresse, consumando così meno spazio per i file di pagina ed eseguendo meno i/o per il page out. Nei moderni e veloci sistemi multiprocessore, spesso dotati di algoritmi di compressione hardware, una ridotta penalizzazione della cpu è altamente preferibile al potenziale costo dell'i/o sulla memoria secondaria.

21.3.5.3 Gestore dei processi

Il gestore dei processi del sistema Windows fornisce i servizi necessari a creazione, eliminazione e uso dei thread e dei processi. Esso non possiede alcuna informazione sulle relazioni parentali o sulle gerarchie fra i processi; questi dettagli sono lasciati al particolare sottosistema d'ambiente relativo al processo. Il gestore dei processi non è coinvolto neppure nello scheduling dei processi, fuorché per determinare le priorità e le affinità dei processi e dei thread, quando essi sono creati. Lo scheduling dei thread ha luogo nel dispatcher del kernel.

Ciascun processo contiene uno o più thread. I processi medesimi possono essere raccolti in grosse unità, chiamate **oggetti job**; il ricorso a oggetti job permette di impostare limiti all'utilizzo della cpu e alla dimensione dei working set e di definire affinità per i processore che controllano più processi in una volta. Gli oggetti job sono usati per gestire le potenti macchine dei centri di elaborazione dati. In Windows xp e versioni successive, gli oggetti job sono stati estesi per fornire funzionalità relative alla sicurezza e alcune applicazioni di terze parti come Google Chrome hanno iniziato a utilizzare i job per questo scopo. In Windows 8, una massiccia modifica architettonica ha consentito ai job di influenzare lo scheduling attraverso un generico meccanismo di limitazione della cpu, e la limitazione/bilanciamento equo effettuati per sessione e per utente. In Windows 10, il supporto a questi meccanismi è stato esteso anche all'i/o di memoria secondaria e all'i/o di rete. Inoltre, Windows 8 consente agli oggetti job di essere annidati, creando gerarchie di limiti, rapporti e quote che il sistema doveva calcolare con precisione. Sono state inoltre fornite anche agli oggetti job ulteriori funzionalità di sicurezza e di gestione dell'alimentazione.

Di conseguenza, tutte le applicazioni Windows Store e tutti i processi di un'applicazione uwp vengono eseguiti nell'ambito di job. Il dam (Desktop Activity Moderator), introdotto in precedenza, implementa il supporto alla Sospensione Connnessa utilizzando i job. Infine, il supporto di Windows 10 per i Container Docker, una parte fondamentale delle sue offerte cloud, utilizza oggetti job, chiamati **silos**. I job sono così passati dall'essere una funzione esoterica di gestione delle risorse nei data center a un meccanismo di base del gestore dei processi per offrire diverse funzionalità.

A causa dell'architettura a strati di Windows e della presenza di sottosistemi di ambiente, la creazione dei processi è piuttosto complessa. Un esempio di creazione di processi nell'ambiente Win32 in Windows 10 è il seguente. Si noti che l'avvio delle applicazioni uwp "Modern" di Windows Store (che sono chiamate **applicazioni pacchettizzate**, o "AppX") è significativamente più complesso e coinvolge fattori al di fuori dello scopo di questa discussione.

1. Un'applicazione Win32 chiama `CreateProcess()`.
2. Viene eseguito un certo numero di conversioni di parametri e conversioni di comportamenti dal mondo Win32 al mondo nt.
3. `CreateProcess()` chiama quindi l'api `NtCreateUserProcess()` nel gestore dei processi dell'executive nt per creare effettivamente il processo e il thread iniziale.
4. Il gestore dei processi chiama il gestore degli oggetti per creare un oggetto processo e restituisce l'handle dell'oggetto a Win32. Chiama quindi il gestore della memoria per inizializzare lo spazio di indirizzamento del nuovo processo, la sua tabella degli handle e altre strutture dati chiave, come il blocco dell'ambiente di processo (pebl), che contiene i dati interni di gestione del processo.
5. Il gestore dei processi richiama di nuovo il gestore degli oggetti per creare un oggetto thread e restituisce l'handle a Win32. Chiama quindi il gestore della memoria per creare il blocco dell'ambiente thread (teb) e il dispatcher per inizializzare gli attributi di scheduling del thread, impostando il suo stato su inizializzazione.
6. Il gestore dei processi crea il contesto iniziale di avvio del thread (che punterà alla routine `main()` dell'applicazione), chiede allo scheduler di contrassegnare il thread come pronto, quindi lo sospende immediatamente mettendolo in uno stato di attesa.
7. Viene inviato un messaggio al sottosistema Win32 per notificare che il processo è in fase di creazione. Il sottosistema esegue del lavoro specifico di Win32 per inizializzare il processo, come il calcolo del livello di arresto e il disegno della clessidra animata o del cursore del mouse "a ciambella".
8. Tornando in `CreateProcess()`, all'interno del processo padre, viene richiamata l'api `ResumeThread()` per riattivare il thread iniziale del processo. Il controllo ritorna al genitore.

- Ora, all'interno del thread iniziale del nuovo processo, il link loader in modalità utente assume il controllo (all'interno di `ntdll.dll`, che viene automaticamente mappata in tutti i processi) e carica tutte le dipendenze di libreria (dll) dell'applicazione, crea il suo heap iniziale, imposta la gestione delle eccezioni e le opzioni di compatibilità delle applicazioni e infine chiama la funzione `main()` dell'applicazione.

Le api di Windows che manipolano la memoria virtuale e i thread, e che duplicano gli handle, accettano come parametro in ingresso un handle del processo, in modo che i sottosistemi possano eseguire operazioni per conto di un nuovo processo senza dover eseguire direttamente nel contesto del nuovo processo. Windows fornisce altresì una chiamata `fork()`, di derivazione unix. Un insieme di funzionalità, tra cui la **riflessione del processo**, che viene utilizzata dall'infrastruttura di segnalazione degli errori Windows (wer) durante i crash di un processo, così come l'implementazione dell'api `fork()` di Linux da parte del sottosistema Windows per Linux, dipendono da questa funzionalità.

Il supporto del debugger da parte del gestore dei processi include la possibilità di sospendere e riavviare i thread, e di creare thread che iniziano l'esecuzione sospendendosi. Vi sono anche api del gestore dei processi in grado di leggere e impostare il contesto del registro di un thread e accedere alla memoria virtuale di un altro processo. I thread possono essere creati nel processo corrente, ma possono anche essere iniettati in un processo diverso. Il debugger fa uso di quest'ultimo meccanismo per eseguire codice all'interno di un processo in fase di debug. Sfortunatamente, la capacità di allocare, manipolare e iniettare sia la memoria sia i thread tra i processi è spesso sfruttata in maniera malevola da programmi maligni.

Nell'ambito dell'executive, i thread esistenti possono aggregarsi temporaneamente a un altro processo: questo metodo è usato dai thread di lavoro che devono operare nel contesto del processo da cui proviene una richiesta di lavoro. Per esempio, il gestore di mm potrebbe utilizzare un thread aggregato quando è necessario l'accesso al working set o alla tabella delle pagine di un processo e il gestore di i/o potrebbe utilizzarlo per aggiornare la variabile di stato in un processo per le operazioni di i/o asincrono.

21.3.5.4 Servizi per la computazione client-server

Come molti altri sistemi operativi moderni, Windows utilizza estensivamente un modello client-server, principalmente come meccanismo di stratificazione, che consente di inserire funzionalità comuni in un "servizio" (l'equivalente di un demone in termini unix), oltre a distinguere i codice di analisi di contenuti (come il lettore pdf o il browser Web) dal codice capace di azioni sul sistema (come la capacità del browser web di salvare un file sulla memoria secondaria o la capacità del lettore pdf di stampare un documento). Per esempio, in un moderno sistema operativo Windows 10, l'apertura del sito web del New York Times con il browser Microsoft Edge comporterà probabilmente da 12 a 16 processi diversi in un'organizzazione complessa di "broker", "renderer/parser", "jitter", servizi e client.

Il "server" più semplice su un computer Windows è il sottosistema di ambiente Win32, che implementa la personalità del sistema operativo dell'api Win32 ereditata dai tempi di Windows 95/98. Anche molti altri servizi, come l'autenticazione dell'utente, le funzionalità di rete, lo spooling della stampante, i servizi web, i file system di rete e il plug-and-play, vengono implementati utilizzando questo modello. Per moderare la memoria necessaria, molti servizi sono spesso accorpati all'interno di pochi processi che eseguono il programma `svchost.exe`. Ogni servizio viene caricato come una libreria dinamica (dll) che realizza il servizio sfruttando le potenzialità dei pool di thread in modalità utente per condividere thread e ricevere messaggi (si veda il Paragrafo 21.3.5.3). Sfortunatamente, questo raggruppamento ha originariamente portato a una scarsa esperienza utente nella risoluzione dei problemi e nel debug dell'uso della cpu e dei memory leak, e ha indebolito la sicurezza generale di ciascun servizio. Pertanto, nelle versioni recenti di Windows 10, se il sistema ha più di 2 gb di ram, ogni servizio dll viene eseguito nel proprio processo `svchost.exe`.

In Windows, il paradigma consigliato per implementare l'elaborazione client-server è utilizzare le rpc per comunicare le richieste, a causa della loro sicurezza intrinseca, dei servizi di serializzazione e delle funzionalità di estensibilità. L'api Win32 supporta lo standard Microsoft del protocollo dce-rpc, denominato ms-rpc, come descritto nel Paragrafo 21.6.2.7.

rpc utilizza diversi meccanismi di trasporto (per esempio, Named Pipes e tcp/ip) che possono essere utilizzati per implementare rpc tra sistemi diversi. Quando una rpc viene eseguita tra un client e un server sullo stesso sistema locale, per il trasporto può essere utilizzata la chiamata di procedura locale avanzata (alpc). Inoltre, poiché rpc è pesante e ha più dipendenze a livello di sistema (incluso il sottosistema d'ambiente Win32 stesso), molti servizi nativi di Windows, così come il kernel, utilizzano direttamente l'alpc, che non è disponibile (né adatta) per i programmati di terze parti.

Una alpc è un meccanismo per lo scambio dei messaggi simile alle socket di dominio unix e agli ipc Mach. Il processo server pubblica un oggetto globalmente visibile che rappresenta una porta di connessione: quando un client necessita di un servizio dal server, apre un handle per l'oggetto porta di connessione del server e trasmette una richiesta di connessione alla porta in questione. Se il server accetta la connessione, l'alpc crea una coppia di oggetti porta di comunicazione, fornendo l'api di connessione del client con il relativo handle alla coppia e poi fornendo l'api di accettazione del server con l'altro handle alla coppia.

A questo punto, i messaggi possono essere inviati attraverso le porte di comunicazione come datagrammi, che si comportano come udp e non richiedono risposte, oppure come richieste, che devono ricevere una risposta. Il client e il server possono quindi utilizzare la messaggistica sincrona, in cui vi è sempre una parte bloccata (in attesa di una richiesta o in attesa di una risposta) o messaggistica asincrona, in cui il meccanismo del pool di thread può essere utilizzato per eseguire attività ogni volta che una richiesta o una risposta viene ricevuta, senza che un thread si blocchi per un messaggio. Per i server che si trovano in modalità kernel, le porte di comunicazione supportano anche un meccanismo di callback, che consente un passaggio immediato al lato del kernel (kt) del thread in modalità utente (ut), eseguendo immediatamente la routine del gestore del server.

Quando viene inviato un messaggio alpc, è possibile scegliere una delle due seguenti tecniche di scambio di messaggi.

- La prima tecnica è adatta a messaggi brevi o di lunghezza media (fino a 64 kb): in questo caso, si usa la coda dei messaggi della porta come mezzo di memorizzazione intermedio, e si copiano i messaggi da un processo, al kernel, all'altro processo. Lo svantaggio di questa tecnica è il doppio buffering, oltre al fatto che i messaggi rimangono nella memoria del kernel fino a quando il destinatario previsto li consuma. Se il destinatario è molto conteso o al momento non disponibile, ciò potrebbe causare il blocco di megabyte di memoria kernel.
- La seconda tecnica è adatta a messaggi più lunghi: in questo caso, si crea un oggetto sezione di memoria condivisa per la porta: i messaggi trasmessi attraverso la coda dei messaggi della porta contengono un "attributo del messaggio", chiamato **attributo di visualizzazione dati**, che fa riferimento all'oggetto sezione. Il lato ricevente "espone" questo attributo, determinando una

mappatura dell'indirizzo virtuale dell'oggetto sezione e una condivisione della memoria fisica. Ciò evita la necessità di copiare messaggi di grandi dimensioni o di inserirli in un buffer nella memoria in modalità kernel. Il mittente inserisce i dati nella sezione condivisa e il destinatario li vede direttamente, non appena consuma un messaggio.

Esistono molti altri modi possibili per implementare la comunicazione client-server, per esempio utilizzando mailslot, pipe, socket, oggetti di sezione associati a eventi, messaggi di finestra e altro. Ognuno ha i suoi usi, vantaggi e svantaggi. rpc e alpc rimangono tuttavia i meccanismi più completi, protetti, sicuri e ricchi di funzionalità per tali comunicazioni, e sono i meccanismi utilizzati dalla maggior parte dei processi e dei servizi Windows.

21.3.5.5 Gestore dell'i/o

Il **gestore dell'i/o** è responsabile di tutti i driver di periferica del sistema, nonché dell'implementazione e della definizione del modello di comunicazione che consente ai driver di comunicare tra loro, con il kernel e con client e consumatori in modalità utente. Inoltre, come nei sistemi operativi basati su unix, l'i/o riguarda sempre un **oggetto file**, anche se il dispositivo non è un file system. Il gestore dell'i/o in Windows consente ai driver di periferica di essere "filtrati" da altri driver, creando uno **stack di dispositivi** attraverso il quale scorre l'i/o e che può essere utilizzato per modificare, estendere o migliorare la richiesta originale. Pertanto, il gestore dell'i/o tiene sempre traccia dei driver di periferica e dei driver filtro che sono stati caricati. Il gestore dell'i/o ha un supporto speciale per i driver del file system a causa della loro importanza, e implementa le interfacce per caricare e gestire i file system. Funziona insieme al mm per fornire i/o di file mappati in memoria e controlla il gestore di cache di Windows, che gestisce la cache per l'intero sistema di i/o. Il gestore di i/o fornisce numerosi modelli di completamento asincrono dell'i/o, compresi l'impostazione di eventi, l'aggiornamento di una variabile di stato nel processo chiamante, l'invio di apc al thread che ha avviato l'operazione, e le porte di completamento di i/o, che permettono a un singolo thread di gestire il completamento dell'i/o di molti thread. Gestisce inoltre i buffer per le richieste di i/o.

I driver dei dispositivi sono organizzati, per ciascun dispositivo, in una lista detta **stack di i/o del dispositivo**. Un driver viene rappresentato nel sistema come **oggetto driver**. Visto che un singolo driver può operare su più dispositivi, i driver vengono rappresentati sullo stack dell'i/o mediante un oggetto dispositivo che contiene un link all'oggetto driver. Inoltre, i driver non hardware possono utilizzare gli oggetti dispositivo come modo per esibire interfacce diverse. Per esempio, ci sono oggetti dispositivo tcp6, udp6, udp, tcp, Rawlp e Rawlp6 di proprietà dell'oggetto driver tcp/ip, anche se questi non rappresentano dispositivi fisici. Allo stesso modo, ogni volume nella memoria secondaria costituisce il proprio oggetto dispositivo, di proprietà dell'oggetto driver del gestore di volumi.

Quando un handle viene aperto su un oggetto dispositivo, il gestore di i/o crea sempre un oggetto file e restituisce un handle del file anziché un handle di dispositivo. Il gestore di i/o converte le richieste ricevute (per esempio, di creazione, lettura, scrittura) in una forma standard, detta **pacchetto di richiesta di i/o** (*i/o request packet*, irp), che inoltra al primo driver nello stack affinché sia elaborato. Dopo che un driver ha elaborato l'irp, richiama il gestore di i/o per inoltrare il pacchetto al driver successivo nello stack o, se l'elaborazione è terminata, per completare l'operazione sull'irp.

Il completamento può avvenire in un contesto differente dalla richiesta originale di i/o. Per esempio, se un driver sta eseguendo la propria parte di un'operazione di i/o ed è obbligato a bloccarsi per lungo tempo, può accadere l'irp a un thread di lavoro per proseguire l'elaborazione nel contesto del sistema. Al thread originale il driver restituisce un indicatore di stato per comunicare che la richiesta di i/o è in corso, in modo che possa continuare l'esecuzione in parallelo con l'operazione di i/o. I pacchetti irp possono anche essere elaborati da procedure di servizio delle interruzioni; l'elaborazione può terminare in un contesto di processo arbitrario. Poiché una qualche elaborazione finale potrebbe essere necessaria nel contesto che ha iniziato l'i/o, il gestore di i/o usa una apc per eseguire l'elaborazione finale, a completamento dell'i/o, nel contesto del thread che ha dato l'avvio all'operazione.

Questo modello a stack è molto flessibile. Quando si costruisce uno stack, vari driver hanno l'opportunità di inserirsi nello stack come **driver filtro**. I driver filtro sono in grado di esaminare e potenzialmente modificare ciascuna operazione di i/o. L'istantanea, o snapshot, del volume (**copie shadow**) e la crittografia del disco (**BitLocker**) sono due esempi built-in di funzionalità implementate utilizzando i driver filtro che, nello stack, vengono eseguite sopra il driver di gestione dei volumi. I driver filtro del file system vengono eseguiti al di sopra del file system e sono stati utilizzati per implementare funzionalità come la gestione della memoria gerarchica, l'istanziazione dei file per l'avvio remoto e la conversione dinamica del formato. Anche terze parti utilizzano i driver filtro del file system per implementare strumenti anti-malware. A causa dell'elevato numero di filtri del file system, Windows Server 2003 e le versioni successive includono ora un componente **gestore filtri**, che funge da unico filtro del file system e che carica i **minifiltri** (minifilter) ordinati per altitudini specifiche (priorità relative). Questo modello consente ai filtri di memorizzare in modo trasparente i dati e le query ripetute senza dover conoscere le rispettive richieste, oltre a fornire un ordinamento del carico più rigoroso.

I driver dei dispositivi di Windows sono scritti in conformità alle specifiche del "modello di driver diswe4 Windows" (*Windows driver model*, wdm). Esso stabilisce i requisiti del driver del dispositivo, incluse le modalità di stratificazione dei driver filtro, la condivisione del codice comune per gestire l'alimentazione e le richieste *plug-and-play*, la costruzione della corretta logica di cancellazione, e così via.

A causa della ricchezza del modello, scrivere un driver wdm completo per ogni periferica aggiunta può rivelarsi un compito improbo. Fortunatamente ciò non è necessario, grazie al cosiddetto modello porta/miniporta. Ciascuna istanza di un dispositivo che rientri in una classe di dispositivi simili, quali i driver audio, i dispositivi sata e i controlleri Ethernet, condivide un driver comune per quella classe, chiamato **driver della porta**. Il driver della porta implementa le operazioni standard per la classe di appartenenza e richiama poi le procedure specifiche del dispositivo nel **driver della miniporta** di quel dispositivo, al fine di realizzare le proprie funzionalità peculiari. Il livello del collegamento fisico dello stack di rete viene implementato in questo modo, con il driver della porta *ndis.sys* che implementa gran parte della funzionalità di elaborazione di rete generica e chiama i driver della miniporta della rete per specifici comandi hardware relativi all'invio e alla ricezione di frame di rete (come Ethernet).

Allo stesso modo, il wdm include un modello di classe/miniclasse, in cui una certa classe di dispositivi può essere implementata in modo generico da un singolo driver di classe, con chiamate a una miniclasse per funzionalità specifiche dell'hardware. Per esempio, il driver del disco di Windows è un driver di classe, così come i driver per cd/dvd e unità a nastro, e i driver per la tastiera e il mouse. Questi tipi di dispositivi non necessitano di un miniclasse, ma il driver di classe della batteria, per esempio, richiede una miniclasse per ciascuno dei vari gruppi di continuità esterni (ups) venduti dai fornitori.

Anche con i modelli porta/miniporta e classe/miniclasse deve essere scritta una significativa quantità di codice che si interfaccia con il kernel. Inoltre, questo modello non si adatta all'hardware personalizzato o ai driver logici (non hardware). A partire da Windows 2000 Service Pack 4, i driver in modalità kernel possono essere scritti utilizzando il kmdf (Kernel-Mode Driver Framework), che fornisce un modello di programmazione semplificato per i driver wdm. Un'altra opzione è l'umdf (User-Mode Driver Framework), che consente ai driver di essere scritti in modalità utente attraverso un driver reflector nel kernel che inoltra le richieste attraverso lo stack i/o del kernel. Questi due framework costituiscono il modello di Windows Driver Foundation, che ha raggiunto la versione 2.1 in Windows 10 e contiene un'api che offre completa compatibilità tra kmdf e umdf, e che è disponibile open-source sulla piattaforma GitHub.

Poiché non è necessario che i driver funzionino in modalità kernel, ed è più facile sviluppare e distribuire i driver in modalità utente, l'umdf è fortemente raccomandato per i nuovi driver, anche perché rende il sistema più affidabile, dato che un errore in un driver in modalità utente non causa un crash del kernel di sistema.

21.3.5.6 Gestore della cache

In molti sistemi operativi, il caching viene eseguito dal sistema dei dispositivi a blocchi, solitamente a livello fisico/di blocco. Windows fornisce invece una funzione di caching centralizzata che opera a livello di file logico/virtuale. Il **gestore della cache** lavora a stretto contatto con mm per fornire servizi cache a tutti i componenti sotto il controllo del gestore di i/o. Ciò significa che la cache può operare su qualsiasi cosa, da file remoti condivisi in rete a file logici su un file system personalizzato. La dimensione della cache cambia dinamicamente in base alla quantità di memoria disponibile nel sistema e può crescere fino a 2 tb su un sistema a 64 bit.

Il gestore della cache mantiene un proprio working set piuttosto che condividere quello del processo di sistema, per consentire un trimming più efficace dei file memorizzati nella cache. Per costruire la cache, il gestore della cache mappa i file nella memoria del kernel e quindi utilizza interfacce speciali verso il gestore mm per aggiungere o tagliare pagine da questo working set privato, consentendo così di sfruttare le funzionalità aggiuntive di memorizzazione nella cache fornite dal gestore della memoria.

La cache è divisa in blocchi da 256 kb, ognuno dei quali può contenere una vista di un file, cioè una parte del file mappata in memoria. Ciascun blocco della cache è descritto da un **blocco di controllo dell'indirizzo virtuale** (vacb), che memorizza l'indirizzo virtuale e l'offset della vista nel file, così come il numero di processi che stanno usando la vista. I vacb risiedono in array mantenuti dal gestore della cache, e ci sono array nella cache per dati critici come per dati con priorità bassa, in modo da migliorare le prestazioni in situazioni in cui la memoria è fortemente contesa.

Quando il gestore dell'i/o riceve una richiesta di lettura di un file a livello utente invia un irp allo stack di i/o per il volume su cui risiede il file. Per i file contrassegnati come memorizzabili nella cache, il file system chiama il gestore della cache per cercare i dati richiesti nelle sue viste dei file nella cache. Il gestore della cache calcola quale elemento dell'array vacb di quel file corrisponda all'offset in byte codificato nella richiesta. O l'elemento in questione punta alla vista interessata nella cache, oppure è nullo; in tal caso, il gestore della cache alloca un blocco della cache, insieme all'elemento corrispondente dell'array vacb, e mappa la vista nel nuovo blocco della cache. Esso tenta poi di copiare i dati dal file mappato al buffer del chiamante; se la copia riesce, l'operazione è completata.

Se la copia fallisce, ciò avviene a causa di una pagina mancante, che spinge il gestore della mm a inviare al gestore di i/o una richiesta di lettura con esplicita esclusione della cache; questi trasmette un'ulteriore richiesta allo stack dei driver del dispositivo per ottenere, questa volta, un'operazione di *paginazione*, che escluda il gestore della cache e legga i dati direttamente dal file nella pagina allocata per il gestore della cache: completata l'operazione, il vacb è impostato in modo da puntare a quella pagina. I dati, ora nella cache, sono copiati nel buffer del chiamante, cosa che completa l'originaria richiesta di i/o. La Figura 21.6 riassume graficamente queste operazioni.

Quando possibile, per le operazioni sincrone sui file memorizzati nella cache, l'i/o viene gestito dal **meccanismo di i/o veloce**. Questo meccanismo è parallelo al normale i/o basato su irp, ma richiama direttamente lo stack del driver piuttosto che passare per un irp, risparmiando tempo e memoria. Poiché non è coinvolto alcun irp, l'operazione non dovrebbe bloccarsi per un lungo periodo di tempo e non può essere accodata a un thread di lavoro. Pertanto, l'operazione fallisce se, quando raggiunge il file system e chiama il gestore della cache, le informazioni non sono già presenti nella cache. Il gestore di i/o ritenta quindi l'operazione utilizzando il normale i/o, con un irp.

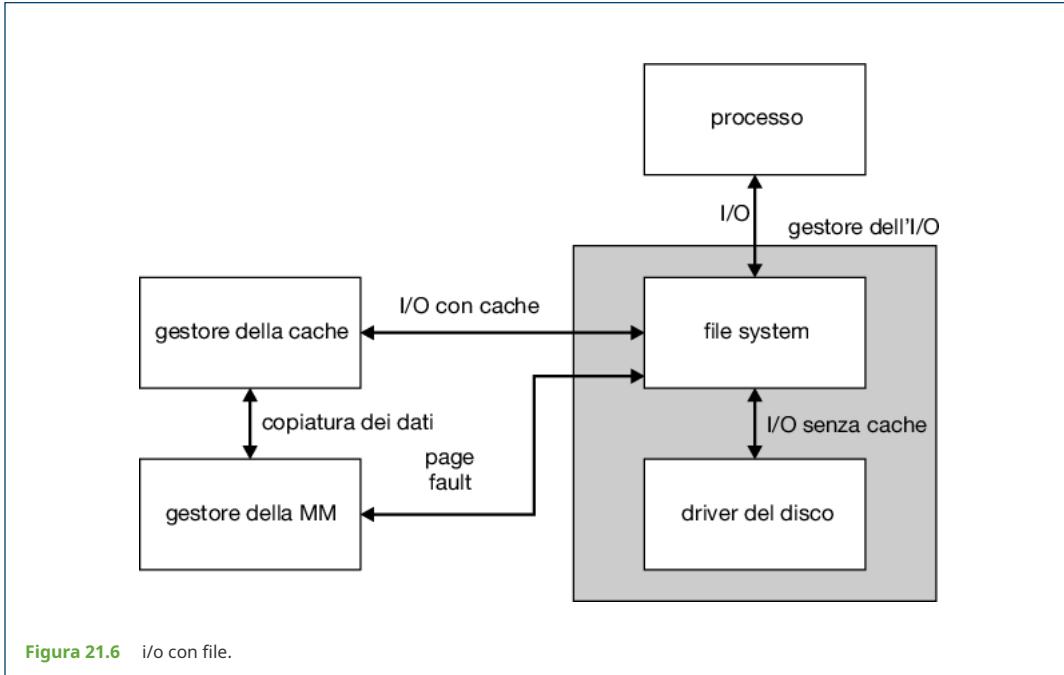


Figura 21.6 i/o con file.

Un'operazione di lettura a livello del kernel si svolge in modo simile, a eccezione del fatto che si può accedere ai dati direttamente dalla cache, anziché copiarli in un buffer nello spazio dell'utente. Per leggere i metadati del file system – le strutture dati che descrivono il file system – il kernel si serve dell'interfaccia del gestore della cache dedicata alla mappatura; per modificarli, usa invece l'interfaccia del gestore della cache per ancorare le pagine. Il **pinning** (cioè l'*ancoraggio*) di una pagina significa associarla a un frame della memoria fisica, in modo che il gestore mm non possa spostarla o scaricarla. Dopo l'aggiornamento dei metadati, il file system richiede al gestore della cache di liberare la pagina ancorata. Una pagina modificata viene etichettata come *dirty*, sicché il gestore mm ricopia la pagina su memoria secondaria.

Per migliorare le prestazioni, il gestore della cache mantiene un piccolo archivio delle richieste di lettura, con cui cerca di predire le richieste successive. Se il gestore rileva uno schema predicable nelle tre precedenti richieste, per esempio un accesso sequenziale in avanti o all'indietro, trasferisce i dati nella cache in anticipo, prima che giunga la successiva richiesta da parte dell'applicazione. In questo modo l'applicazione può trovare i propri dati già nella cache senza dover aspettare l'i/o della memoria secondaria.

Altro compito del gestore della cache è di ordinare al gestore mm lo svuotamento del contenuto della cache. Per default, il gestore della cache opera secondo la tecnica della scrittura differita (*write-back caching*): accumula scritture per 4-5 secondi e poi risveglia il thread di trasferimento su disco. Quando l'operazione deve invece seguire la tecnica della scrittura diretta (*write-through policy*), un processo può segnalarlo, impostando un flag all'apertura del file, o può richiamare un'esplicita funzione di svuotamento della cache.

Un processo di scrittura veloce potrebbe arrivare a riempire tutte le pagine libere della cache prima che il thread di trasferimento abbia la possibilità di attivarsi per scaricare le pagine su disco; il thread previene tale eventualità nel modo seguente. Quando la quantità di memoria libera nella cache si riduce, il gestore della cache sospende temporaneamente i processi che tentano di scrivere dati e risveglia il thread di trasferimento per scaricare le pagine su disco. Se il processo di scrittura implementa in realtà un reindirizzamento per un file system di rete, congelarlo troppo a lungo potrebbe far scadere i trasferimenti di rete e indurre la ritrasmissione dei dati; pertanto, si sprecherebbe banda di rete. Per impedire tale spreco, i reindirizzatori di rete possono chiedere al gestore della cache di limitare le scritture accumulate nella cache.

Poiché un file system di rete necessita di trasferire i dati fra un disco e l'interfaccia di rete, il gestore della cache fornisce anche un'interfaccia dma per spostare i dati direttamente. Il trasferimento diretto evita la duplicazione dei dati attraverso un buffer intermedio.

21.3.5.7 Monitor della sicurezza dei riferimenti

Grazie alla centralizzazione della gestione delle entità del sistema nel gestore degli oggetti, Windows può realizzare un meccanismo uniforme per la validazione run-time degli accessi e per le verifiche di sicurezza per ogni elemento del sistema accessibile dagli utenti. Inoltre, anche le entità non gestite dal gestore degli oggetti possono avere accesso alle routine api per l'esecuzione di controlli di sicurezza. Ogni volta che un thread apre un handle su una struttura dati protetta (come un oggetto), il **monitor della sicurezza dei riferimenti** (*security reference monitor, srm*) controlla il token di sicurezza utilizzato e il descrittore di sicurezza dell'oggetto, che contiene due liste di controllo di accesso, la lista di controllo di accesso discrezionale (dacl) e la lista di controllo di accesso di sistema (sacl), per verificare se il processo ha i diritti di accesso necessari. Il token di sicurezza utilizzato è in genere il token del processo del thread, ma può anche essere il token del thread stesso, come descritto di seguito.

Ogni processo ha un token di sicurezza associato. Quando il processo di login (`lsass.exe`) autentica un utente, il token di sicurezza è collegato al primo processo dell'utente (`userinit.exe`) e copiato per ciascuno dei suoi processi figli. Il token contiene l'identità di

sicurezza (sid) dell'utente, i sid dei gruppi a cui appartiene l'utente, i privilegi dell'utente, il livello di integrità del processo, gli attributi e le attestazioni associati all'utente e qualsiasi altra caratteristica rilevante. Per impostazione predefinita, i thread non hanno i propri token, ma condividono il token comune del processo. Tuttavia, utilizzando un meccanismo chiamato **impersonamento**, un thread in esecuzione in un processo con un token di sicurezza appartenente a un utente può impostare un token specifico del thread appartenente a un altro utente, in modo da poter impersonare quell'utente. A questo punto, il token utilizzato diventa il token del thread e tutte le operazioni, le quote e le limitazioni sono soggette a quel token. In seguito, il thread può scegliere di "ripristinare" la sua vecchia identità rimuovendo il token specifico del thread, in modo che il token effettivo torni a essere quello del processo.

Questa funzione di rappresentazione è fondamentale per il modello client-server, in cui i servizi devono agire per conto di una varietà di client con diversi id di sicurezza. Il diritto di impersonare un utente viene spesso fornito come parte di una connessione da un processo client a un processo server. L'impersonamento consente al server di accedere ai servizi di sistema come se fosse il client e di accedere o creare oggetti e file per conto del client. Il processo del server deve essere affidabile e deve essere scritto attentamente per essere efficace contro gli attacchi. In caso contrario, un client potrebbe assumere il controllo di un processo server e quindi impersonare qualsiasi utente che in seguito effettui una richiesta. Windows fornisce delle api per supportare l'impersonamento a livello alpe (e quindi rpc e dcom), a livello named pipe e a livello Winsock.

L'srm è anche responsabile della manipolazione dei privilegi nei token di sicurezza. Agli utenti sono richiesti privilegi speciali per modificare l'ora del sistema, caricare un driver o modificare le variabili di ambiente del firmware. Inoltre, alcuni utenti possono disporre di potenti privilegi che annullano le regole di controllo di accesso predefinite, per esempio quegli utenti che devono eseguire operazioni di backup o ripristino del file system (aggirando le restrizioni di lettura/scrittura), eseguire il debug dei processi (aggirando le funzionalità di sicurezza) e così via.

Anche il livello di integrità del codice in esecuzione in un processo è rappresentato da un token. I livelli di integrità sono un tipo di meccanismo di etichettatura obbligatorio, come accennato in precedenza. Per impostazione predefinita, un processo non può modificare un oggetto con un livello di integrità superiore a quello del codice in esecuzione nel processo, indipendentemente dalle altre autorizzazioni concesse. Inoltre, non può leggere da un altro oggetto processo con un livello di integrità più elevato. Gli oggetti possono anche proteggersi dall'accesso in lettura modificando manualmente la politica obbligatoria associata al proprio descrittore di sicurezza. All'interno di un oggetto (come un file o un processo), il livello di integrità è memorizzato nella sacl, mentre le autorizzazioni discrezionali tipiche di utente e gruppo sono memorizzate nella dacl.

Sono stati introdotti livelli di integrità per rendere più difficile al codice prendere il controllo di un sistema attaccando i software di analisi di contenuto esterno, come il browser o il lettore pdf, poiché si prevede che tali software funzionino a un livello di integrità basso. Per esempio, Microsoft Edge funziona a "bassa integrità", così come Adobe Reader e Google Chrome. Un'applicazione standard, come Microsoft Word, viene invece eseguita a "media integrità". Infine, un'applicazione eseguita da un amministratore o un programma di installazione vengono solitamente eseguiti con "alta integrità".

La creazione di applicazioni che verranno eseguite a livelli di integrità inferiori impone agli sviluppatori di implementare questa funzionalità di sicurezza, perché devono creare un modello client-server per supportare un broker e un parser o un renderer, come anticipato in precedenza. Per ottimizzare questo modello di sicurezza, Windows 8 ha introdotto il **contenitore di applicazioni**, spesso chiamato **AppContainer**, un'estensione speciale dell'oggetto token. Durante l'esecuzione in un AppContainer, il token di processo di un'applicazione viene automaticamente modificato nei seguenti modi.

1. Il livello di integrità del token è impostato su basso. Ciò significa che l'applicazione non può scrivere o modificare la maggior parte degli oggetti (file, chiavi, processi) sul sistema, né può leggere da qualsiasi altro processo sul sistema.
2. Tutti i sid di gruppi e utenti sono disabilitati (ignorati) nel token. Ipotizzando che l'applicazione sia stata lanciata dall'utente Anne, che appartiene al gruppo World, qualsiasi file accessibile ad Anne o a World sarà inaccessibile a questa applicazione.
3. Quasi tutti i privilegi vengono rimossi dal token. Ciò impedisce che vengano autorizzate chiamate di sistema potenti o operazioni a livello di sistema.
4. Un token speciale AppContainer, che corrisponde all'hash sha-256 dell'identificatore del pacchetto dell'applicazione, viene aggiunto al token. Questo è l'unico identificatore di sicurezza valido nel token, quindi qualsiasi oggetto che desideri essere direttamente accessibile a questa applicazione deve fornire esplicitamente al sid dell'AppContainer l'accesso in lettura o scrittura.
5. Viene aggiunto al token un insieme di sid di abilitazioni, in base al file manifest dell'applicazione. Quando l'applicazione viene installata per la prima volta, queste abilitazioni vengono mostrate all'utente, che deve accettarle prima che l'applicazione venga resa disponibile.

Possiamo osservare che il meccanismo AppContainer modifica il modello di sicurezza da un sistema discrezionale, in cui l'accesso alle risorse protette è definito da utenti e gruppi, in un sistema mandatario, in cui ogni applicazione ha una propria identità di sicurezza e l'accesso avviene sulla base delle applicazioni. Questa separazione di privilegi e autorizzazioni rappresenta un grande passo in avanti nella sicurezza, ma pone un potenziale problema nell'accesso alle risorse, che è però alleviato con l'aiuto di abilitazioni e broker.

Le abilitazioni vengono utilizzate dai broker di sistema implementati da Windows per eseguire varie azioni per conto di applicazioni pacchettizzate. Per esempio, supponiamo che l'applicazione pacchettizzata di Harold non abbia accesso al file system di Harold, poiché il sid Harold è disabilitato. In questa situazione, un broker potrebbe verificare l'abilitazione di riproduzione dei file multimediali (Play User Media) dell'utente e consentire al lettore musicale di leggere tutti i file mp3 presenti nella directory My Music di Harold. Harold non sarà quindi obbligato a contrassegnare tutti i suoi file con il sid dell'AppContainer del suo lettore multimediale preferito, a condizione che l'applicazione sia in possesso dell'abilitazione **Play User Media** e che Harold abbia accettato questa abilitazione quando ha scaricato l'applicazione.

Un ultimo compito dell'srm è la registrazione su log degli eventi di verifica della sicurezza. Lo standard iso Common Criteria (il successore dell'Orange Book del Dipartimento della Difesa americano) richiede che un sistema sicuro debba poter rilevare e registrare tutti i tentativi d'accesso alle proprie risorse, al fine di facilitare l'identificazione dei tentativi d'accesso non autorizzato. Poiché l'srm è responsabile dei controlli d'accesso, esso genera la gran parte delle informazioni di verifica nel log della sicurezza, che vengono quindi scritte da `lsass.exe` nel log degli eventi di sicurezza.

21.3.5.8 Gestore plug-and-play

Il sistema operativo usa il **gestore plug-and-play** (PnP) per riconoscere eventuali cambiamenti nella configurazione hardware del calcolatore e adattarvi opportunamente il sistema. I dispositivi PnP utilizzano protocolli standard per farsi identificare dal sistema. Il gestore PnP riconosce automaticamente i dispositivi installati e mentre il sistema è funzionante rileva eventuali cambiamenti nei dispositivi. Il gestore tiene anche traccia delle risorse impiegate da ciascun dispositivo, insieme con le risorse che potenzialmente potrebbero essere utilizzate e si occupa del caricamento dei driver necessari. Questa gestione delle risorse fisiche (che sono principalmente le interruzioni, i canali dma e gli intervalli di memoria per l'i/o) ha l'obiettivo d'identificare una configurazione hardware che permetta a tutti i dispositivi di funzionare correttamente. Il gestore PnP e il modello di driver di Windows vedono i driver o come driver del bus, che rilevano e enumerano i dispositivi su un bus (per esempio, pci o usb) oppure come driver di funzione (*function driver*), che implementano le funzionalità di un particolare dispositivo sul bus.

Il gestore PnP opera la riconfigurazione dinamica come segue. Innanzitutto ottiene una lista di dispositivi da ciascun driver di bus. Successivamente carica il driver installato (se è necessario, dopo averne trovato uno) e invia un comando `add-device` al driver appropriato per ciascun dispositivo. Lavorando in tandem con gli speciali **arbitri delle risorse** (*resource arbiter*) di proprietà dei vari driver del bus, il manager PnP individua quindi le assegnazioni di risorse ottimali e invia un comando `start-device` a ciascun driver per specificare l'assegnamento delle risorse per quel dispositivo. Se un dispositivo deve essere riconfigurato il gestore PnP invia un comando `query-stop` che richiede al driver se il dispositivo può essere temporaneamente disabilitato. Se il driver può disabilitare il dispositivo, allora tutte le operazioni sospese vengono completate e non si può avviare alcuna nuova operazione. Successivamente, il gestore PnP invia un comando `stop`, dopodiché può riconfigurare il dispositivo con un altro comando `start-device`.

Il gestore PnP prevede altri comandi, come `query-remove`, che si usa quando l'utente sta per rimuovere un dispositivo removibile, come una memoria usb, e che opera in modo simile a `query-stop`. Il comando `surprise-remove` si usa quando c'è un malfunzionamento di dispositivo, oppure, più comunemente, quando un utente estraе un dispositivo senza prima comunicarlo al sistema. Il comando `remove`, infine, richiede che il driver smetta di usare il dispositivo in maniera permanente.

Molti programmi presenti nel sistema sono interessati all'aggiunta o alla rimozione dei dispositivi e per questa ragione il gestore PnP offre funzionalità di notifica. Per esempio, una notifica fornisce ai menu dei file della gui le informazioni di cui hanno bisogno per aggiornare l'elenco dei volumi del disco quando un nuovo dispositivo di archiviazione viene collegato o rimosso. L'installazione di periferiche si traduce spesso in aggiunta a nuovi servizi ai processi `svchost.exe` nel sistema. In molti casi questi servizi vengono eseguiti ogni volta che il sistema si avvia e continuano a funzionare anche se il dispositivo originale non viene mai inserito nel sistema. Windows 7 ha introdotto a suo tempo un **meccanismo di attivazione dei servizi** nel **gestore scm** (*service control manager*) (`services.exe`), che gestisce i servizi di sistema. Mediante questo meccanismo i servizi possono essere configurati per essere eseguiti solo quando scm riceve una notifica dal gestore PnP che segnala che il dispositivo di interesse è stato aggiunto al sistema.

21.3.5.9 Gestore dell'alimentazione

Windows collabora con l'hardware per implementare sofisticate strategie per l'efficienza energetica, come descritto nel Paragrafo 21.2.8. Le politiche che guidano queste strategie sono attuate dal **gestore dell'alimentazione** (*power manager*). Il gestore dell'alimentazione rileva le condizioni del sistema, come il carico della cpu o dei dispositivi di i/o, e migliora l'efficienza energetica riducendo le prestazioni e la reattività del sistema quando le richieste sono basse. Il gestore dell'alimentazione può anche mettere l'intero sistema nella modalità *sleep*, molto efficiente, o anche scrivere tutti i contenuti della memoria su disco e spegnere l'alimentazione per consentire al sistema di passare in uno stato di *ibernazione* (*hyperivation*).

Il vantaggio principale dello *sleep* è la rapidità con cui è possibile passare in questo stato: spesso sono sufficienti pochi secondi dopo aver chiuso il coperchio di un computer portatile. Anche il ripristino dallo *sleep* è abbastanza veloce. Nello stato di *sleep* viene tolta l'alimentazione a cpu e dispositivi di i/o, ma la memoria continua a essere alimentata a sufficienza per non perderne il contenuto. Come abbiamo osservato in precedenza, tuttavia, sui dispositivi mobili, questi pochi secondi peggiorano l'esperienza utente, e quindi il gestore dell'alimentazione utilizza il dam per avviare lo stato di Sospensione Connessa non appena lo schermo viene spento. La Sospensione Connessa blocca virtualmente il dispositivo, ma non lo sospende.

L'*ibernazione* richiede molto più tempo, perché l'intero contenuto della memoria deve essere trasferito sul disco prima che il sistema venga spento. Tuttavia, il fatto che il sistema sia realmente spento costituisce un vantaggio significativo. Se l'alimentazione del sistema viene persa, per esempio quando viene sostituita la batteria su un computer portatile o quando un sistema desktop viene scollegato dalla rete elettrica, i dati memorizzati non verranno persi. A differenza dello spegnimento, l'*ibernazione* salva il sistema attualmente in uso, in modo che un utente possa riprendere da dove aveva lasciato. Inoltre, poiché l'*ibernazione* non richiede alimentazione, un sistema è in grado di rimanere in questo stato per un tempo indefinito. Questa funzione è estremamente utile su sistemi desktop e server, ed è anche utilizzata su laptop quando la batteria raggiunge un livello critico (perché mettere il sistema in modalità *sleep* quando la batteria è scarica potrebbe comportare la perdita di tutti i dati qualora la batteria si esaurisse completamente).

In Windows 7, il gestore dell'alimentazione include anche un gestore dell'alimentazione del processore (ppm), che implementa in modo specifico strategie come il core parking, il throttling e il boosting della cpu, e altro ancora. Windows 8 ha inoltre introdotto il **Power Framework** (pofx), che funziona con i driver di funzione per implementare specifici stati di alimentazione. Grazie al pofx, i dispositivi possono segnalare la loro gestione interna dell'alimentazione (velocità di clock, assorbimento di corrente/potenza e così via) al sistema, che può utilizzare le informazioni per controllarli in maniera più raffinata. Per esempio, invece di limitarsi semplicemente a spegnere o accendere il dispositivo, il sistema può accendere o spegnere le sue componenti specifiche.

Come il gestore PnP, il gestore dell'alimentazione fornisce notifiche al resto del sistema sui cambiamenti nello stato dell'alimentazione. Alcune applicazioni vogliono sapere quando il sistema è in procinto di essere chiuso in modo che possano iniziare a salvare su memoria secondaria il proprio stato e, come menzionato prima, il dam ha bisogno di sapere quando lo schermo viene spento e riacceso.

21.3.5.10 Registro di sistema

Windows mantiene molte delle sue informazioni di configurazione nei database interni, chiamati **hive**, che sono gestiti dal gestore della configurazione di Windows, comunemente noto come il **registro di sistema** (*registry*). Il gestore del registry è implementato come componente dell'executive.

Ci sono hive distinti per le informazioni di sistema, le preferenze di ciascun utente, le informazioni sul software, la sicurezza e le opzioni di avvio. Inoltre, come parte del nuovo modello di applicazione e sicurezza introdotto con AppContainers e con le applicazioni pacchettizzate uwp Modern/Metro in Windows 8, ciascuna di tali applicazioni dispone di un proprio hive, chiamato hive dell'applicazione.

Il registry rappresenta lo stato di configurazione in ogni hive come uno spazio dei nomi gerarchico di chiavi (directory), ognuna delle quali può contenere un insieme di valori di dimensione arbitraria. Nell'api Wn32, questi valori sono dotati di un tipo specifico, come una stringa unicode, una stringa ansi, un intero o dati binari non tipizzati, ma il registry tratta tutti i valori allo stesso modo, lasciando ai livelli api superiori il compito di dedurre la struttura in base al tipo e alla dimensione. Pertanto, per esempio, nulla impedisce a un intero a 32 bit di essere una stringa unicode da 999 byte.

In teoria, le nuove chiavi e i valori vengono creati e inizializzati all'installazione di un nuovo software e vengono poi modificati per riflettere le modifiche nella configurazione di tale software. In pratica, il registry viene spesso usato come un database generico, come un meccanismo di comunicazione tra processi e per molti altri tali scopi.

Riavviare le applicazioni, o addirittura l'intero sistema, ogni volta che viene apportata una modifica nella configurazione sarebbe una seccatura. I programmi si basano allora su vari tipi di notifiche, come quelli forniti dal gestore PnP e dal gestore dell'alimentazione, per venire a conoscenza dei cambiamenti nella configurazione del sistema. Anche il registry fornisce notifiche che permettono ai thread di essere avvisati quando vengono apportate modifiche a una parte del registro di sistema. I thread possono così rilevare i cambiamenti di configurazione apportati al registry e adattarvisi. Inoltre, le chiavi di registro sono oggetti gestiti dal gestore degli oggetti e espongono un oggetto evento al dispatcher. Ciò consente ai thread di mettersi in uno stato di attesa associato all'evento, che il gestore della configurazione segnalera se la chiave (o uno qualsiasi dei suoi valori) viene modificata.

Qualora vengano apportate modifiche significative al sistema, per esempio quando vengono installati aggiornamenti del sistema operativo o dei driver, vi è il pericolo che i dati di configurazione siano danneggiati (per esempio, se un driver funzionante viene sostituito da un driver non funzionante o se l'installazione di un'applicazione non riesce correttamente e lascia informazioni parziali nel registry). Prima di apportare questo genere di modifiche Windows crea un punto di ripristino. Il punto di ripristino contiene una copia degli hive prima del cambiamento e può essere utilizzato per ritornare alla vecchia versione degli hive permettendo a un sistema danneggiato di funzionare di nuovo.

Per migliorare la stabilità della configurazione del registry Windows ha aggiunto, a partire da Windows Vista, un meccanismo di transazione che può essere utilizzato per prevenire che il registry venga aggiornato parzialmente con un insieme di modifiche di configurazione correlate. Le transazioni del registry possono essere parte di transazioni più generali amministrate dal gestore delle transazioni del kernel (*kernel transaction manager*, ktm), che può anche includere transazioni del file system. Le transazioni ktm non hanno la semantica completa delle transazioni nei normali database e non hanno soppiantato il ripristino del sistema nel recupero da danni alla configurazione del registry causati dall'installazione del software.

21.3.5.11 Avvio

L'avvio di un pc Windows inizia quando si fornisce alimentazione all'hardware e il firmware contenuto nella rom è eseguito. In macchine più vecchie questo firmware era conosciuto come il bios, ma i sistemi più moderni utilizzano uefi (Unified Extensible Firmware Interface), più veloce e più generale e consente un migliore utilizzo delle caratteristiche dei processori moderni. Inoltre, uefi include una funzionalità chiamata **avvio protetto** che fornisce controlli di integrità tramite la verifica della firma digitale di tutti i componenti firmware e di avvio. Questo controllo della firma digitale garantisce che solo i componenti di avvio di Microsoft e il firmware del fornitore siano presenti all'avvio, impedendo il caricamento di codice di terze parti nella fase iniziale.

Il firmware esegue la **diagnostica post** (*power-on self-test*), identifica molti dei dispositivi collegati al sistema e li inizializza a uno stato di accensione e poi costruisce la descrizione utilizzata dalla interfaccia avanzata di configurazione e alimentazione (acpi). Successivamente, il firmware trova il disco di sistema, carica il programma `bootmgr` di Windows (`bootmgfw.efi` sui sistemi uefi) e ne inizia l'esecuzione.

Su una macchina che è stata sospesa viene successivamente caricato il programma `winresume.efi` che ripristina dal disco il sistema in esecuzione, in modo che il sistema possa continuare dal punto in cui era prima della sospensione. Su una macchina che è stata arrestata, il `bootmgfw.efi` esegue un'ulteriore inizializzazione del sistema e quindi carica `WinLoad.efi`, che a sua volta carica `hal.dll`, il kernel (`ntoskrnl.exe`), tutti i driver necessari all'avvio, e l'hive di sistema. `WinLoad` trasferisce poi l'esecuzione al kernel.

La procedura è leggermente diversa sui sistemi Windows 10 in cui la modalità Virtual Secure è abilitata (e l'hypervisor è attivato). In questo caso, `winload.efi` caricherà `hvloader.exe` o `hvloader.dll`, che per prima cosa inizializzano l'hypervisor. Sui sistemi Intel, l'hypervisor è `hvix64.exe`, mentre i sistemi amd usano `hvax64.exe`. L'hypervisor imposta quindi vtl 1 (Secure World) e vtl 0 (Normal World) e torna a `winload.efi`, che carica il kernel sicuro (`securekernel.exe`) e le sue dipendenze. Viene quindi chiamato il punto di ingresso del kernel sicuro, che inizializza vtl 1 e ritorna al loader su vtl 0, che riprende con i passaggi sopra descritti.

Mentre il kernel si inizializza, crea diversi processi. Il **processo idle** funge da contenitore di tutti i thread inattivi, in modo che il tempo di inattività della cpu dell'intero sistema possa essere facilmente calcolato. Il processo di sistema contiene tutti i thread interni del kernel e altri thread di sistema creati dai driver per effettuare polling, ordinaria amministrazione e altri lavori in background. Il processo di compressione della memoria, una novità di Windows 10, ha un working set composto da pagine compresse in attesa utilizzate dal gestore dello storage per abbassare la pressione sul sistema e ottimizzare il paging. Infine, se vsm è abilitato, il processo sicuro di sistema rappresenta il fatto che il kernel sicuro è stato caricato.

Il primo processo in modalità utente, anch'esso creato dal kernel, è **smss** (*session manager subsystem, sottosistema per la gestione della sessione*), che è simile al processo `init` di unix. Il processo smss prosegue l'inizializzazione del sistema, tra l'altro instaurando i file di paginazione e creando alcune sessioni utente iniziali. Ogni sessione rappresenta un utente connesso, a eccezione della sessione 0, che viene utilizzata per eseguire processi in background a livello di sistema, come `lsass` e `services`. A ogni sessione viene fornita una propria istanza di un processo smss, che viene chiusa una volta creata la sessione. In ciascuna di queste sessioni, questo smss temporaneo carica il sottosistema dell'ambiente Win32 (`csrss.exe`) e il relativo driver (`win32k.sys`). Quindi, in ogni sessione diversa da 0, smss esegue il processo `winlogon`, che avvia `logonui`. Questo processo acquisisce le credenziali dell'utente

affinché `lsass` possa far accedere un utente, quindi avvia i processi `userinit` ed `explorer`, che implementa la shell di Windows (menu Start, desktop, icone, centro notifiche e così via). Il seguente elenco illustra alcuni di questi aspetti del boot.

- `smss` completa l'inizializzazione del sistema e avvia un processo `smss` per la sessione 0 e uno per la prima sessione di accesso (1).
- `wininit` viene eseguito nella sessione 0 per inizializzare la modalità utente e avviare `lsass` e `services`.
- `lsass`, il sottosistema di sicurezza, implementa funzionalità come l'autenticazione degli utenti. Se le credenziali dell'utente sono protette da vsm attraverso Credential Guard, `lsass` avvia anche `lsaiso` e `bioiso` come Trustlet vtl 1.
- `services` contiene il gestore del controllo dei servizi, o `scm`, che supervisiona tutte le attività in background del sistema, inclusi i servizi in modalità utente. Un certo numero di servizi sarà registrato per l'esecuzione all'avvio del sistema, mentre altri servizi verranno avviati solo su richiesta o quando attivati da un evento come la presenza di un dispositivo.
- `csrss` è il processo del sottosistema d'ambiente Win32. Viene avviato in ogni sessione, principalmente perché gestisce l'input del mouse e della tastiera, che deve essere distinto per ogni utente.
- `winlogon` viene eseguito in ogni sessione di Windows diversa dalla sessione 0 per permettere l'accesso a un utente mediante l'avvio di `logonui`, che presenta l'interfaccia per l'accesso dell'utente.

A partire da Windows xp, il sistema ottimizza il processo di avvio mediante il precaricamento delle pagine dai file su memoria secondaria in base agli avvii precedenti del sistema. I pattern di accesso al disco all'avvio vengono anche utilizzati per disporre i file di sistema su disco in modo da ridurre il numero di operazioni di i/o richieste. Windows 7 ha ridotto i processi necessari per avviare il sistema consentendo l'avvio dei servizi solo quando necessario, piuttosto che all'avvio del sistema. Windows 8 ha ulteriormente ridotto il tempo di avvio parallelizzando tutti i caricamenti dei driver attraverso un pool di thread nel sottosistema PnP e supportando uefi per rendere più efficienti le operazioni di avvio. Tutti questi approcci hanno contribuito a una drastica riduzione del tempo di avvio del sistema, ma alla fine non era più possibile apportare ulteriori miglioramenti.

Per risolvere i problemi relativi all'avvio, in particolare sui sistemi mobili, in cui la ram e i core sono limitati, Windows 8 ha introdotto anche l'**avvio ibrido** (Hybrid Boot). Questa funzione combina l'ibernazione con una semplice disconnessione dell'utente corrente. Quando l'utente arresta il sistema e tutte le altre applicazioni e sessioni sono terminate, il sistema ritorna al prompt `logonui` e quindi viene ibernato. Quando il sistema viene riaccesso, ritorna molto rapidamente alla schermata di accesso, che offre ai driver la possibilità di reinizializzare i dispositivi e dà l'impressione di un avvio completo, mentre in realtà il lavoro per completare l'avvio è ancora in corso.

21.4 Terminal services e cambio rapido utente

Windows fornisce una console con interfaccia grafica che si interfaccia con l'utente tramite tastiera, mouse e monitor. La maggior parte dei sistemi supporta anche l'audio e il video. L'ingresso audio è utilizzato dal software di riconoscimento vocale di Windows, che rende il sistema più comodo e agevola la sua accessibilità per gli utenti disabili. Windows 7 ha aggiunto il supporto per l'**hardware multi-touch**, consentendo agli utenti di inserire dati toccando lo schermo e mediante i movimenti delle dita. La funzionalità di input tramite video viene utilizzata sia per l'accessibilità che per la sicurezza: Windows Hello è una funzionalità di sicurezza in cui è possibile utilizzare avanzate termocamere 3D e riconoscimento facciale per identificare in modo univoco l'utente senza richiedere credenziali tradizionali. Nelle versioni più recenti di Windows 10, l'hardware di rilevamento degli occhi, in cui l'input del mouse viene sostituito dalle informazioni sulla posizione e sullo sguardo degli occhi, può essere utilizzato per l'accessibilità. Le future esperienze di input si evolveranno probabilmente a partire da Microsoft **HoloLens**, un prodotto per la realtà aumentata.

Il pc è stato, ovviamente, pensato come un computer a uso personale, una macchina intrinsecamente a singolo utente. Le moderne versioni di Windows, tuttavia, supportano la condivisione di un pc tra più utenti. Ogni utente che è connesso mediante una gui possiede una sessione creata per rappresentare l'ambiente gui che verrà utilizzato e per contenere tutti i processi creati per eseguire le applicazioni. Windows consente la presenza di sessioni multiple allo stesso tempo su una singola macchina, ma supporta solo una console, costituita da tutti i monitor, le tastiere e i mouse collegati al pc. Una sola sessione alla volta può essere collegata alla console. Dalla schermata di accesso visualizzata sulla console gli utenti possono creare nuove sessioni o connettersi a una sessione esistente creata in precedenza. Ciò consente a più utenti di condividere un singolo pc senza doversi disconnettere e riconnettere a ogni cambio di utente. Microsoft chiama questo uso delle sessioni **cambio rapido utente** (*fast user switching*). Il sistema macos ha una funzionalità simile.

Gli utenti possono anche creare nuove sessioni o connettersi a sessioni esistenti su un pc da una sessione in esecuzione su un altro pc Windows. Il terminal server (ts) collega una delle finestre gui nella sessione locale di un utente alla sessione nuova o esistente, chiamata desktop remoto, sul computer remoto. L'uso più comune del desktop remoto è la connessione a una sessione sul proprio pc di lavoro dal proprio pc di casa. I desktop remoti possono anche essere utilizzati per la risoluzione di problemi da remoto: un utente remoto può essere invitato a condividere una sessione con l'utente connesso sulla console. L'utente remoto può osservare le azioni dell'utente e può persino prendere il controllo del desktop per aiutare nella risoluzione dei problemi riscontrati. Quest'ultimo utilizzo dei servizi terminal sfrutta la funzione di "mirroring", in cui l'utente secondario condivide la stessa sessione piuttosto che crearne una distinta.

Molte società utilizzano sistemi terminal server aziendali mantenuti in data center per eseguire tutte le sessioni utente che accedono alle risorse aziendali, piuttosto che permettere agli utenti di accedere a tali risorse dai pc presenti nell'ufficio di ciascun utente. Ogni computer server può gestire molte decine di sessioni di desktop remoto. In questi casi si parla di **elaborazione thin-client**, in cui i singoli computer si basano su un server per svolgere molte funzioni. Affidandosi ai terminal server dei data center si migliora l'affidabilità, la gestibilità e la sicurezza delle risorse informatiche aziendali.

21.5 File system

Il file system nativo di Windows è ntfs, usato per tutti i volumi locali. Tuttavia, le memorie usb, le memorie flash delle fotocamere e i dischi esterni possono essere formattati con il file system fat a 32 bit per questioni di portabilità, fat è un formato di file system molto più vecchio ed è riconosciuto da molti sistemi diversi da Windows, come il software in esecuzione sulle macchine fotografiche. Uno degli svantaggi è che il file system fat non limita l'accesso ai file agli utenti autorizzati. L'unica soluzione per la protezione dei dati con fat consiste nell'eseguire un programma per crittografare i dati prima di memorizzarli nel file system.

Al contrario, ntfs utilizza acl per controllare l'accesso ai singoli file e supporta la crittografia implicita di singoli file o di interi volumi (utilizzando la funzionalità BitLocker di Windows). ntfs implementa molte altre caratteristiche, tra cui il recupero dei dati, la tolleranza ai guasti, i file e i file system di grandi dimensioni, i flussi multipli di dati, i nomi Unicode, i file sparsi, il journaling, le copie ombra dei volumi e la compressione dei file.

21.5.1 Struttura interna di ntfs

L'entità fondamentale dell'ntfs è il volume: si crea con l'utilità di amministrazione dei dischi del sistema operativo ed è basato su una partizione logica del disco. Un volume può occupare parte di un disco, un intero disco, o anche più dischi. Il gestore dei volumi può proteggere il contenuto del volume con vari livelli di RAid.

L'ntfs non tratta direttamente i singoli settori dei dischi, ma usa **cluster** (*unità di assegnazione*) come unità di allocazione. La dimensione di un cluster, che è una potenza di 2, si stabilisce nella fase di formattazione di un file system. La dimensione di default di un cluster si basa sulla dimensione del volume ed è pari a 4 kb per volumi più grandi di 2 gb. Vista la dimensione degli attuali dischi, ha senso utilizzare una dimensione di cluster maggiore di quella di default per raggiungere prestazioni migliori, anche se il vantaggio che si ottiene va a scapito di una maggior frammentazione interna.

Come indirizzi relativi al disco l'ntfs usa **numeri di cluster logici** (*logical cluster number*, lcn), che assegna numerando i cluster dall'inizio alla fine del disco. Il sistema può quindi calcolare un offset (espresso in byte) relativo al disco fisico moltiplicando l'lcn per la dimensione del cluster.

Un file dell'ntfs non è una semplice sequenza di byte come in unix; è invece un oggetto strutturato costituito da **attributi**. Ciascun attributo di un file è una sequenza indipendente di byte che può essere creata, eliminata, letta e scritta. Alcuni attributi sono comuni a tutti i file, per esempio il nome (o i nomi, se il file ha degli alias come per esempio un nome ms-dos), l'ora di creazione, e il descrittore di sicurezza che stabilisce il controllo degli accessi. I dati dell'utente sono memorizzati in *attributi di dati*.

Molti dei tradizionali file di dati posseggono attributi di dati privi di nome contenenti tutti i dati del file. Si possono però creare altri flussi di dati con nomi espliciti. Le interfacce IProp del modello com (discusso più avanti in questo capitolo) impiegano un flusso di dati con nome per memorizzare proprietà su file ordinari, comprese le anteprime delle immagini. In generale, si possono aggiungere attributi secondo necessità; vi si accede seguendo la sintassi *nomefile:attributo*. ntfs restituisce solo la dimensione dell'attributo privo di nome in risposta a richieste di informazioni sui file, per esempio quando si esegue il comando `dir`.

Ogni file dell'ntfs è descritto da uno o più elementi contenuti in un array memorizzato in un file speciale detto tabella principale dei file (*master file table*, mft); la dimensione di uno di questi elementi è determinata al momento della creazione del file system, e varia da 1 kb a 4 kb. Gli attributi di piccole dimensioni sono memorizzati nella mft stessa, e sono detti attributi residenti; gli attributi più grandi, per esempio la massa anonima dei dati, sono invece attributi non residenti e sono memorizzati in una o più estensioni contigue (*extent*) nei dischi, e un puntatore a ogni estensione è memorizzato nell'mft. Se un file è molto piccolo anche l'attributo dei dati si può memorizzare nell'elemento dell'mft, mentre se un file possiede molti attributi o se è molto frammentato, e richiede quindi molti puntatori per individuare tutti i frammenti, un solo elemento nella mft potrebbe essere insufficiente. In questo caso il file è descritto da un elemento di base del file che punta a elementi aggiuntivi contenenti altri puntatori e attributi.

Ogni file in un volume ntfs possiede un unico identificatore detto riferimento al file (*file reference*); si tratta di 64 bit che consistono di un numero del file di 48 bit e di un numero di sequenza di 16 bit. Il numero del file è il numero dell'elemento (cioè l'indice dell'array) nella mft che descrive il file; il numero di sequenza viene incrementato ogni volta che si riutilizza un elemento della mft. Ciò permette all'ntfs di eseguire controlli interni di coerenza, per esempio per rilevare un riferimento a un file cancellato dopo che l'elemento della mft è stato riusato per un nuovo file.

21.5.1.1 Albero B+ di ntfs

Lo spazio dei nomi dell'ntfs, come in unix, è organizzato come una gerarchia di directory. Ogni directory usa una struttura dati detta albero B+ per memorizzare un indice dei nomi dei file contenuti nella directory; in un albero B+ la lunghezza di ogni percorso dalla radice a una foglia è la stessa e viene eliminato il costo della riorganizzazione dell'albero. La radice indice di una directory contiene il livello più alto di un albero B+; nel caso di una directory di grandi dimensioni, questo livello contiene i puntatori alle estensioni del disco nelle quali è memorizzato il resto dell'albero. Ogni elemento della directory contiene il nome del file e il suo riferimento, così come una copia dell'ora dell'ultimo aggiornamento e della dimensione del file presa dagli attributi residenti nella mft; il motivo per cui le copie di queste informazioni sono memorizzate nella directory è che in questo modo è più efficiente elencarne i contenuti: tutti i nomi, le dimensioni e gli orari sono disponibili all'interno della directory stessa, cosicché non c'è bisogno di recuperare questi attributi dagli elementi della mft per ognuno dei file.

21.5.1.2 Metadati di ntfs

I metadati di ogni volume dell'ntfs sono tutti memorizzati in alcuni file, il primo dei quali è proprio la mft. Il secondo, usato durante la procedura di recupero dei dati a seguito del danneggiamento della mft, contiene una copia dei primi 16 elementi della mft. Anche un certo numero di file successivi è dedicato a compiti specifici, come descritto di seguito.

- Il file **di log** registra tutte le modifiche ai metadati del sistema.

- Il **file del volume** contiene il nome del volume, la versione di ntfs con cui il volume è stato formattato, e un bit che indica se il volume richiede un controllo di coerenza, con l'utilizzo del programma `chkdsk`, a seguito di un possibile guasto.
- La **tavella di definizione degli attributi** indica i tipi di attributi usati nel volume, e le operazioni da eseguire per ognuno di loro.
- La **directory radice** è la directory di livello più alto nella gerarchia del file system.
- Il **file bitmap** tiene traccia dei cluster allocati ai file e dei cluster liberi.
- Il **file d'avvio** contiene il codice d'avvio di Windows e deve risiedere a uno specifico indirizzo del disco in modo che sia facilmente reperibile da un semplice caricatore d'avvio collocato su rom. Esso contiene inoltre l'indirizzo fisico della mft.
- Il **file dei cluster guasti** tiene traccia delle aree del volume non funzionanti; ntfs sfrutta queste informazioni per il ripristino a seguito di errori.

Il mantenimento di tutti i metadati ntfs in veri e propri file offre una proprietà utile. Come discusso nel Paragrafo 21.3.5.6, il gestore della cache memorizza nella cache i dati contenuti nei file. Poiché tutti i metadati ntfs risiedono in file, questi dati possono essere memorizzati nella cache utilizzando gli stessi meccanismi che si impiegano per i dati comuni.

21.5.2 Ripristino

In molti semplici file system un calo di tensione al momento sbagliato può danneggiare le strutture dati del file system così gravemente da compromettere un intero volume. Molti file system di unix, incluso ufs, ma non zfs, memorizzano nei dischi metadati ridondanti e tentano il ripristino dei dati a seguito di un crash del sistema usando il programma `fscck` per controllare tutte le strutture dati del file system e ripristinarne forzatamente uno stato coerente; si tratta di una procedura che spesso comporta l'eliminazione dei file danneggiati e il rilascio di cluster di dati su cui erano stati scritti dati degli utenti, ma che non erano stati correttamente registrati nelle strutture di metadati del file system: è un processo che può essere lento e può portare a significative perdite di dati.

Per conferire robustezza al file system l'ntfs adotta un orientamento differente: tutti gli aggiornamenti delle strutture dati del file system sono eseguiti all'interno di transazioni. Prima che una struttura dati sia modificata, la transazione scrive nel log le informazioni necessarie per ripetere o annullare l'operazione; dopo che la struttura dati è stata cambiata, la transazione scrive un'annotazione di conferma nel log per indicare il successo della transazione.

A seguito di un crash, il sistema è in grado di riportare le strutture dati del file system a uno stato coerente elaborando le informazioni contenute nel log, prima ripetendo le operazioni confermate, poi annullando le operazioni che non erano state confermate prima del crash. Periodicamente (di solito ogni 5 secondi) si scrive un elemento di controllo nel log: il sistema non fa uso degli elementi che precedono l'elemento di controllo al fine di ripristinare i dati dopo un crash; tali elementi si possono quindi eliminare in modo che il file contenente il log non cresca eccessivamente. La prima volta che si accede a un volume dopo l'avviamento del sistema, l'ntfs esegue automaticamente la procedura di ripristino.

Questa strategia non garantisce l'integrità di tutti i contenuti dei file degli utenti dopo un crash; assicura solo che le strutture dati del file system (cioè i file di metadati) siano integre e riflettano uno stato coerente precedente al crash. Sarebbe possibile estendere il metodo delle transazioni ai file degli utenti e Microsoft ha mosso i primi passi in questa direzione con Windows Vista.

Il log è memorizzato nel terzo file di metadati all'inizio del volume; viene creato nella fase di formattazione del file system e ha una dimensione massima fissata. È costituito da due parti: l'area per il log, che è una coda circolare di elementi, e l'area di riavvio, contenente informazioni contestuali quali il punto dell'area per il log dal quale l'ntfs dovrebbe cominciare la lettura durante una procedura di ripristino. In effetti l'area di riavvio contiene due copie di queste informazioni, in modo che il ripristino sia possibile se una copia è stata danneggiata durante il crash.

La funzione di logging è fornita dal *servizio di gestione del file di log*; oltre a scrivere gli elementi nel file di log e a eseguire operazioni di ripristino, questo servizio tiene traccia dello spazio libero nel file di log: quando esso si riduce eccessivamente il servizio di gestione del file di log accoda le transazioni inevase, e l'ntfs sospende tutte le nuove operazioni di i/o. Una volta che tutte le operazioni in corso sono state completate, l'ntfs attiva il gestore della cache per trasferire tutti i dati nei dischi, reimposta il file di log e, infine, esegue le transazioni in coda.

21.5.3 Sicurezza

La sicurezza di un volume dell'ntfs deriva dal modello a oggetti del sistema Windows. Ogni file ntfs fa riferimento a un descrittore di sicurezza, che specifica il proprietario del file, e a una lista di controllo degli accessi, che contiene le autorizzazioni di accesso concesse o negate a ciascun utente o gruppo elencato. Le prime versioni di ntfs utilizzavano un attributo descrittore di sicurezza separato per ciascun file. A partire da Windows 2000, l'attributo descrittore di sicurezza punta a una copia condivisa, con un notevole risparmio di spazio su disco e sulla cache, poiché numerosissimi file hanno descrittori di sicurezza identici.

Nel suo normale funzionamento, ntfs non fa rispettare i permessi nell'attraversamento delle directory dei nomi di percorso del file. Tuttavia, per garantire compatibilità con posix, questi controlli possono essere abilitati. I controlli sull'attraversamento sono intrinsecamente più onerosi, visto che la moderna analisi dei nomi di percorso dei file è basata sul matching dei prefissi anziché sull'analisi, directory per directory, dei nomi di percorso. Il matching dei prefissi è un algoritmo che cerca le stringhe in una cache e trova la voce con la corrispondenza più lunga. Per esempio, la voce `\foo\bar\dir` è una corrispondenza per `\foo\bar\dir2\dir3myfile`. La cache di matching dei prefissi permette di iniziare l'attraversamento della struttura da una posizione più profonda, risparmiando diversi passaggi. Il rispetto dei controlli di attraversamento richiede che l'accesso dell'utente sia controllato a ogni livello di directory. Per esempio, un utente potrebbe non avere il permesso di attraversare `\foo\bar`, quindi iniziare dall'accesso a `\foo\bar\dir` sarebbe un errore.

21.5.4 Compressione

L'ntfs è in grado di eseguire la compressione dei dati di un singolo file o di tutti i file di dati di un'intera directory. Per comprimere un file, divide i dati in unità di compressione, cioè blocchi di 16 cluster contigui. Al momento della scrittura di ciascuna unità di compressione si applica un algoritmo di compressione dei dati; se il risultato occupa meno di 16 cluster, si scrive nei dischi la

versione compressa. Durante la lettura l'ntfs è in grado di determinare se i dati sono stati compressi; in questo caso la lunghezza dell'unità di compressione memorizzata è inferiore ai 16 cluster; per migliorare le prestazioni durante la lettura di unità di compressione contigue, l'ntfs legge e decomprime in anticipo rispetto alle richieste dell'applicazione.

Nel caso di file sparsi o file contenenti prevalentemente zeri, l'ntfs adotta un'altra tecnica per risparmiare spazio: i cluster che contengono solo zeri non sono realmente assegnati o memorizzati nei dischi, ma si lasciano dei buchi (*gap*) nella sequenza dei numeri virtuali dei cluster memorizzati nell'elemento dell'mft relativo al file. Se nella lettura di un file ritrova un buco nella sequenza in questione, l'ntfs riempie semplicemente di zeri la parte interessata del buffer per l'i/o del chiamante. Questa tecnica è adottata anche nel sistema operativo unix.

21.5.5 Punti di montaggio, collegamenti simbolici e collegamenti fisici

I punti di montaggio sono una forma di collegamento simbolico peculiare delle directory di ntfs, introdotti con Windows 2000. Essi offrono agli amministratori la possibilità di organizzare i volumi del disco in maniera più flessibile, rispetto all'utilizzo di nomi globali (quali le lettere delle unità). I punti di montaggio sono realizzati come collegamento simbolico a dati associati contenenti il vero nome del volume. È presumibile che i punti di montaggio soppiantino completamente le lettere delle unità, ma solo in seguito a una lunga transizione dovuta alla dipendenza di molte applicazioni dalle lettere delle unità.

Windows Vista ha introdotto il supporto a una forma più generale di collegamenti simbolici, simili a quelli che si trovano in unix. I collegamenti possono essere assoluti o relativi, possono puntare a oggetti che non esistono e possono puntare a file e directory, anche attraversando i volumi. ntfs supporta anche i collegamenti fisici (hard link), in cui un singolo file ha una voce in più di una directory sullo stesso volume.

21.5.6 Giornale delle modifiche

L'ntfs mantiene un giornale in cui descrive tutti i cambiamenti avvenuti nel file system. I servizi in modalità utente possono ricevere notifica delle modifiche intervenute e individuare in un secondo momento i file modificati, leggendo dal giornale. Il servizio di indicizzazione del contenuto usa il giornale delle modifiche per individuare i file che devono essere nuovamente indicizzati, mentre il servizio di replicazione dei file lo utilizza per identificare i file che devono essere replicati in rete.

21.5.7 Copie ombra dei volumi

Windows ha la capacità di portare un volume in uno stato conosciuto per poi creare una copia ombra (*shadow*), utilizzabile per creare copie di backup coerenti del volume. La stessa tecnica viene chiamata istantanea (*snapshot*) in altri file system. Si tratta di una variante della copiatura su scrittura: i blocchi modificati dopo la creazione di una copia ombra sono mantenuti nella forma originale su quest'ultima. Affinché la copia raggiunga uno stato coerente con il volume è richiesta la cooperazione delle applicazioni, dal momento che il sistema non può sapere quando i dati usati dall'applicazione siano in uno stato stabile a partire dal quale l'applicazione stessa può essere riavviata in modo sicuro.

La versione server di Windows usa copie ombra per rendere prontamente disponibili le vecchie versioni dei file memorizzate sui file server. Gli utenti possono così usufruire di documenti conformi alla versione originale memorizzata sui file server. Gli utenti possono avvalersi di questa caratteristica per recuperare file cancellati per errore, o semplicemente per consultare una versione precedente dei file, il tutto senza la necessità di un nastro contenente copie di riserva.

21.6 Servizi di rete

I servizi di rete di Windows operano sia secondo il modello client-server sia secondo il modello peer-to-peer; il sistema operativo fornisce anche strumenti per la gestione della rete. I componenti del sottosistema di networking permettono la trasmissione dei dati, la comunicazione fra processi, la condivisione dei file attraverso la rete e la possibilità di stampare impiegando stampanti remote.

21.6.1 Interfacce di rete

Nella descrizione dei servizi di rete di Windows si fa riferimento a due **interfacce di rete interne**, dette **ndis** (*network device interface specification*) e **tdi** (*transport driver interface*). L'interfaccia ndis fu sviluppata nel 1989 da Microsoft e da 3Com al fine di separare gli adattatori di rete dai protocolli di trasporto, in modo che gli uni potessero essere modificati senza che ciò avesse effetto sugli altri. L'ndis costituisce l'interfaccia fra lo strato di collegamento dei dati e lo strato di rete nel modello osi e permette a molti protocolli di funzionare alla presenza di diversi adattatori di rete. Analogamente, la tdi è l'interfaccia fra lo strato di trasporto (strato 4) e lo strato di sessione (strato 5) del modello osi: essa permette a ogni componente dello strato di sessione di adoperare ogni meccanismo di trasporto disponibile. (Necessità simili hanno portato al cosiddetto meccanismo *stream* di unix.) L'interfaccia tdi può gestire sia il trasporto privo di connessione sia quello basato sulla connessione, ed è dotata di funzioni per la trasmissione di tutti i tipi di dati.

21.6.2 Protocolli

Nel sistema Windows i protocolli di trasporto sono realizzati come driver che si possono caricare e scaricare dinamicamente, anche se in pratica il sistema deve di solito essere riavviato a seguito di una di queste modifiche. Il sistema Windows dispone di diversi protocolli di networking. Di seguito se ne illustrano alcuni.

21.6.2.1 Protocollo smb

Il **protocollo smb** (*server message-block*) fu introdotto per la prima volta con l'ms-dos 3.1 e si usa per trasmettere richieste di i/o sulla rete. Esso tratta quattro tipi di messaggi. I messaggi *Session control* sono comandi per l'instaurazione e la chiusura della connessione di un oggetto di ridirezione (*redirector*) con una risorsa condivisa del server. I messaggi *File* vengono usati da un redirector per accedere a file del server. I messaggi *Printer* sono usati per trasmettere dati a una coda di stampa remota e per ricevere informazioni sullo stato della stampa, mentre i messaggi *Message* si usano per comunicare con un altro sistema in rete. Una versione del protocollo smb è stato pubblicato come **Common Internet File System (cifs)** ed è disponibile su una serie di sistemi operativi.

21.6.2.2 Protocollo tcp/ip

La pila di protocolli tcp/ip che si usa nella rete Internet è diventata lo standard *de facto* per la comunicazione in rete. Il sistema operativo Windows usa i protocolli tcp/ip per mettere in comunicazione un'ampia gamma di sistemi operativi e di piattaforme. Il pacchetto tcp/ip di Windows comprende un protocollo per la gestione di rete snmp (*simple network-management protocol*), il protocollo dhcp (*dynamic host-configuration protocol*) per la configurazione dinamica dei calcolatori che si connettono alla rete e il vecchio servizio wins (*Windows Internet name service*) di risoluzione dei nomi. Windows Vista ha introdotto una nuova implementazione di tcp/ip che supporta sia **IPv4** sia **IPv6** nello stesso stack di rete. Questa nuova implementazione supporta anche l'offloading dello stack di rete su hardware avanzato, per ottenere prestazioni molto elevate sui server.

Windows fornisce un firewall software che limita le porte tcp utilizzabili dai programmi per la comunicazione di rete. I firewall di rete sono comunemente implementati nei router e costituiscono una misura di sicurezza molto importante. Avere un firewall integrato nel sistema operativo rende un router hardware inutile e fornisce una gestione più integrata e una maggior facilità di utilizzo.

21.6.2.3 Protocollo pptp

Il **protocollo pptp** (*point-to-point tunneling protocol*) è un protocollo messo a disposizione dal sistema Windows per la comunicazione fra moduli server di accesso remoto residenti in calcolatori server Windows e altri sistemi client connessi alla rete Internet; i server possono cifrare i dati trasmessi, e gestiscono **reti private virtuali** (*virtual private network, vpn*) multiprotocollo nella rete Internet.

21.6.2.4 Protocollo HTTP

Il protocollo http è utilizzato per avere (*get*) o fornire (*put*) informazioni utilizzando il World Wide Web. Windows implementa http utilizzando un driver in modalità kernel, quindi i server web sono in grado di funzionare con un basso overhead di connessione allo stack di rete. http è un protocollo abbastanza generale che Windows rende disponibile come opzione di trasporto per l'implementazione di rpc.

21.6.2.5 Protocollo per la realizzazione e lo sviluppo distribuiti di contenuti

Webdav (*web distributed authoring and versioning*) è un protocollo, basato su http, per la realizzazione e lo sviluppo coordinato di contenuti in rete. Windows installa all'interno del file system un oggetto di ridirezione Webdav; ciò permette al protocollo in questione di cooperare con altre funzionalità del file system, quali la crittografia. I file personali possono in tal modo essere memorizzati con sicurezza in un luogo pubblico. Visto che Webdav utilizza http, che è un protocollo *get/put*, Windows deve memorizzare localmente sulla cache i file in modo che i programmi possano usare operazioni di *read/write* su parti di essi.

21.6.2.6 Pipe con nome

Le pipe con nome sono un meccanismo di trasmissione dei messaggi. Un processo può utilizzare le pipe con nome per comunicare con altri processi sulla stessa macchina. Dal momento che le pipe con nome sono accessibili tramite l'interfaccia del file system, i meccanismi di sicurezza utilizzati per oggetti file si applicano anche alle pipe con nome. Il protocollo smb supporta le pipe con nome, che possono dunque essere utilizzate anche per la comunicazione tra processi su sistemi diversi.

Il nome di una pipe con nome segue una convenzione detta unc (*uniform naming convention*). Un nome unc si presenta in modo simile a un tipico nome di file remoto.

Il suo formato è `\nome_del_server\nome_della_condivisione\x\y\z`, dove `nome_del_server` identifica un server della rete; `nome_della_condivisione` specifica qualsiasi risorsa resa disponibili agli utenti della rete, per esempio directory, file, pipe con nome e stampanti, e la parte terminale `\x\y\z` è un ordinario nome di percorso di un file.

21.6.2.7 *rpc*

Una rpc (*remote procedure call*) è un meccanismo di comunicazione client-server che permette a un'applicazione residente in una certa macchina di eseguire una chiamata di procedura relativa a codice residente in un'altra macchina. Quando il client invoca una procedura remota, il sistema delle rpc richiama una procedura locale detta stub che impacca i suoi argomenti in un messaggio e li invia tramite la rete a un determinato processo server, dopo di che si blocca. Nel frattempo, il server estrae gli argomenti dal messaggio, richiama la procedura, impacca i risultati in un messaggio, e li spedisce allo stub del client; quest'ultimo riprende l'elaborazione, riceve il messaggio, estrae i risultati della rpc e li restituisce al chiamante. Impacchettamento ed estrazione degli argomenti sono a volte chiamati marshaling. Il codice dello stub e i descrittori necessari per impacchettare e spacchettare gli argomenti per una rpc sono compilati da una specifica scritta in midl (**Microsoft Interface Definition Language**).

La funzione rpc di Windows aderisce al diffuso standard rpc Distributed Computing Environment, cosicché i programmi che usano la rpc di questo sistema sono facilmente adattabili ad altri sistemi. Lo standard rpc è dettagliato: nasconde molte differenze esistenti fra le architetture dei calcolatori, per esempio la dimensione dei numeri binari e l'ordine dei bit e dei byte nelle parole, specificando formati convenzionali dei dati per i messaggi rpc.

21.6.2.8 *Modello com*

Il modello com (*component object model*) è un meccanismo per la comunicazione fra processi originariamente sviluppato per Windows. Gli oggetti com mettono a disposizione una precisa interfaccia per la manipolazione dei loro dati. A titolo d'esempio, com funge da infrastruttura per la realizzazione della tecnologia Microsoft ole (*object linking and embedding*, ossia "collegamento e integrazione degli oggetti"), grazie alla quale è possibile inserire fogli di calcolo nei documenti Word. Diversi servizi Windows forniscono interfacce com. Windows è dotato di un'estensione distribuita del modello com nota come dcom, che permette lo sviluppo trasparente di applicazioni distribuite su una rete grazie all'uso di rpc.

21.6.3 Redirector e server

Nel sistema operativo Windows un'applicazione può usare l'api di i/o per accedere ai file di un altro calcolatore come se essi fossero locali, ammesso che sul computer remoto giri un server cifs come quelli forniti da Windows. Un oggetto di ridirezione (*redirector*) è un oggetto lato client che trasmette richieste di i/o per file remoti, poi soddisfatte da un server. Per motivi legati alle prestazioni e alla sicurezza, il redirector e i server sono eseguiti in modalità kernel.

Più in dettaglio, l'accesso a un file remoto avviene come segue:

1. l'applicazione richiama il gestore dell'i/o per richiedere l'apertura di un file, fornendo un nome nel formato standard unc;
2. il gestore dell'i/o costruisce un pacchetto di richiesta di i/o com'è descritto nel Paragrafo 21.3.5.5;
3. il gestore dell'i/o rileva che la richiesta si riferisce a un file remoto, e chiama un driver detto **mup** (*multiple universal-naming-convention provider*);
4. il mup invia in modo asincrono l'irp a tutti i redirector registrati;
5. un redirector capace di soddisfare la richiesta risponde al mup; per evitare di porre in futuro la stessa domanda a tutti i redirector, il mup usa una cache per annotare l'identità del redirector capace di trattare questo file;
6. il redirector trasmette la richiesta sulla rete al sistema remoto;
7. i driver di rete del sistema remoto ricevono la richiesta e la passano al driver server;
8. il driver server passa la richiesta al driver appropriato del file system locale;
9. l'appropriato driver del dispositivo è chiamato per accedere ai dati;
10. i risultati sono restituiti al driver server, che li rispedisce al redirector richiedente, il quale li restituisce all'applicazione chiamante tramite il gestore dell'i/o.

Un processo simile avviene nel caso delle applicazioni che usano l'api di rete Win32 anziché i servizi unc, in questo caso invece di un mup si usa un modulo detto instradatore multiplo.

Per motivi legati alla portabilità il redirector e i server usano le api tdi per il trasporto di rete; le richieste stesse sono espresse per mezzo di un protocollo di più alto livello, che di solito è il protocollo smb menzionato nel Paragrafo 21.6.2. L'elenco dei redirector è contenuto nell'hive di sistema del registro.

21.6.3.1 *File system distribuito*

I nomi unc, che fanno esplicito riferimento al nome del server, non si dimostrano sempre appropriati, poiché vi può essere disponibilità di numerosi file server per gli stessi contenuti, e i nomi unc includono esplicitamente il nome del server. Windows fornisce un **protocollo di file system distribuito** (*distributed file system*, dfs), grazie al quale un amministratore di rete può smistare i file da più server utilizzando un solo spazio dei nomi distribuito.

21.6.3.2 *Reindirizzamento delle cartelle e caching lato client*

Al fine di migliorare l'esperienza degli utenti che operano spesso su pc diversi per motivi professionali, Windows permette agli amministratori di assegnare agli utenti un profilo mobile, che mantiene su server le preferenze dell'utente e altre impostazioni. Per memorizzare automaticamente i documenti e gli altri file dell'utente su un server, si applica quindi il reindirizzamento delle cartelle.

Questa tecnica funziona a dovere finché uno dei calcolatori non è più collegato alla rete, come nel caso di un portatile su un aereo. Per fornire agli utenti un accesso non in linea ai loro file reindirizzati, Windows utilizza il **caching lato client** (*client-side caching, csc*). Questo meccanismo è anche impiegato quando il computer è in linea per mantenere copie dei file del server sulla macchina locale, al fine di ottenere prestazioni migliori. I file sono inviati al server non appena cambiano e, se il calcolatore si disconnette, i file sono ancora disponibili; l'aggiornamento del server è rinviato alla volta successiva in cui il calcolatore tornerà in linea.

21.6.4 Domìni

Per molti ambienti di rete esistono gruppi naturali di utenti, per esempio gli studenti di un laboratorio scolastico o gli impiegati di un'azienda. Molto spesso si vuole che tutti i membri di un gruppo possano accedere a risorse condivise messe a disposizione dai calcolatori del gruppo. Per gestire i diritti d'accesso globale all'interno di un tale gruppo, il sistema Windows fa uso del concetto di dominio. In precedenza, questi domini non avevano alcuna relazione con il dns (*domain name system*) che trasformava nomi di calcolatori collegati alla rete Internet in indirizzi ip; ora invece sono strettamente correlati.

In particolare, un dominio Windows è un gruppo di stazioni di lavoro e server che condivide le politiche di sicurezza e ha una comune database degli utenti. Poiché il sistema Windows impiega per l'autenticazione e la fidatezza il protocollo Kerberos, un dominio Windows è quello che nel Kerberos si chiama *realm* (realm). Nel sistema Windows si segue un approccio gerarchico per stabilire relazioni di fiducia tra i domini collegati. Le relazioni di fiducia si basano sul dns e sono relazioni transitiv che si possono trasmettere in entrambe le direzioni nella gerarchia. Questo metodo riduce il numero di relazioni di fiducia richieste da $n * (n - 1)$ a $O(n)$. Le stazioni di lavoro nel dominio si fidano del controllore di dominio affinché dia informazioni corrette sui diritti d'accesso di ciascun utente (caricati nel token di accesso dell'utente da lsass). Ogni utente si riserva comunque la possibilità di limitare l'accesso alle proprie stazioni di lavoro, indipendentemente dalle disposizioni del controllore il dominio.

21.6.5 Active Directory

Active Directory è lo strumento con cui Windows implementa i **servizi del protocollo ldap** (*lightweight directory-access protocol*). Active Directory memorizza le informazioni di topologia relative ai domini, gestisce i profili individuali e di gruppo degli utenti di un dominio, con le relative parole d'ordine e offre una memorizzazione nel dominio per funzionalità Windows che ne hanno bisogno, come i criteri di gruppo (*Windows group policy*).

Gli amministratori impiegano i criteri di gruppo per stabilire standard uniformi per le preferenze e il software dei desktop. Per molte aziende commerciali che si occupano di tecnologia dell'informazione, l'uniformità consente di ridurre drasticamente il costo dell'elaborazione.

21.7 Interfaccia di programmazione

L'api Win32 è l'interfaccia fondamentale ai servizi del sistema operativo Windows. Questo paragrafo descrive cinque aspetti chiave dell'api Win32: l'accesso agli oggetti del kernel, la condivisione degli oggetti fra i processi, la gestione dei processi, la comunicazione fra i processi e la gestione della memoria.

21.7.1 Accesso agli oggetti del kernel

Il kernel del sistema operativo Windows mette a disposizione dei programmi applicativi molti servizi: i programmi usufruiscono di questi servizi manipolando oggetti del kernel. Un processo accede a un oggetto del kernel di nome `xxx` chiamando la funzione `CreateXXX()` per ottenere un handle di un'istanza di `xxx`; questo handle è unico per il processo. Se la funzione `Create` non va a buon fine riporta il valore `0` o una costante specifica detta `INVALID_HANDLE_VALUE`, secondo l'oggetto che si sta aperto. Un processo può chiudere un handle richiamando la funzione `CloseHandle()`, e il sistema può eliminare l'oggetto se il contatore che totalizza il numero degli handle che fanno riferimento all'oggetto in tutti i processi arriva a zero.

21.7.2 Condivisione degli oggetti tra processi

Il sistema Windows fornisce ai processi tre modi di condividere un oggetto. Il primo consiste nel fatto che un processo figlio può ereditare un handle dell'oggetto: quando invoca la funzione `CreateXXX()`, il genitore fornisce una struttura `SECURITIES_ATTRIBUTES` il cui campo `bInheritHandle` ha valore `TRUE`; questo campo crea un handle ereditabile. A questo punto si può creare il processo figlio, passando il valore `TRUE` all'argomento `bInheritHandle` della funzione `CreateProcess()`. Nella Figura 21.7 è mostrato un esempio di codice che crea l'handle di un semaforo ereditato dal processo figlio.

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE hSemaphore = CreateSemaphore(&sa, 1, 1, NULL);
WCHAR wszCommandLine[MAX_PATH];
StringCchPrintf(wszCommandLine, countof(wszCommandLine),
    L"another process.exe %d", hSemaphore);
CreateProcess(L"another process.exe", wszCommandLine,
    NULL, NULL, TRUE, . . .);
```

Figura 21.7 Codice che permette a un figlio di condividere un oggetto ereditandone l'handle.

Nell'ipotesi che il processo figlio sappia quali handle sono condivisi, la condivisione degli oggetti permette di realizzare la comunicazione fra genitore e figlio; nell'esempio della Figura 21.7 il processo figlio otterrebbe il valore dell'handle dal primo argomento della riga di comando e potrebbe quindi condividere il semaforo con il processo genitore.

Il secondo metodo di condivisione degli oggetti presuppone che l'oggetto abbia ricevuto un nome dal processo che lo ha creato, cosicché un altro processo possa richiedere l'apertura dell'oggetto riferendosi al suo nome. Questa tecnica ha due svantaggi: il primo è che il sistema operativo Windows non fornisce un modo di controllare se un oggetto con un certo nome già esiste; il secondo è che lo spazio dei nomi degli oggetti è globale, senza alcuna distinzione relativa ai tipi di oggetti. Due applicazioni potrebbero per esempio creare un oggetto denominato `foo`, mentre ciò che si voleva erano due oggetti distinti e forse anche di diverso tipo.

Il vantaggio degli oggetti con nome è che processi non correlati possono facilmente condividerli: un primo processo richiamerebbe una delle funzioni `CreateXXX()`, e fornirebbe un nome come parametro; un secondo processo potrebbe ottenere un handle di questo oggetto chiamando `OpenXXX()` (o `CreateXXX()`) con lo stesso nome, com'è mostrato dall'esempio nella Figura 21.8.

```

// Processo A
. . .
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, L"MySEM1");
. . .
// Processo B
. . .
HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, L"MySEM1");
. . .

```

Figura 21.8 Codice per la condivisione di un oggetto ricercandone il nome.

Il terzo modo di condividere oggetti si basa sulla funzione `DuplicateHandle()`; esso richiede altri metodi di comunicazione fra processi per trasmettere l'handle duplicato. Se un processo possiede un handle con un certo valore, un altro processo può ottenerne uno dello stesso oggetto, e pertanto condividerlo; un esempio di questo metodo è mostrato nella Figura 21.9.

```

// il processo A vuole fornire al processo B
// l'accesso a un semaforo
// processo A
DWORD dwProcessBId; // deve esserci; da qualche meccanismo IPC
HANDLE hSemaphore = CreateSemaphore(NULL, 1, 1, NULL);
HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE,
    dwProcessBId);
HANDLE hSemaphoreCopy;
DuplicateHandle(GetCurrentProcess(), hSemaphore,
    hProcess, &hSemaphoreCopy,
    0, FALSE, DUPLICATE_SAME_ACCESS);
// invia il valore del semaforo al Processo B
// usando un messaggio o un oggetto in memoria condivisa
. . .

// Processo B
HANDLE hSemaphore = // valore del semaforo dal messaggio
// uso di hSemaphore per accedere al semaforo
. . .

```

Figura 21.9 Codice per la condivisione di un oggetto tramite il passaggio di un handle.

21.7.3 Gestione dei processi

Nel sistema Windows un processo è un'istanza caricata di un'applicazione e un thread è un'unità eseguibile di codice che può essere sottoposta a scheduling dal dispatcher del kernel: un processo contiene uno o più thread. Un processo viene creato quando un thread in qualche altro processo invoca l'api `CreateProcess()`, che carica le librerie dinamiche usate dal processo e crea un thread iniziale nel processo; tramite la funzione `CreateThread()` si possono creare thread aggiuntivi: ogni thread possiede il suo stack, la cui dimensione predefinita è di 1 mb se non è altrimenti specificato da un argomento di `CreateThread()`.

21.7.3.1 Regola di scheduling

Le priorità nell'ambiente Win32 sono basate sul modello di scheduling del kernel nativo (nt), ma non tutti i valori di priorità possono essere scelti. L'ambiente Win32 usa quattro classi di priorità:

1. `IDLE_PRIORITY_CLASS` (livello di priorità 4)
2. `BELOW_NORMAL_PRIORITY_CLASS` (livello di priorità 6)
3. `NORMAL_PRIORITY_CLASS` (livello di priorità 8)
4. `ABOVE_NORMAL_PRIORITY_CLASS` (livello di priorità 10)
5. `HIGH_PRIORITY_CLASS` (livello di priorità 13)

6. REALTIME_PRIORITY_CLASS (livello 24).

Ordinariamente i processi appartengono alla classe NORMAL_PRIORITY_CLASS, sempre che il genitore del processo non sia di classe IDLE_PRIORITY_CLASS, oppure non sia stata specificata un'altra classe al momento della chiamata alla funzione `CreateProcess()`. La classe di priorità di un processo diventa la classe predefinita per tutti i thread eseguiti nel processo. La classe di priorità si può modificare tramite la funzione `SetPriorityClass()`, o passando un argomento al comando `START`. Si noti che solo gli utenti che godono del privilegio che consente d'incrementare le priorità di scheduling possono assegnare un processo alla classe REALTIME_PRIORITY_CLASS; gli amministratori e gli appartenenti al gruppo *Power users* godono automaticamente di questo privilegio.

Quando un utente passa da processi interattivi a eseguire un carico di lavoro, e viceversa, il sistema deve schedulare i thread appropriati in modo da fornire una buona reattività, ma ciò porta a un accorciamento dei quanti di tempo di esecuzione. Inoltre, una volta che l'utente ha scelto un particolare processo, si aspetta da quel processo un buon throughput. Per questo motivo, Windows ha una regola di scheduling speciale per i processi che non appartengono alla classe REALTIME_PRIORITY_CLASS. Windows distingue tra il processo associato alla finestra in primo piano sullo schermo e gli altri processi (in background). Quando un processo si sposta in primo piano, Windows aumenta il quanto di tempo a disposizione di tutti i suoi thread di un fattore 3: i thread cpu-bound del processo in primo piano verranno quindi eseguiti tre volte più a lungo rispetto a thread simili dei processi in background. Questo comportamento non è abilitato per i sistemi server, poiché i sistemi server utilizzano sempre quanti di tempo maggiori (di un fattore 6) rispetto ai sistemi client. Per entrambi i tipi di sistemi, tuttavia, i parametri di scheduling possono essere personalizzati tramite la finestra di dialogo del sistema o le chiavi di registro appropriate.

21.7.3.2 Priorità dei thread

La priorità iniziale di un thread è determinata dalla sua classe, ma si può modificare tramite la funzione `SetThreadPriority()`; essa accetta un argomento che specifica una priorità relativa alla priorità di base della classe del thread:

- `THREAD_PRIORITY_LOWEST`: base - 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base - 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Per regolare la priorità si usano altri due valori convenzionali. Il kernel ha due classi di priorità (Paragrafo 21.3.4.3): la classe per le elaborazioni real-time (livelli 16-31) e la classe a priorità variabile (livelli 1-15); `THREAD_PRIORITY_IDLE` rappresenta il livello di priorità 16 per i thread d'elaborazione real-time, e il livello di priorità 1 per i thread a priorità variabile.

`THREAD_PRIORITY_TIME_CRITICAL` rappresenta il livello di priorità 31 per i thread d'elaborazione real-time, e il livello 15 per i thread a priorità variabile.

Il kernel regola dinamicamente la priorità di un thread in funzione del fatto che si tratti di un thread con prevalenza di i/o o con prevalenza d'elaborazione; l'api Win32 fornisce un modo di disattivare questo processo di regolazione tramite le funzioni `SetProcessPriorityBoost()` e `SetThreadPriorityBoost()`.

21.7.3.3 Sospensione e ripristino dei thread

Un thread può essere creato in uno stato sospeso o può essere collocato in uno stato sospeso in seguito utilizzando la funzione `SuspendThread()`. Prima che un thread sospeso possa essere schedulato dal dispatcher del kernel deve essere ripristinato dallo stato sospeso con l'uso della funzione `ResumeThread()`. Entrambe le funzioni impostano un contatore in modo che se un thread viene sospeso per due volte, deve essere ripristinato per due volte prima che possa essere eseguito.

21.7.3.4 Sincronizzazione dei thread

Per sincronizzare l'accesso concorrente agli oggetti condivisi dai thread il kernel fornisce oggetti di sincronizzazione come semafori e mutex, che sono oggetti dispatcher, come descritto nel Paragrafo 21.3.4.3. I thread possono anche essere sincronizzati con servizi del kernel che operano su oggetti kernel, come thread, processi e file, perché anche questi sono oggetti dispatcher. La sincronizzazione con oggetti dispatcher del kernel può essere ottenuta mediante l'uso delle funzioni `WaitForSingleObject()` e `WaitForMultipleObjects()`, che aspettano la segnalazione di uno o più oggetti dispatcher.

Un altro metodo di sincronizzazione è disponibile per i thread all'interno dello stesso processo che vogliono eseguire codice esclusivo. L'oggetto sezione critica Win32 è un oggetto mutex in modalità utente che può spesso essere acquisito e rilasciato senza entrare nel kernel. In un sistema multiprocessore, una sezione critica Win32 tenterà di ciclare continuamente in attesa del rilascio di una sezione critica detenuta da un altro thread. Se l'attesa dura troppo, il thread che vuole effettuare l'acquisizione allocherà un mutex kernel e cederà la sua cpu. Le sezioni critiche sono particolarmente efficienti perché il mutex kernel viene allocato solo quando c'è contesa e quindi utilizzato solo dopo aver tentato il continuo ciclare. La maggior parte dei mutex nei programmi non è mai effettivamente contesa: il risparmio è quindi notevole.

Prima di utilizzare una sezione critica, alcuni thread nel processo devono chiamare `InitializeCriticalSection()`. Ogni thread che voglia acquisire il mutex chiama `EnterCriticalSection()` e poi chiama `LeaveCriticalSection()` per rilasciare il mutex. Esiste anche una funzione `TryEnterCriticalSection()` che tenta di acquisire il mutex senza che il thread si blocchi.

Per i programmi che utilizzano lock di lettura-scrittura in modalità utente piuttosto che i mutex, Win32 supporta i **lock swr (slim reader-writer)**. I lock swr hanno api simili a quelle per le sezioni critiche, come `InitializeSRWLock`, `AcquireSRWLockXXX` e `ReleaseSRWLockXXX`, dove `XXX` ha valore `Exclusive` o `Shared`, a seconda se il thread desidera l'accesso in scrittura o se è sufficiente l'accesso in lettura all'oggetto protetto da lock. L'api Win32 supporta anche le **variabili condizionali**, che possono essere utilizzate sia con le sezioni critiche sia con i lock swr.

21.7.3.5 Pool di thread

La frequente creazione e cancellazione di thread può imporre un alto costo alle applicazioni e ai servizi che usano ciascuno di loro per svolgere piccole quantità di lavoro. I pool di thread di Win32 offrono tre servizi ai programmi in modalità utente: una coda a cui si possono delegare le richieste di lavoro (tramite la funzione `SubmitThreadPoolWork()`), una api (`RegisterWaitForSingleObject()`) utilizzabile per legare le chiamate di ritorno agli handle in attesa e alcune api per lavorare con i timer (`CreateThreadpoolTimer()` e `WaitForThreadpoolTimerCallbacks()`) e per creare un legame con le chiamate di ritorno alle code di completamento di i/o (`BindIoCompletionCallback()`).

L'obiettivo dei pool di thread è migliorare le prestazioni e ridurre l'impatto della memoria: i thread sono relativamente costosi e un processore non può eseguire un solo thread per volta, indipendentemente dal numero di thread disponibili. I pool di thread tentano di ridurre il numero di thread eseguibili con un lieve differimento delle richieste di lavoro (riutilizzando ciascun thread per più richieste), fornendo al contempo thread a sufficienza per sfruttare efficacemente le cpu della macchina. Le api per la chiamata di ritorno associata ad attese o timer permettono al pool di thread di ridurre il numero di thread in un processo, usando molto meno di quanti ne sarebbero necessari se un processo dovesse dedicare thread separati per servire l'attesa di un handle, di un timer o di una porta di completamento.

21.7.3.6 Fibre

Una fibra (*fiber*) è codice utente sottoposto a un regime di scheduling definito dall'utente. Le fibre lavorano interamente in modalità utente e il kernel non è a conoscenza della loro esistenza. Il meccanismo della fibra utilizza i thread di Windows come se fossero delle cpu per l'esecuzione delle fibre. Le fibre sono pianificate in maniera cooperativa, nel senso che non subiscono mai prelazione, ma devono cedere esplicitamente il thread su cui sono in esecuzione. Quando una fibra cede un thread, un'altra fibra può essere pianificata per l'esecuzione su quel thread dal sistema run-time (il codice run-time del linguaggio di programmazione).

Il sistema crea una fibra richiamando una tra le due funzioni `ConvertThreadToFiber()` o `CreateFiber()`; la differenza principale fra queste due funzioni è che la seconda non avvia l'esecuzione della fibra appena creata: per cominciare l'esecuzione l'applicazione deve richiamare la funzione `SwitchToFiber()`; per terminare l'esecuzione di una fibra l'applicazione deve richiamare la funzione `DeleteFiber()`.

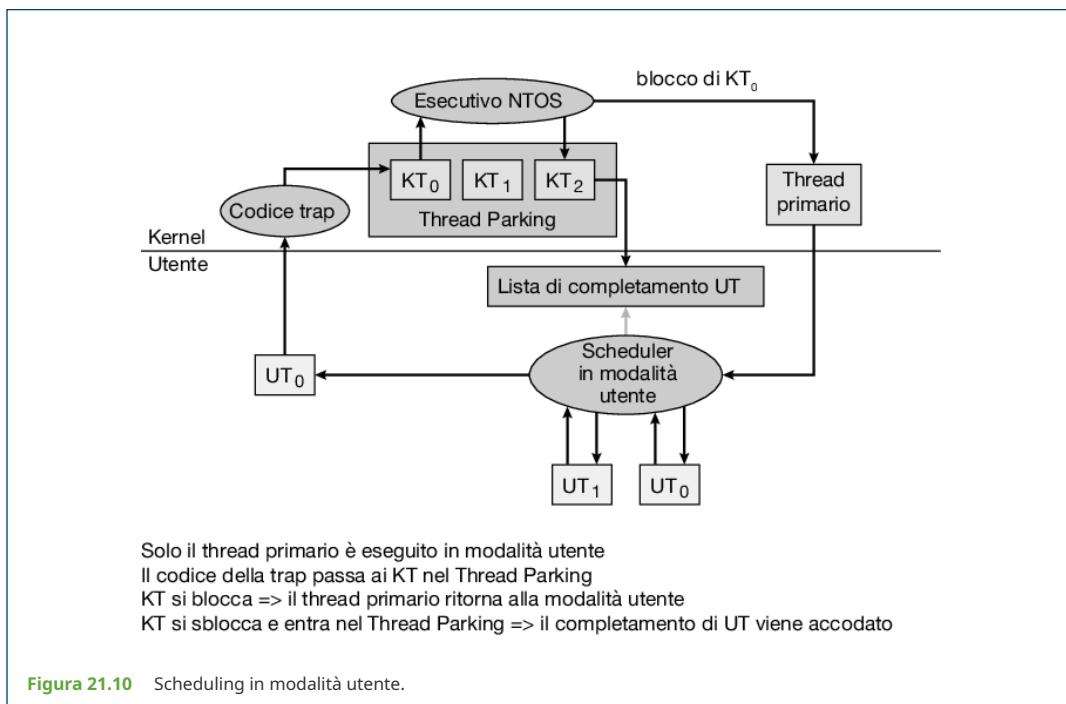
Le fibre non sono raccomandate per i thread che utilizzano le api Win32 invece delle funzioni della libreria C standard a causa di potenziali incompatibilità. I thread Win32 in modalità utente possiedono un teb (*thread-environment block*) che contiene numerosi campi specifici del thread utilizzati dalle api Win32. Le fibre devono condividere il teb del thread su cui sono in esecuzione. Ciò può portare alcuni problemi quando un'interfaccia Win32 mette le informazioni di stato nel teb per una data fibra e in seguito le informazioni vengono sovrascritte da una fibra differente. Le fibre sono incluse nell'api Win32 per facilitare il porting di applicazioni legacy unix che sono state scritte per un modello di thread in modalità utente come Pthreads.

21.7.3.7 UMS e ConcRT

Un nuovo meccanismo di Windows 7, lo scheduling in modalità utente (*User-Mode Scheduling*, ums), risolve diversi limiti delle fibre. Si ricordi, prima di tutto, che le fibre sono inaffidabili per l'esecuzione di api Win32 perché non hanno propri teb. Quando un thread che esegue una fibra si blocca nel kernel, lo scheduler utente perde per un certo tempo il controllo della cpu, mentre il dispatcher del kernel si fa carico dello scheduling. I problemi possono verificarsi quando le fibre cambiano lo stato nel kernel di un thread, come i token di priorità o di personalizzazione, o quando iniziano un i/o asincrono.

ums fornisce un modello alternativo riconoscendo che ogni thread di Windows è in realtà formato da due thread: un thread del kernel (kt) e un thread utente (ut). Ogni tipo di thread ha un proprio stack e un proprio insieme di registri salvati. kt e ut appaiono come un singolo thread al programmatore perché i thread ut non si possono mai bloccare, ma devono sempre entrare nel kernel dove avviene un passaggio implicito al corrispondente kt. ums utilizza i teb di ogni ut per identificarli univocamente. Quando un ut entra nel kernel viene effettuato un passaggio esplicito al kt corrispondente all'ut identificato dal teb corrente. La ragione per cui il kernel non sa quale ut è in esecuzione è che i thread ut possono invocare uno scheduler in modalità utente, come fanno le fibre. In ums, però, lo scheduler scambia i thread ut, oltre a scambiare i teb.

Quando un ut entra nel kernel, il suo kt può essere bloccato. Quando ciò accade, il kernel passa a un thread di scheduling, che ums chiama thread primario, e usa questo thread per rientrare nello scheduler in modalità utente in modo da poter scegliere un altro ut da eseguire. Alla fine, un kt bloccato completerà le sue operazioni e sarà pronto per tornare alla modalità utente. Poiché ums ha già riattivato lo scheduler in modalità utente per eseguire un ut diverso, accoda l'ut corrispondente al kt completato a una lista di completamento in modalità utente. Quando lo scheduler in modalità utente deve scegliere un nuovo ut a cui passare, può esaminare la lista di completamento e trattare qualsiasi ut sulla lista come candidato per lo scheduling. Le funzionalità principali di ums sono mostrate nella Figura 21.10.



A differenza delle fibre, ums non è destinato all'utilizzo diretto da parte del programmatore. I dettagli della scrittura di uno scheduler in modalità utente possono essere molto impegnativi e ums non include un tale scheduler. Gli scheduler provengono invece dalle librerie di un linguaggio di programmazione costruite su ums. Microsoft Visual Studio 2010 dispone di Concurrency Runtime (Concrt), un ambiente di programmazione concorrente per C++. Concrt fornisce uno scheduler in modalità utente insieme a funzionalità per la scomposizione dei programmi in task che possono essere poi pianificati sulle cpi disponibili. Concrt fornisce il supporto per gli stili `par_for` dei costrutti, oltre a una gestione rudimentale delle risorse e a primitive di sincronizzazione dei task. Tuttavia, a partire da Visual Studio 2013, la modalità di scheduling ums non è più disponibile in Concrt. Alcune significative misure delle prestazioni hanno mostrato che i programmi veramente paralleli e ben scritti non trascorrono molto tempo nel cambio di contesto tra le loro attività. I vantaggi offerti da ums in questo caso non superano la complessità della gestione di uno scheduler separato: in alcuni casi, anche lo scheduler nt predefinito ha ottenuto risultati migliori.

21.7.3.8 Winsock

Winsock è l'api di Windows per i socket. Winsock è un'interfaccia a livello di sessione in gran parte compatibile con i socket unix, ma dotata di alcune estensioni aggiuntive di Windows. Winsock fornisce un'interfaccia standardizzata a molti protocolli di trasporto che possono avere diversi schemi di indirizzamento, in modo che qualsiasi applicazione Winsock possa funzionare su qualsiasi pila di protocolli compatibile con Winsock. Winsock ha subito un importante aggiornamento in Windows Vista con l'aggiunta del tracing, del supporto IPv6, della impersonazione, di nuove api di sicurezza e di molte altre caratteristiche.

Winsock segue il modello Windows Open System Architecture (wosa), che fornisce un'interfaccia del provider di servizi (spi) standard tra le applicazioni e i protocolli di rete. Le applicazioni possono caricare e scaricare i protocolli stratificati che costruiscono funzionalità addizionali, come una sicurezza aggiuntiva, sopra agli strati dei protocolli di trasporto. Winsock supporta operazioni asincrone e notifiche, multicasting affidabile, socket sicuri e socket in modalità kernel. C'è anche il supporto per modelli di utilizzo semplificati, come la funzione `wsaConnectByName()` che accetta la destinazione come una stringa che specifica il nome o l'indirizzo IP del server e il servizio o il numero di porta di destinazione.

21.7.4 Comunicazione fra processi mediante messaggi di Windows

Le applicazioni Win32 gestiscono la comunicazione tra processi in molti modi. Un tipico modo, che offre alte prestazioni, consiste nell'utilizzo di rpc locali o named pipe; un altro nell'utilizzo di oggetti kernel condivisi, come oggetti di sezione con nome, e un oggetto di sincronizzazione, come un evento. Un ulteriore altro metodo, particolarmente diffuso per le applicazioni con interfaccia utente grafica Win32, è basato sullo scambio di messaggi di Windows. Un thread può spedire un messaggio a un altro thread o a una finestra richiamando una fra le funzioni `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()` e `SendMessageCallback()`. La differenza fra le funzioni `Post` e quelle `Send` è che le prime sono asincrone: restituiscono immediatamente il controllo, quindi il thread chiamante non sa esattamente quando il messaggio giungerà a destinazione; le funzioni `Send`, invece, sono sincrone e bloccano quindi il chiamante finché il messaggio non sia stato recapitato.

Un thread può trasmettere dati insieme con un messaggio; in questo caso i dati devono essere copiati, perché processi diversi hanno spazi d'indirizzi distinti. Il sistema copia i dati richiamando `SendMessage()` per trasmettere un messaggio di tipo `WM_COPYDATA` con

una struttura dati `COPYDATASTRUCT` contenente la lunghezza e l'indirizzo dei dati da trasferire; quando si trasmette il messaggio, il sistema operativo copia i dati in una nuova regione della memoria, e ne fornisce l'indirizzo virtuale al processo ricevente.

Ogni thread Win32 ha la propria coda d'ingresso dalla quale riceve messaggi. Si tratta di un'architettura più affidabile rispetto alla condivisione della coda d'ingresso dell'ambiente Windows a 16 bit, perché con le code distinte un'applicazione bloccata non impedisce la ricezione dei dati per le altre applicazioni. Se un'applicazione Win32 non impiega la funzione `GetMessage()` per trattare gli eventi nella sua coda d'ingresso, la coda si riempirà, e dopo circa 5 secondi il sistema contrasseggerà l'applicazione come "Non rispondente". Si noti che lo scambio di messaggi è soggetto al meccanismo del livello di integrità introdotto in precedenza. Pertanto, un processo potrebbe non essere in grado di inviare un messaggio come `WM_COPYDATA` a un processo con un livello di integrità superiore, a meno che non venga utilizzata un'api Windows speciale per rimuovere la protezione (`ChangeWindowMessageFilterEx`).

21.7.5 Gestione della memoria

L'api Win32 mette a disposizione delle applicazioni molti modi per utilizzare la memoria: la memoria virtuale, i file associati allo spazio d'indirizzi di memoria, gli *heap*, la memoria locale dei thread e la memoria fisica *awe*.

21.7.5.1 Memoria virtuale

Per prenotare o eseguire l'assegnazione di una regione della memoria virtuale, un'applicazione invoca la funzione `VirtualAlloc()`, e per eseguire le operazioni inverse invoca `VirtualFree()`. Queste funzioni permettono all'applicazione di specificare l'indirizzo virtuale a partire dal quale si deve assegnare la memoria (in caso contrario, viene selezionato un indirizzo casuale, opzione consigliata per motivi di sicurezza); esse operano con multipli della dimensione della pagina di memoria ma, per ragioni storiche, restituiscono sempre la memoria allocata con arrotondamento a 64 kb. Alcuni esempi di queste funzioni sono riportati dalla Figura 21.11. Le funzioni `VirtualAllocEx()` e `VirtualFreeEx()` possono essere utilizzate per allocare e liberare memoria in un processo esterno, mentre la `VirtualAllocExNuma()` può essere utilizzata per far leva sulla località di memoria sui sistemi numa.

```
// riserva 16 MB in cima al nostro spazio di indirizzi
PVOID pBuf = VirtualAlloc(NULL, 0x1000000,
    MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
// assegna gli 8 MB superiori dello spazio allocato
VirtualAlloc((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_COMMIT, PAGE_READWRITE);
// fa qualcosa con la memoria
. .
// annulla l'assegnazione
VirtualFree((LPVOID)((DWORD_PTR)pBuf + 0x800000), 0x800000,
    MEM_DECOMMIT);
// rilascia tutto lo spazio allocato
VirtualFree(pBuf, 0, MEM_RELEASE);
```

Figura 21.11 Frammenti di codice per l'assegnazione della memoria virtuale.

21.7.5.2 Mappatura di file in memoria

Un'applicazione può anche far uso della memoria associando un file al proprio spazio d'indirizzi; si tratta anche di un metodo conveniente per la condivisione di memoria fra due processi: entrambi i processi associano lo stesso file alla loro memoria virtuale. La mappatura di un file è un procedimento a più fasi, come dimostra l'esempio della Figura 21.12.

```

// imposta lo spazio per la mappatura del file a 8MB
DWORD dwSize = 0x800000;
// apre il file, o lo crea se non esiste
HANDLE hFile = CreateFile(L"somefile.ext",
    GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// crea la mappatura del file
HANDLE hMap = CreateFileMapping(hFile,
    PAGE_READWRITE | SEC_COMMIT, 0, dwSize, L"SHM_1");
// ottiene una vista dello spazio mappato
PVOID pBuf = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS,
    0, 0, 0, dwSize);
// fa qualcosa con il file mappato
...
// annulla la mappatura del file
UnmapViewOfFile(pBuf);
CloseHandle(hMap);
CloseHandle(hFile);

```

Figura 21.12 Frammenti di codice per la mappatura in memoria di un file.

Se un processo desidera mappare uno spazio d'indirizzi al solo scopo di condividere memoria con un altro processo, la presenza di un file è superflua: il processo può invocare `CreateFileMapping()` specificando come handle del file il valore `0xffffffff`, oltre a una certa dimensione; l'oggetto risultante si può condividere per ereditarietà, per nome o per duplicazione dell'handle.

21.7.5.3 Heap

Il terzo modo in cui un'applicazione può usare la memoria, analogo all'uso di `malloc()` e `free()` nel C standard o di `new()` e `delete()` in C++, è rappresentato dallo heap. Uno *heap* nell'ambiente Win32 è semplicemente una regione riservata di uno spazio d'indirizzi. Un processo Win32 è dotato al momento della sua creazione di uno heap predefinito (*default heap*): poiché molte applicazioni Win32 sono multithread è necessario sincronizzare gli accessi allo heap per evitare che le sue strutture dati siano danneggiate da accessi concorrenti di thread diversi. Il vantaggio dello heap è che può essere utilizzato per creare allocazioni anche solo di 1 byte, poiché le pagine di memoria sottostanti sono già state salvate. Sfortunatamente, la memoria heap non può essere condivisa o contrassegnata come di sola lettura, poiché tutte le allocazioni dello heap condividono le stesse pagine. Tuttavia, utilizzando `HeapCreate()`, un programmatore può creare il proprio heap, che può essere contrassegnato come di sola lettura con `HeapProtect()`, creato come heap eseguibile o anche assegnato su un nodo numa specifico.

Win32 mette a disposizione le seguenti funzioni per la gestione e l'assegnazione di uno heap privato: `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()` e `HeapDestroy()`; l'api Win32 fornisce anche le funzioni `HeapLock()` e `HeapUnlock()` che garantiscono a un thread l'accesso esclusivo a uno heap. Si noti che queste funzioni eseguono solo la sincronizzazione e non "bloccano" realmente le pagine proteggendole da codice maligno o bug che ignorino il livello heap.

Lo heap originale Win32 era ottimizzato per un uso efficiente dello spazio. Ciò ha portato a notevoli problemi con la frammentazione dello spazio degli indirizzi nei programmi dei server di grandi dimensioni in esecuzione per lunghi periodi di tempo. Un nuovo progetto di heap a bassa frammentazione (lfh), introdotto in Windows xp, ha notevolmente ridotto il problema della frammentazione. Il gestore di heap di Windows 7 è in grado di attivare automaticamente lfh quando conviene. Inoltre, lo heap è un obiettivo primario degli aggressori che sfruttano vulnerabilità per attacchi double-free, use-after-free e altri attacchi correlati alla corruzione della memoria. Ogni versione di Windows, inclusa Windows 10, ha aggiunto più casualità, entropia e protezioni di sicurezza per impedire agli autori di attacchi di indovinare l'ordine, le dimensioni, la posizione e il contenuto delle allocazioni dello heap.

21.7.5.4 Memoria locale dei thread

Un quarto modo in cui un'applicazione può usare la memoria è rappresentato dal meccanismo di memorizzazione locale dei thread (tls). Di solito le funzioni che fanno uso di dati statici o globali non operano correttamente in un ambiente multithread: la funzione run-time `strtok()` del linguaggio C, per esempio, impiega una variabile statica per tener traccia della posizione corrente durante la scansione di una sequenza di caratteri; affinché due thread concorrenti possano eseguire correttamente `strtok()` devono tenere distinte le variabili che mantengono la posizione corrente. tls fornisce un meccanismo per mantenere le istanze delle variabili globali per la funzione che viene eseguita, ma non condivise con alcun altro thread.

Il meccanismo tls può essere messo in atto sia tramite metodi dinamici sia tramite metodi statici. Il metodo dinamico è illustrato nella Figura 21.13. Esso alloca spazio dello heap globale e lo collega al blocco di ambiente del thread che Windows assegna a ogni thread

in modalità utente. Il teb è facilmente accessibile da ogni thread e viene utilizzato non solo per tls, ma per tutte le informazioni di stato in modalità utente specifiche dei thread.

```
// riserva spazio per una variabile
DWORD dwVarIndex = TlsAlloc();
// assicura che uno spazio sia disponibile
if (dwVarIndex == TLS_OUT_OF_INDEXES)
    return;
// gli assegna il valore 10
TlsSetValue(dwVarIndex, (LPVOID)10);
// ottieni il valore
DWORD dwVar = (DWORD)(DWORD_PTR)TlsGetValue(dwVarIndex);
// rilascia l'indice
TlsFree(dwVarIndex);
```

Figura 21.13 Codice per l'assegnazione dinamica di memoria locale a un thread.

Per usare in un thread una variabile statica locale, in modo da assicurare che ogni thread ne possieda una copia privata, un'applicazione dovrebbe dichiarare la variabile come segue:

```
-declspec(thread) DWORD cur_pos = 0;
```

21.7.5.5 Memoria AWE

Un ultimo modo per permettere alle applicazioni di utilizzare la memoria sfrutta la funzionalità awe (Address Windowing Extension). Questo meccanismo consente a uno sviluppatore di richiedere direttamente pagine di ram fisiche libere al gestore della memoria (tramite la `AllocateUserPhysicalPages()`) e successivamente di collocare la memoria virtuale sulle pagine fisiche utilizzando `VirtualAlloc()`. Richiedendo varie regioni della memoria fisica (incluso il supporto scatter-gather), un'applicazione in modalità utente può accedere a una quantità maggiore di memoria fisica rispetto allo spazio degli indirizzi virtuali; ciò è utile su sistemi a 32 bit, che possono così utilizzare più di 4 gb di ram. Inoltre, l'applicazione può ignorare gli algoritmi di caching, di paging e di colorazione del gestore della memoria. Come nel caso di ums, awe può quindi offrire a determinate applicazioni un modo per ottenere prestazioni o personalizzazioni oltre a quelle offerte da Windows per impostazione predefinita. sql Server, per esempio, utilizza la memoria awe.

21.8 Sommario

- Il sistema operativo Windows è stato concepito da Microsoft come sistema operativo portabile ed estendibile, capace di trarre vantaggio dalle innovazioni nelle tecniche e nell'hardware.
- Windows gestisce l'elaborazione parallela simmetrica e supporta diversi ambienti operativi, tra cui processori sia a 32 sia a 64 bit e i calcolatori numa.
- L'uso degli oggetti del kernel per fornire i servizi fondamentali e la gestione del modello di elaborazione client-server permette a Windows di offrire un'ampia gamma di ambienti d'applicazione.
- Windows fornisce la gestione della memoria virtuale, servizi integrati di caching e scheduling con prelazione.
- Per proteggere i dati dell'utente e garantire l'integrità del programma, Windows supporta meccanismi di sicurezza elaborati e mitigazione degli exploit e sfrutta la virtualizzazione dell'hardware.
- Windows è eseguibile su un'ampia varietà di calcolatori; in questo modo gli utenti possono scegliere e aggiornare i propri calcolatori, e i dispositivi che li compongono, in funzione delle loro disponibilità finanziarie e delle prestazioni richieste senza dover modificare le applicazioni eseguite.
- Includendo le funzionalità di internazionalizzazione, Windows può essere eseguito in diversi paesi e in molte lingue.
- Windows dispone di sofisticati algoritmi di scheduling e di gestione della memoria che offrono prestazioni e scalabilità.
- Le versioni recenti di Windows hanno aggiunto funzionalità di gestione dell'alimentazione e di sospensione e riattivazione rapida, e hanno ridotto l'utilizzo delle risorse in diverse aree per diventare più utili su sistemi mobili come telefoni e tablet.
- Il gestore di volumi di Windows e il file system NTFS forniscono un insieme sofisticato di funzionalità per sistemi desktop e server.
- L'ambiente di programmazione api Win32 è ricco di funzionalità ed è in continua espansione, in modo da consentire ai programmati di utilizzare tutte le funzionalità di Windows nei loro programmi.

Esercizi di ripasso

21.1 Che tipo di sistema operativo è Windows? Descrivete due delle sue caratteristiche principali.

21.2 Elencate gli obiettivi di progetto di Windows. Descrivetene due in dettaglio.

21.3 Descrivete il processo di avvio di un sistema Windows.

21.4 Descrivete i tre livelli principali dell'architettura del kernel di Windows.

21.5 Che lavoro svolge il gestore degli oggetti?

21.6 Che tipi di servizio offre il gestore dei processi?

21.7 Che cos'è una chiamata di procedura locale?

21.8 Quali sono le responsabilità del gestore dell'i/o?

21.9 Quali tipologie di networking sono supportate da Windows? Come sono implementati i protocolli di trasporto da Windows? Descrivete due protocolli di rete.

21.10 Descrivete l'organizzazione dello spazio dei nomi di ntfs.

21.11 Come tratta le strutture dati ntfs? Come viene ripristinato ntfs dopo un crash di sistema? Che cosa viene garantito dopo che è avvenuto un ripristino?

21.12 Come alloca la memoria utente Windows?

21.13 Descrivete alcuni dei modi in cui un'applicazione utilizza la memoria per mezzo della api Win32.

Esercizi

- 21.14 Quando e perché si dovrebbe usare la funzionalità della procedura di chiamata differita in Windows?
- 21.15 Che cos'è un handle e come fa un processo a ottenerne uno?
- 21.16 Descrivete il funzionamento del gestore della memoria virtuale. In che modo il gestore della mm può migliorare le prestazioni?
- 21.17 Descrivete un'applicazione utile della funzionalità che vietи l'accesso alle pagine fornita da Windows.
- 21.18 Descrivete le tre tecniche previste per trasmettere i dati in una procedura di chiamata locale. Quali differenti contesti portano all'applicazione dell'una o dell'altra tecnica di scambio dei messaggi?
- 21.19 Da quale entità è gestita la cache di Windows? E con quali modalità?
- 21.20 Per quali aspetti la struttura della directory ntfs differisce da quella adottata dai sistemi operativi unix?
- 21.21 Che cos'è un processo e in che modo il sistema Windows lo gestisce?
- 21.22 Che cosa sono le "fibre" disponibili in Windows? In che cosa differiscono dai thread?
- 21.23 In che cosa lo scheduling in modalità utente (ums) di Windows 7 differisce dalle fibre? Individuate alcuni compromessi tra fibre e ums.
- 21.24 ums ritiene che il thread sia composto da due parti, un ut e un kt. In che cosa potrebbe essere utile consentire agli ut di continuare l'esecuzione in parallelo con i loro kt?
- 21.25 Quali sono i vantaggi e gli svantaggi in termini di prestazioni nel permettere ai kt e agli ut di essere eseguiti su diversi processori?
- 21.26 Perché l'auto-mappa occupa grandi quantità di spazio degli indirizzi virtuali, ma non occupa memoria virtuale aggiuntiva?
- 21.27 In che modo l'auto-mappa facilita al gestore della mm lo spostamento delle pagine della tabella delle pagine da e verso il disco? Dove sono conservate le pagine della tabella delle pagine sul disco?
- 21.28 Quando un sistema Windows si sospende, il sistema viene spento. Supponiamo di sostituire la cpu o aumentare la quantità di ram su un sistema che è stato sospeso. Il sistema potrebbe funzionare? Motivate la risposta.
- 21.29 Fornite un esempio che mostra come l'uso di un conteggio di sospensione è utile nella sospensione e nella ripresa dei thread in Windows.

APPENDICE A

Prospettiva storica

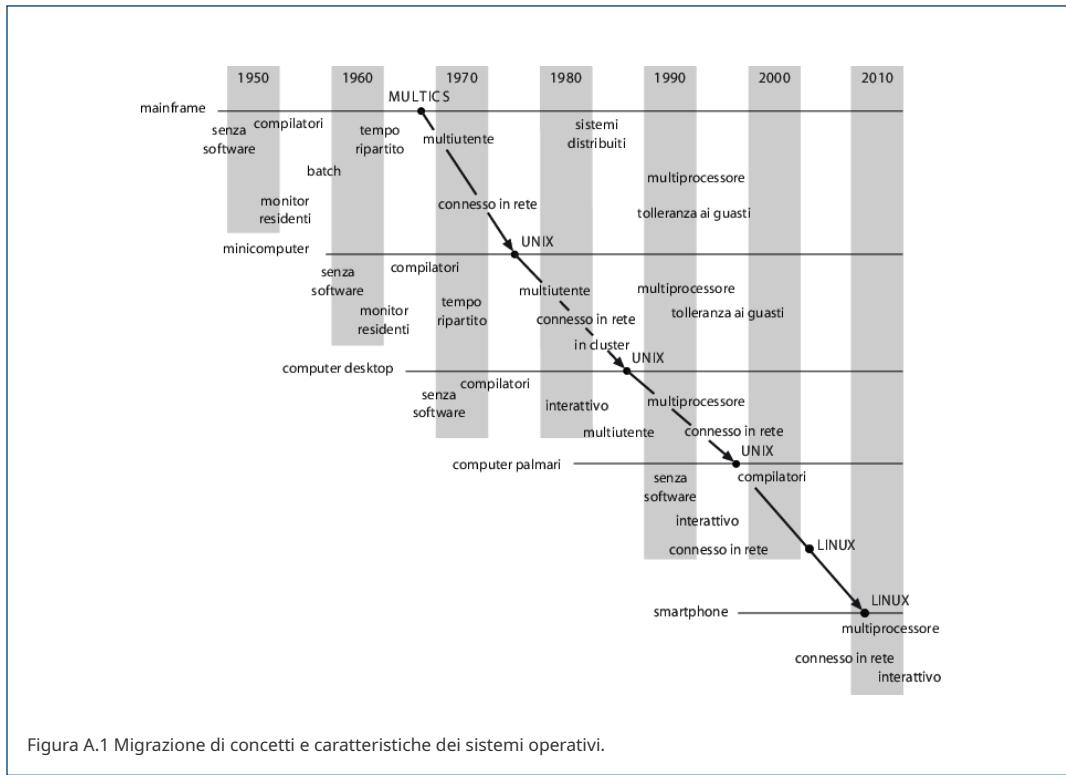
Ora che siamo in grado di comprendere i concetti alla base dei sistemi operativi (scheduling della cpu, gestione della memoria, processi e così via) è possibile esaminare come questi stessi concetti siano stati applicati in più sistemi operativi molto influenti del passato. Alcuni di essi, come il sistema xds-940 o il sistema the, sono stati pezzi unici, altri, come l'os/360, hanno avuto una grande diffusione. L'ordine di presentazione è stato scelto in modo da evidenziare analogie e differenze tra i sistemi, quindi non è strettamente cronologico o ordinato per importanza. Chiunque si occupi di sistemi operativi dovrebbe avere confidenza con tutti questi sistemi.

Nelle note bibliografiche alla fine di questa appendice sono inclusi riferimenti a ulteriori letture. I documenti scritti dai progettisti dei sistemi sono importanti sia per il contenuto tecnico sia per lo stile e il gusto.

A.1 Migrazione delle caratteristiche

Una ragione per studiare le prime architetture e i primi sistemi operativi risiede nel fatto che le funzionalità che una volta erano disponibili solamente su grandi sistemi sono ora riscontrabili su sistemi molto piccoli. In effetti, uno studio dei sistemi operativi per mainframe e per microcomputer mostra che molte funzionalità una volta disponibili soltanto su mainframe sono state adottate dai microcomputer. Gli stessi concetti di un sistema operativo sono quindi adatti per diverse classi di calcolatori: mainframe, minicomputer, microcomputer e dispositivi palmari. Per capire i moderni sistemi operativi è allora necessario conoscere il tema della migrazione delle caratteristiche e la lunga storia di molte funzionalità di un sistema operativo, come illustrato dalla Figura A.1.

Un buon esempio di migrazione delle caratteristiche inizia dal sistema operativo **multics** (*multiplexed information and computing services*). multics fu sviluppato tra il 1965 e il 1970 dal mit come una **utilità** per il calcolo e veniva eseguito su un mainframe (il ge 645) grande e complesso. Molte delle idee sviluppate per multics vennero in seguito utilizzate dai Bell Laboratories (uno dei partner iniziali nello sviluppo di multics) per il progetto di unix. Il sistema operativo unix fu progettato intorno al 1970 per un minicomputer pdp-11. Intorno al 1980 le caratteristiche di unix diventarono le basi per i sistemi operativi unix-like utilizzati sui microcomputer. Le stesse funzionalità sono presenti ora in diversi sistemi operativi recenti per microcomputer, come Microsoft Windows, e Macos. Linux include alcune di queste stesse funzionalità, che possono oggi essere impiegate da dispositivi mobili.



A.2 Primi sistemi

Rivolgiamo ora la nostra attenzione a una rassegna storica dei primi sistemi elaborativi. Va notato che la storia della computazione inizia molto prima dei “computer”, con i telai e le macchine calcolatrici. Ciononostante, inizieremo la nostra analisi dai computer del ventesimo secolo.

Prima degli anni '40, i sistemi elaborativi erano progettati e implementati per eseguire compiti specifici e prefissati. Modificare uno di questi compiti richiedeva un grande sforzo e del lavoro manuale. Tutto questo cambiò negli anni '40, quando Alan Turing, John von Neumann e altri colleghi lavorarono, sia insieme che separatamente, all'idea di un **computer a programma memorizzato (stored program computer)** di uso più generale. Una tale macchina doveva memorizzare sia un programma sia i dati da elaborare e il programma memorizzato doveva fornire istruzioni per il trattamento dei dati.

Questa fondamentale idea permise la rapida nascita di diversi computer general purpose, ma molta della storia di queste macchine è circondata dal mistero e dai segreti del loro sviluppo durante la seconda guerra mondiale. È probabile che il primo computer general purpose a programma memorizzato funzionante fu il Manchester Mark 1, sperimentato con successo nel 1949. Il primo computer commerciale fu il suo discendente Ferranti Mark 1, messo in vendita nel 1951.

I primi calcolatori erano macchine molto grandi che si gestivano da una console dalla quale il programmatore, che era anche l'operatore del sistema elaborativo, poteva scrivere e gestire un programma. Il programma veniva dapprima caricato manualmente nella memoria impiegando gli interruttori del pannello frontale, un'istruzione alla volta, oppure inserendo un nastro perforato o un pacco di schede perforate; quindi si premavano i pulsanti appropriati per impostare l'indirizzo iniziale e per avviare l'esecuzione. Durante l'esecuzione del programma, il programmatore/operatore poteva controllarne l'esecuzione per mezzo di spie situate sulla console. Se si riscontravano errori, il programmatore poteva fermare il programma, esaminare il contenuto della memoria e dei registri e mettere a punto il programma direttamente dalla console. L'esito veniva stampato, oppure trascritto perforando appositi nastri o schede di carta per essere stampato successivamente.

A.2.1 Sistemi di elaborazione dedicati

Con il passare del tempo furono sviluppati altri programmi, e altro hardware. I lettori di schede, le stampanti e i nastri magnetici divennero d'uso comune. Per facilitare la programmazione furono progettati assemblatori, caricatori e linker, e create librerie di funzioni comuni che si potevano copiare in un programma senza dover essere riscritte, consentendo il riutilizzo di codice già scritto.

Le procedure che eseguivano i/o assunsero un'importanza particolare. Ogni nuovo dispositivo di i/o aveva le sue caratteristiche che richiedevano un'attenta programmazione. Per ogni dispositivo di i/o si scriveva una procedura speciale, chiamata driver di dispositivo, capace di interagire con buffer, flag, registri, bit di controllo e bit di stato di ogni specifico dispositivo. Un compito semplice, come la lettura di un carattere da un lettore di nastri, poteva implicare complesse sequenze di operazioni specifiche del dispositivo. Invece di dover scrivere ogni volta il codice necessario, s'impiegava semplicemente il driver di dispositivo della libreria.

In seguito apparvero i compilatori per i linguaggi fortran, cobol e altri linguaggi, che resero molto più semplice la programmazione, ma più complesso il funzionamento del calcolatore. Per preparare l'esecuzione di un programma scritto in fortran, il programmatore doveva prima caricare nel calcolatore il compilatore del fortran. Il compilatore normalmente si teneva su un nastro magnetico, che quindi doveva essere montato nell'unità a nastri. Il programma veniva letto dal lettore di schede e scritto in un altro nastro. Il compilatore del fortran produceva in uscita un programma in linguaggio assembler che, a sua volta, doveva essere assemblato; quest'operazione richiedeva il montaggio di un altro nastro in cui si trovava l'assemblatore. Il risultato prodotto dall'assemblatore si doveva linkare alle procedure della libreria. Infine, si poteva leggere ed eseguire la forma binaria del programma. Si effettuava il caricamento nella memoria, la correzione e la messa a punto dalla console e infine l'esecuzione.

Il tempo di preparazione necessario per eseguire un job poteva essere molto elevato; ciascun job consisteva di molti passi distinti:

1. il caricamento del nastro con il compilatore del linguaggio fortran;
2. l'esecuzione del compilatore;
3. la rimozione del nastro con il compilatore;
4. il caricamento del nastro dell'assemblatore;
5. l'esecuzione dell'assemblatore;
6. la rimozione del nastro dell'assemblatore;
7. il caricamento del programma oggetto;
8. l'esecuzione del programma oggetto.

Se durante una di queste fasi si riscontrava un errore, il programmatore/operatore doveva ricominciare tutto da capo. Ogni fase poteva implicare il caricamento e la rimozione di nastri magnetici, nastri di carta e schede perforate.

Il tempo necessario per la preparazione di un job era un vero problema: durante il montaggio dei nastri e l'attività del programmatore alla console la cpu restava inattiva. Si ricordi che i primi calcolatori disponibili erano molto pochi e quei pochi erano molto costosi; un calcolatore poteva costare milioni di dollari, anche senza considerare i costi del lavoro dei programmatori, della corrente elettrica, del raffreddamento e così via; il tempo d'uso di un calcolatore assumeva quindi un valore molto elevato e per ammortizzare i costi d'investimento era necessaria una **utilizzazione** elevata.

A.2.2 Sistemi di elaborazione condivisi

Per ottimizzare la produttività di un calcolatore si trovò una soluzione su due fronti. Innanzitutto si ricorreva a un operatore professionista, e il programmatore quindi non doveva più lavorare alla macchina. Di conseguenza, non appena si terminava un job, l'operatore poteva iniziare quello successivo; poiché l'operatore aveva più esperienza nel montaggio dei nastri di quanta ne avesse un programmatore, anche il tempo di preparazione si ridusse. Il programmatore doveva fornire le schede o i nastri necessari insieme con una breve descrizione del modo d'esecuzione del programma. Naturalmente, l'operatore non poteva effettuare il debug di un

programma non corretto dalla console, giacché che non era in grado di capire il programma stesso. Perciò, quando si presentavano errori nel programma, si stampava un'immagine dell'intero contenuto della memoria e dei registri e il programmatore doveva fare le correzioni basandosi sulle informazioni così ottenute. In questo modo l'operatore poteva continuare immediatamente con il lavoro successivo, ma al programmatore restava un compito di correzione più difficile.

Un secondo aspetto era che i job che avevano requisiti simili venivano raggruppati in lotti (*batch*) ed eseguiti nel calcolatore come un unico gruppo per ridurre il tempo di preparazione. Si supponga per esempio che l'operatore ricevesse un job scritto in fortran, uno scritto in cobol e uno scritto ancora in fortran. Per eseguirli in quell'ordine avrebbe dovuto predisporre la macchina prima per il fortran, quindi per il cobol e poi di nuovo per il fortran, tutto ciò richiedeva il caricamento dei nastri compilatori, e così via. Se, invece, eseguiva i due job scritti in fortran come un unico batch, poteva predisporre il calcolatore una volta sola per il linguaggio fortran, risparmiando tempo.

Ma c'erano ancora alcuni problemi. Per esempio, l'operatore doveva stabilire, osservando la console, quando un programma era terminato e *perché*, ossia se era terminato in maniera normale o anomala, quindi, se era necessario, doveva stampare l'immagine del contenuto della memoria, caricare il job successivo nel lettore di schede o nel lettore del nastro e infine riavviare il calcolatore. Durante il passaggio da un job a un altro, la cpu restava inattiva. Per eliminare anche questo tempo morto, fu sviluppato il cosiddetto **sequenzializzatore automatico dei job**; questa tecnica permise la creazione dei primi rudimentali sistemi operativi. Fu creato un piccolo programma, chiamato **monitor residente** per il trasferimento automatico del controllo dell'elaborazione da un job a quello successivo (Figura A.2). Questo monitor si trovava sempre, cioè *risiedeva*, nella memoria.

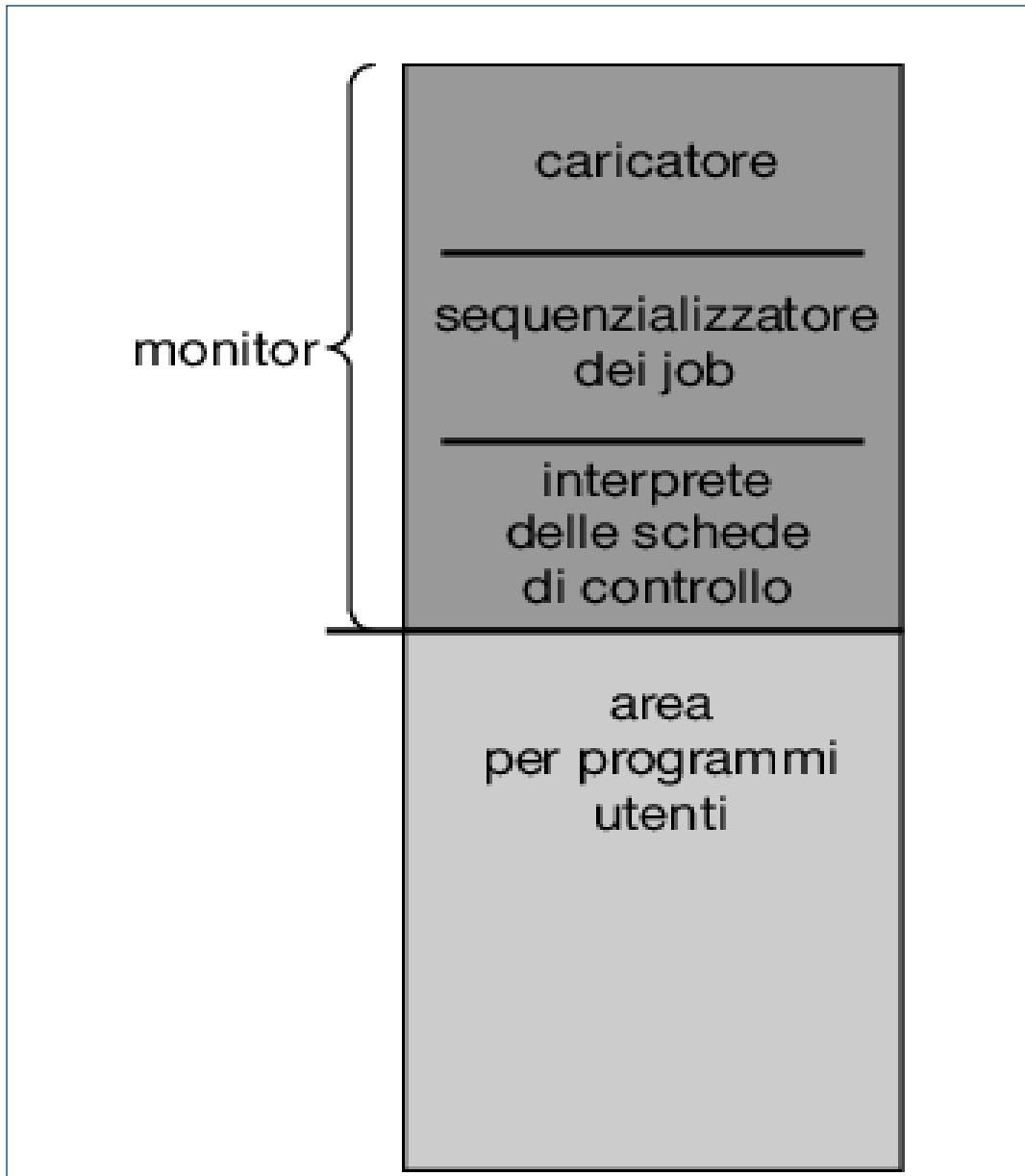


Figura A.2 Configurazione della memoria per un monitor residente.

Al momento dell'accensione del calcolatore, s'invocava il monitor residente che trasferiva il controllo a un programma. Terminato il programma, il controllo tornava al monitor residente che lo passava al programma successivo; in questo modo il monitor residente sequenzializzava automaticamente i programmi e i job.

Ci si può domandare come facesse il monitor residente a sapere quale programma doveva eseguire. In precedenza all'operatore si consegnava una breve descrizione dei programmi da eseguire e dei relativi dati. Per passare queste informazioni direttamente al monitor furono adottate le **schede di controllo**. L'idea era semplice. Oltre al programma o ai dati, il programmatore inseriva le schede di controllo, che contenevano le direttive che indicavano al monitor residente quale programma doveva eseguire. Per esempio, un normale programma utente avrebbe potuto richiedere l'esecuzione di uno di questi tre programmi: compilatore del linguaggio fortran (FTN), assemblatore (ASM), oppure programma utente (RUN). Per ciascuno di questi programmi si può impiegare una scheda di controllo diversa:

- \$FTN — esegui il compilatore del linguaggio fortran;
- \$ASM — esegui l'assemblatore;
- \$RUN — esegui il programma utente.

Queste schede indicavano al monitor residente quali erano i programmi da eseguire. Per definire i limiti di ogni job si potevano adoperare altre due schede di controllo:

- \$JOB — prima scheda di un job;
- \$END — ultima scheda di un job.

Queste due schede avrebbero potuto essere utili per contabilizzare le risorse di macchina impiegate dal programmatore.

Per definire il nome del job, il codice d'addebito e così via si potevano usare opportuni parametri. Per altre funzioni, come la richiesta di caricamento o rimozione di un nastro da parte dell'operatore, si potevano definire altre schede di controllo.

Le schede di controllo dovevano essere distinte dalle schede dei dati o del programma. Tale distinzione si otteneva per mezzo di un carattere o di un simbolo speciale riportato sulla scheda stessa. Parecchi sistemi identificavano una scheda di controllo riportando nella prima colonna il carattere \$. Altri usano un simbolo diverso: il Job Control Language (jcl) della ibm, per esempio, usava due barre (//) inserite nelle prime due colonne. Nella Figura A.3 è riportato un esempio di preparazione del pacco di schede per un sistema batch.

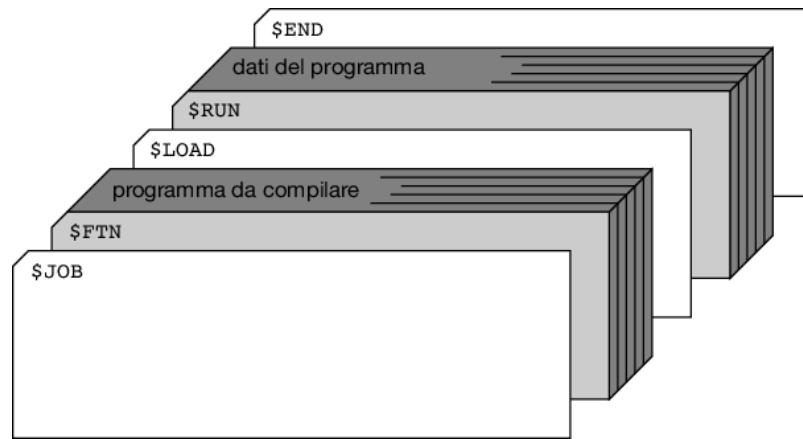


Figura A.3 Pacco di schede per un sistema batch.

Un monitor residente era quindi composto da diversi elementi chiaramente identificabili.

- L'**interprete delle schede di controllo**, responsabile della lettura e dell'esecuzione delle istruzioni contenute sulle schede stesse.
- Il **caricatore**, invocato dall'interprete delle schede di controllo per caricare nella memoria opportuni programmi di sistema e programmi applicativi.
- I **driver dei dispositivi**, usati sia dall'interprete delle schede di controllo sia dal caricatore per l'esecuzione delle operazioni di i/o. Spesso i programmi di sistema e quelli applicativi si servivano di questi stessi driver di dispositivi, che garantivano un funzionamento corretto e un risparmio di spazio nella memoria e di tempo di programmazione.

Questi sistemi batch lavoravano piuttosto bene. Il monitor residente garantiva la sequenzializzazione automatica dei lavori, così com'era indicata dalle schede di controllo. Quando una scheda di controllo indicava che si doveva eseguire un programma, il monitor caricava nella memoria il programma e gli trasferiva il controllo. Terminato il programma, il controllo tornava al monitor, il quale leggeva la scheda di controllo successiva, caricava il nuovo programma e così via. Si ripeteva il ciclo fino a che tutte le schede di controllo del job in corso erano state interpretate. Quindi, il monitor continuava automaticamente a lavorare, passando al job successivo.

Il passaggio ai sistemi batch con sequenzializzatore automatico dei lavori si è verificato per migliorare le prestazioni. Il problema, abbastanza semplice, consisteva nella lentezza dell'intervento umano, che veniva pertanto sostituito da programmi di un sistema operativo: il sequenzializzatore automatico dei lavori, infatti, eliminava la perdita di tempo derivante dalla preparazione e dalla sequenzializzazione dei lavori da parte dell'operatore.

Com'è stato evidenziato, anche con questo sistema, tuttavia, la cpu spesso rimaneva inattiva. Il problema a questo punto riguardava i dispositivi meccanici di i/o, intrinsecamente più lenti dei dispositivi elettronici. I tempi caratteristici di una cpu, anche se lenta, sono misurabili in microsecondi. Quindi, mentre questa eseguiva migliaia di istruzioni al secondo, un lettore di schede veloce, invece, poteva leggere 1200 schede al minuto, vale a dire 20 al secondo. Di conseguenza, la differenza di velocità tra la cpu e i suoi dispositivi di i/o poteva essere di tre o più ordini di grandezza. Con il tempo, i perfezionamenti tecnologici hanno portato allo sviluppo di dispositivi di i/o più veloci, ma contemporaneamente era aumentata anche la velocità delle cpu, quindi il problema non solo non si era risolto, ma si era aggravato.

A.2.3 i/o sovrapposto

Una soluzione diffusa al problema dell'i/o prevedeva la sostituzione dei lettori di schede e delle stampanti con unità a nastro magnetico. Alla fine degli anni Cinquanta e all'inizio degli anni Sessanta la maggior parte dei sistemi elaborativi era costituita da sistemi batch che utilizzavano come dispositivi di lettura i lettori di schede e come dispositivi di scrittura le stampanti o i perforatori di schede. Tuttavia, anziché far leggere le schede direttamente dalla cpu, si eseguiva un primo passaggio di copiatura delle schede in nastro magnetico. Quindi, quando un nastro era sufficientemente pieno, lo si rimuoveva e lo si portava al calcolatore. Quando una scheda veniva richiesta da un programma, si leggeva dal nastro il settore corrispondente. Analogamente, i risultati dell'elaborazione venivano inviati al nastro e il contenuto del nastro stesso veniva passato solo in una fase successiva alla stampante. I lettori di schede e le stampanti anziché dal calcolatore centrale erano gestiti fuori linea (*off-line*) (Figura A.4).

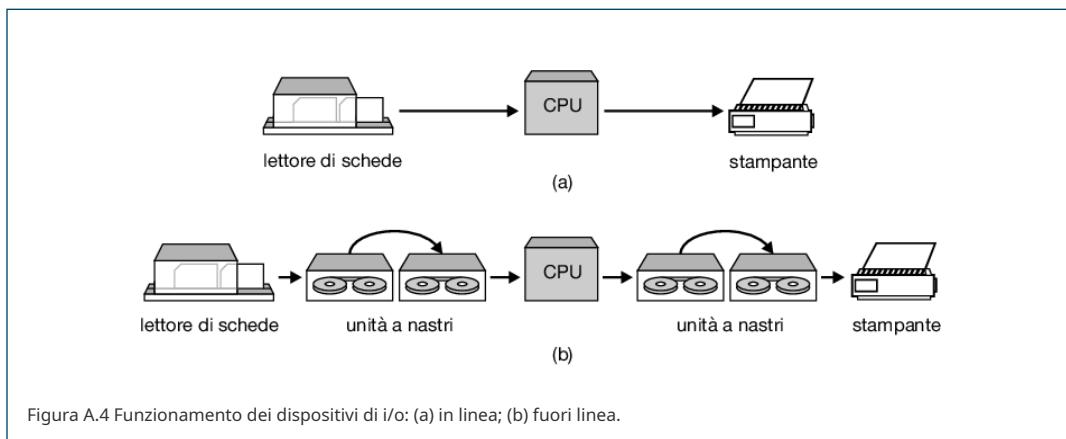


Figura A.4 Funzionamento dei dispositivi di i/o: (a) in linea; (b) fuori linea.

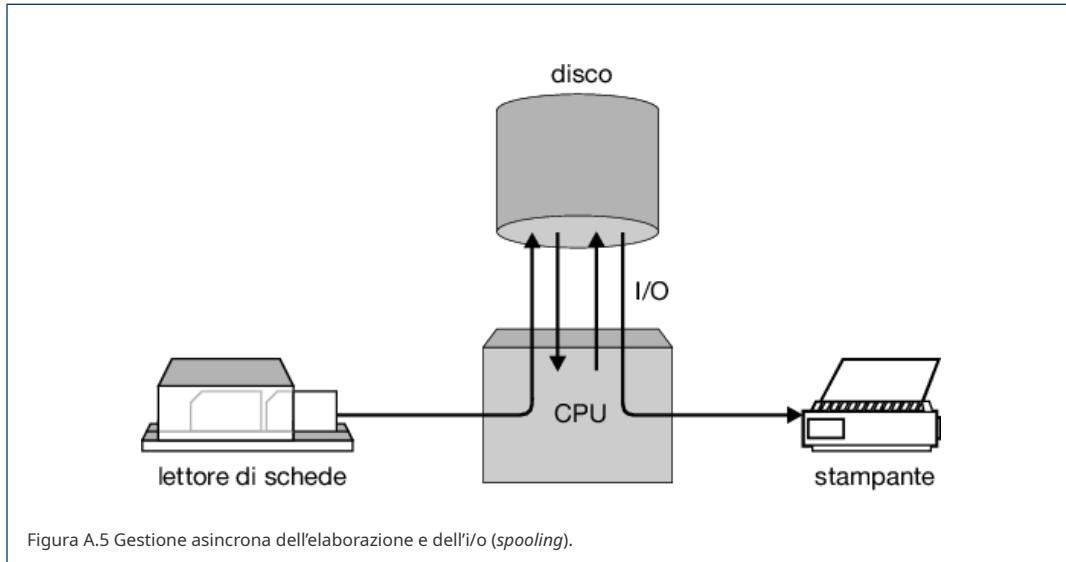
Il vantaggio principale dato da questo metodo consisteva nel fatto che il calcolatore principale non dipendeva più dalla velocità dei lettori di schede e delle stampanti, ma da quella delle unità a nastro magnetico, che, comunque, erano molto più veloci dei dispositivi di i/o precedenti. L'uso del nastro magnetico come i/o poteva essere applicato con qualsiasi dispositivo, si trattasse di lettori di schede, perforatori, plotter, nastri perforati o stampanti.

Il guadagno effettivo ottenibile derivava dalla possibilità di impiegare per una stessa cpu più sistemi lettore-nastro e nastro-stampante. Se la cpu è in grado di elaborare i dati in ingresso con una velocità doppia rispetto a quella con cui il lettore legge le schede, allora due lettori contemporaneamente attivi possono produrre una quantità di nastro sufficiente per tenere costantemente occupata la cpu. D'altra parte c'è uno svantaggio: ora si riscontra un ritardo maggiore nell'esecuzione di un particolare job d'elaborazione; prima occorre eseguirne la lettura nel nastro, poi bisogna aspettare che il nastro sia riempito completamente con altri job. Quando il nastro è pieno deve essere riavvolto, rimosso, portato manualmente al calcolatore e infine montato su un'unità a nastri libera. Naturalmente, tutto questo è ragionevole per i sistemi batch. Molti job simili si possono raggruppare in batch in un nastro, prima di portare quest'ultimo al calcolatore.

Per qualche tempo i job continuarono a essere preparati con le procedure off-line, ma ben presto nella maggior parte dei sistemi queste procedure furono sostituite; la maggior diffusione di sistemi con unità disco permise di perfezionare le operazioni off-line. Il problema nelle unità a nastro era dovuto al fatto che il lettore di schede non poteva scrivere a un'estremità del nastro mentre la cpu leggeva dall'altra estremità. Prima di poter essere riavvolto e letto, il nastro doveva essere completamente scritto, giacché si trattava di un **dispositivo ad accesso sequenziale**. I sistemi a disco hanno eliminato questo problema, essendo **dispositivi ad accesso diretto**. La

testina si sposta da una zona all'altra del disco, e può passare rapidamente dalla zona usata dal lettore per memorizzare il contenuto di nuove schede alla posizione richiesta dalla cpu per leggere la scheda successiva.

In un sistema a dischi le schede vengono trasferite direttamente dal lettore al disco, e la posizione delle schede viene registrata in una tabella del sistema operativo. Durante l'esecuzione di un job d'elaborazione, il sistema operativo soddisfa le richieste di lettura delle schede leggendo dal disco i dati corrispondenti alle schede richieste. Analogamente, quando il job richiede alla stampante di stampare una riga, quella riga viene copiata nella memoria del sistema e scritta nel disco. I risultati vengono effettivamente stampati quando il job è completato. Questo tipo di gestione asincrona dell'elaborazione e dell'i/o (Figura A.5) – il cosiddetto **spooling** (*simultaneous peripheral operation on line*) – adopera fondamentalmente il disco come un enorme mezzo di bufferizzazione, per leggere in anticipo il maggior numero possibile di dati dai dispositivi d'ingresso e per memorizzare i dati in uscita fino a che i dispositivi di emissione siano in grado di accettarli.



Lo *spooling* si usa anche per elaborare dati in postazioni remote. La cpu invia i dati a una stampante remota su linee di comunicazione, oppure accetta in ingresso un intero job proveniente da un lettore di schede remoto. Le operazioni remote si compiono alla loro propria velocità, senza l'intervento della cpu; la cpu deve semplicemente essere informata del completamento delle operazioni in modo da intervenire sul batch successivo di dati.

Con lo spooling si sovrappone l'i/o di un job con la computazione di altri job; anche in un sistema semplice è infatti possibile leggere i dati in ingresso di un job mentre si stampano i risultati di un altro, contemporaneamente si può eseguirne un terzo (o più), leggendo le sue "schede" dal disco e "stampando" i risultati ancora nel disco.

Lo spooling ha un effetto positivo sulle prestazioni del sistema: impiegando poco spazio del disco e alcune tabelle, la computazione di un job può essere sovrapposta all'i/o di altri job; quindi permette alla cpu e ai dispositivi di i/o di lavorare a velocità maggiori. Lo spooling conduce in modo naturale alla multiprogrammazione, che è il fondamento di tutti i moderni sistemi operativi.

A.3 Atlas

Il sistema operativo Atlas è stato progettato alla University of Manchester in Inghilterra fra la fine degli anni Cinquanta e l'inizio degli anni Sessanta. Molte sue caratteristiche di base, che a quel tempo erano originali, sono diventate componenti standard dei sistemi operativi moderni. I driver dei dispositivi costituivano la parte principale del sistema. Inoltre, le chiamate di sistema erano state aggiunte attraverso un insieme di istruzioni speciali, chiamate *extra codes*.

L'Atlas era un sistema operativo batch con spooling. Ciò permetteva al sistema di eseguire lo scheduling dei job d'elaborazione secondo la disponibilità dei dispositivi periferici, come unità a nastri magnetici, lettori di nastri perforati, perforatori di nastri, stampanti, lettori di schede o perforatori di schede.

La caratteristica più notevole di questo sistema operativo era la sua gestione della memoria. A quel tempo la **memoria a nuclei magnetici** era ancora nuova e costosa. In molti calcolatori, come l'ibm 650, si usava un tamburo per la memoria centrale. Anche nel sistema Atlas si usava un tamburo per la memoria centrale, ma come cache per il tamburo si adoperava una piccola quantità di memoria a nuclei magnetici. Per trasferire automaticamente le informazioni tra la memoria a nuclei magnetici e il tamburo si usava la paginazione su richiesta.

Il sistema operativo Atlas era eseguito su un calcolatore con parole di 48 bit. Gli indirizzi erano di 24 bit, ma erano codificati in decimali, il che permetteva di accedere a un milione di parole. A quel tempo era uno spazio d'indirizzi estremamente grande. La memoria fisica era costituita di un tamburo di 98 K parole e di 16 K parole di memoria a nuclei. La memoria era divisa in pagine di 512 parole, quindi forniva 32 pagine fisiche di memoria. Una memoria associativa di 32 registri realizzava l'associazione dagli indirizzi virtuali agli indirizzi fisici.

Se si verificava un page fault, si richiamava un algoritmo di sostituzione delle pagine. Si teneva sempre vuota una pagina fisica della memoria, perciò un trasferimento dal tamburo si poteva avviare immediatamente. L'algoritmo di sostituzione delle pagine tentava di predire il comportamento futuro dell'accesso alla memoria, basandosi sul comportamento passato. S'impostava un bit di riferimento per ciascuna pagina fisica a ogni accesso a tale pagina. I bit di riferimento venivano letti nella memoria ogni 1024 istruzioni. Gli ultimi 32 valori di questi bit si usavano per definire il tempo trascorso dal più recente riferimento (t_1) e l'intervallo di tempo trascorso tra gli ultimi due riferimenti (t_2). Si sceglievano le pagine per la sostituzione nel seguente ordine:

1. qualsiasi pagina con $t_1 > t_2 + 1$ si ritiene non sia più in uso e viene rimpiazzata;
2. se $t_1 \leq t_2$ per tutte le pagine, allora si sostituisce la pagina dove la differenza $t_2 - t_1$ è maggiore.

L'algoritmo di sostituzione delle pagine presuppone che i programmi accedano alla memoria in modo ciclico. Se il tempo trascorso tra gli ultimi due riferimenti è t_2 , il riferimento successivo è atteso dopo t_2 unità di tempo. Se non si verifica alcun riferimento ($t_1 > t_2$), si suppone che la pagina non sia più utilizzata e quindi viene sostituita. Se tutte le pagine sono ancora in uso, si sostituisce la pagina che non sarà più richiesta per il periodo di tempo più lungo. Si suppone che il riferimento successivo avvenga entro $t_2 - t_1$ unità di tempo.

A.4 XDS-940

Il sistema operativo xds-940 è stato progettato alla University of California di Berkeley all'inizio degli anni Sessanta. Come il sistema Atlas, impiegava la paginazione per la gestione della memoria; ma diversamente dall'Atlas l'xds-940 era un sistema time sharing.

La paginazione si usava solo per la rilocazione e non per la paginazione su richiesta. La memoria virtuale di ogni processo utente era di 16 K parole, mentre la memoria fisica conteneva 64 K parole, e le pagine erano ciascuna di 2 K parole. La tabella delle pagine era conservata in registri. Poiché la memoria fisica era più grande della memoria virtuale di un processo utente, parecchi processi potevano essere contemporaneamente nella memoria. Il numero degli utenti poteva essere aumentato condividendo le pagine, quando queste ultime contenevano codice rientrante di sola lettura. I processi erano conservati in un tamburo e venivano caricati nella memoria e da essa scaricati secondo le necessità.

Il sistema xds-940 fu costruito a partire da un xds-930 modificato. Le modifiche erano rappresentative di quelle apportate a un calcolatore semplice per permettere di scrivere adeguatamente un sistema operativo: fu aggiunto il duplice modo di funzionamento (di sistema e utente); alcune istruzioni, come quelle di i/o e d'arresto furono definite come privilegiate. Il tentativo di eseguire un'istruzione privilegiata in modalità utente avrebbe causato una trap per il sistema operativo.

All'insieme di istruzioni da eseguire in modalità utente fu aggiunta un'istruzione di chiamata di sistema. Quest'istruzione si usava per creare nuove risorse, come file, permettendo al sistema operativo di gestire le risorse fisiche. I file, per esempio, si registravano nel tamburo in blocchi di 256 parole. Per gestire i blocchi di tamburo liberi si usava una mappa di bit. Ogni file aveva un blocco indice con puntatore ai blocchi di dati effettivi. I blocchi indice erano concatenati tra loro.

Il sistema xds-940 forniva anche chiamate di sistema per permettere ai processi di creare, avviare, sospendere ed eliminare sottoprocessi. Un programmatore poteva costruire un sistema di processi. Processi separati potevano condividere la memoria per scopi di comunicazione e sincronizzazione. La creazione di processi definiva una struttura ad albero, dove un processo costituiva la radice e i suoi sottoprocessi i nodi sottostanti nell'albero. Ognuno dei sottoprocessi poteva, a sua volta, creare altri sottoprocessi.

A.5 THE

Il sistema operativo the è stato progettato alla Technische Hogeschool di Eindhoven in Olanda a metà degli anni Sessanta. Si tratta di un sistema batch che funzionava su un calcolatore olandese, l'el x8, con 32 K parole di 27 bit. Il sistema si fece notare soprattutto per il suo progetto pulito, in particolare per la sua struttura a strati e il suo uso di un insieme di processi concorrenti sincronizzati tramite semafori.

Tuttavia, diversamente dal sistema xds-940, l'insieme dei processi del sistema the era statico. Il sistema operativo stesso era stato progettato come un insieme di processi cooperanti. Inoltre, erano stati creati cinque processi utenti, i quali servivano come agenti attivi per la compilazione, l'esecuzione e la stampa di programmi utenti. Una volta terminato un job, il processo ritornava alla coda d'ingresso per selezionarne un altro.

Si usava un algoritmo di scheduling della cpu con priorità. Le priorità si ricalcolavano ogni 2 secondi ed erano inversamente proporzionali al tempo di cpu usato recentemente, vale a dire negli ultimi 8-10 secondi. Questo schema offriva una priorità più alta ai processi con prevalenza di i/o e ai processi nuovi.

La gestione della memoria era limitata dall'assenza di adeguato supporto nell'hardware sottostante. Tuttavia, poiché il sistema era limitato e i programmi utenti si potevano scrivere solo in Algol, s'impiegava uno schema di paginazione gestito dal sistema operativo. Il compilatore Algol generava automaticamente chiamate alle procedure di sistema, le quali assicuravano che le informazioni richieste si trovassero nella memoria, effettuando lo swapping se era necessario. La memoria ausiliaria era un tamburo di 512 K parole. Si adoperavano pagine di 512 parole, con un criterio di sostituzione delle pagine lru.

Un altro punto importante del sistema the era il controllo delle situazioni di stallo. Per evitare le situazioni di stallo si ricorreva all'algoritmo del banchiere.

Il sistema Venus è strettamente correlato al sistema the. Anche il progetto del sistema Venus era basato su una struttura a strati, che impiegava semafori per sincronizzare i processi. I livelli inferiori del progetto, però, erano realizzati in microcodice, sicché il sistema era molto più veloce. Per la gestione della memoria si usava uno schema di memoria paginata e segmentata; inoltre il sistema è stato progettato come sistema time sharing, anziché come sistema batch.

A.6 RC 4000

Il sistema rc 4000, come il sistema the, si fece notare soprattutto per i concetti applicati nella progettazione. È stato progettato alla fine degli anni Sessanta per il calcolatore danese rc 4000 dalla Regnecentralen, e in particolare da P. Brinch Hansen. L'obiettivo non era quello di progettare un sistema batch, oppure un sistema time sharing, o qualsiasi altro sistema specifico, ma quello di creare il nucleo, o kernel, di un sistema operativo, sul quale fosse possibile costruire un sistema operativo completo. Così, la struttura del sistema fu concepita a strati, e furono forniti solo i livelli inferiori, cioè il kernel.

Il kernel gestiva un insieme di processi concorrenti. Lo scheduler della cpu seguiva un criterio rr. Anche se i processi potevano condividere la memoria, il meccanismo principale di comunicazione e sincronizzazione era il **sistema di messaggi** fornito dal kernel. I processi potevano comunicare tra loro scambiandosi messaggi di dimensione fissa, 8 parole di lunghezza. Tutti i messaggi si memorizzavano in apposite sezioni della memoria (*buffer*) che facevano parte di un gruppo comune di sezioni. Quando uno di questi buffer per i messaggi non era più necessario, veniva restituita al gruppo comune.

A ogni processo era associata una **coda di messaggi**, che conteneva tutti i messaggi inviati a quel processo che non erano ancora stati ricevuti. I messaggi venivano rimossi dalla coda in ordine fifo. Il sistema disponeva di quattro operazioni primitive, che erano eseguite in modo atomico:

- `send-message (inreceiver, inmessage, outbuffer);`
- `wait-message (outsender, outmessage, outbuffer);`
- `send-answer (outresult, inmessage, inbuffer);`
- `wait-answer (outresult, outmessage, inbuffer).`

Le ultime due operazioni permettevano ai processi di scambiarsi più messaggi alla volta.

Queste primitive richiedevano che un processo servisse la sua coda di messaggi in ordine fifo, e che si bloccasse mentre altri processi stavano gestendo i suoi messaggi. Per rimuovere tali limitazioni, i progettisti fornirono altre due primitive di comunicazione, che permettevano a un processo di attendere l'arrivo del messaggio successivo oppure di rispondere e servire la sua coda in qualunque ordine:

- `wait-event (inprevious-buffer, outnext-buffer, outresult);`
- `get-event (outbuffer).`

Anche i dispositivi di i/o erano trattati come processi. I driver dei dispositivi erano composti di codice che convertiva i segnali d'interruzione e i registri dei dispositivi in messaggi; così un processo scriveva in un terminale inviandogli un messaggio. Il driver del dispositivo riceveva il messaggio ed emetteva il carattere al terminale. L'immissione di un carattere interrompeva il sistema e determinava l'attivazione del driver del dispositivo, il quale a sua volta creava un messaggio contenente il carattere immesso e lo inviava al processo in attesa.

A.7 CTSS

Il sistema ctss (*compatible time-sharing system*) è stato progettato al mit come sistema sperimentale time sharing, e realizzato su un calcolatore ibm 7090. ctss fece la sua apparizione nel 1961. Gestiva fino a 32 utenti interattivi. Gli utenti disponevano di un insieme di comandi interattivi, che permetteva loro di manipolare i file e di compilare ed eseguire programmi interagendo con il sistema tramite un terminale.

L'ibm 7090 aveva una memoria di 32 K parole di 36 bit. Il monitor impiegava 5 K parole, lasciando le altre 27 K parole agli utenti. Le immagini della memoria utente s'avvicendavano tra la memoria e un tamburo rapido. Per lo scheduling della cpu s'impiegava un algoritmo a code multiple con retroazione. Il quanto di tempo per il livello i era di $2 * i$ unità di tempo. Se un programma non terminava la propria sequenza di operazioni di cpu entro un quanto di tempo, veniva abbassato al livello successivo della coda, e otteneva un quanto di tempo doppio. Il programma che si trovava al livello massimo, con il quanto più corto, veniva eseguito per primo. Il livello iniziale di un programma era stabilito dalla sua dimensione in modo che il quanto di tempo fosse lungo almeno come il tempo di swap.

Il ctss ha avuto un notevole successo ed era in uso fino al 1972. Benché limitato, esso è riuscito a dimostrare che il time sharing era un modo d'elaborazione conveniente e pratico. Un risultato ottenuto dal ctss è stato quello di favorire lo sviluppo dei sistemi time sharing, tra i quali quello del multics.

A.8 MULTICS

Il sistema operativo multics è stato progettato al mit tra il 1965 e il 1970 come un'estensione naturale del ctss. Il ctss e altri tra i primi sistemi time sharing ebbero un tale successo che si sentì subito il desiderio di procedere rapidamente verso sistemi più grandi e perfezionati. Quando divennero disponibili calcolatori più grandi, i progettisti del ctss avviarono lo sviluppo di una **utilità** di elaborazione time sharing: il servizio elaborativo sarebbe stato fornito come l'energia elettrica; si sarebbero potuti collegare, attraverso linee telefoniche, grandi sistemi elaborativi a terminali di uffici e case di tutta una città. Il sistema operativo era un sistema time sharing continuamente in funzione con un vasto file system di programmi e dati condivisi.

Il multics è stato progettato da un gruppo del mit, il ge (che più tardi ha venduto il proprio dipartimento di informatica alla Honeywell), e dai Bell Laboratories (che uscirono dal progetto nel 1969). L'architettura del calcolatore ge 635 di base fu modificata in quella di un nuovo calcolatore chiamato ge 645, soprattutto per consentire la segmentazione paginata della memoria.

Un indirizzo virtuale era composto di un numero di segmento di 18 bit e di un offset di parola di 16 bit. I segmenti venivano quindi suddivisi in pagine di 1 K parole, per le quali si usava l'algoritmo di sostituzione con seconda chance.

Lo spazio d'indirizzi virtuali segmentato è stato unito al file system; ogni segmento era un file e ai segmenti si faceva riferimento tramite il nome del file corrispondente. Lo stesso file system era una struttura ad albero a più livelli, che permetteva agli utenti di creare le proprie strutture delle directory.

Come il ctss, il multics impiegava uno scheduling della cpu a code multiple con retroazione. La protezione era garantita da una lista di controllo degli accessi associata a ogni file e da un insieme di anelli di protezione per i processi in esecuzione. Il sistema, che è stato scritto quasi completamente in pl/I, comprendeva circa 300.000 righe di codice. È stato esteso a un sistema multiprocessore che permetteva di sospendere dal servizio una cpu per motivi di manutenzione mentre il sistema continuava la sua esecuzione.

A.9 OS/360 di IBM

La più lunga linea di sviluppo dei sistemi operativi è indubbiamente quella dei calcolatori ibm. I primi calcolatori ibm, come l'ibm 7090 e l'ibm 7094, sono i primi esempi dello sviluppo delle procedure di i/o comuni, seguite da un monitor residente, istruzioni privilegiate, protezione della memoria e semplice elaborazione batch. Questi sistemi sono stati sviluppati separatamente, spesso da siti indipendenti. Il risultato è stato che l'ibm si è trovata di fronte a molti calcolatori diversi, con linguaggi diversi e programmi di sistema diversi.

L'ibm/360, che fece la sua comparsa nella metà degli anni Sessanta, fu progettato per modificare tale situazione. Il suo progetto ([Mealy et al. 1966]) prevedeva una famiglia di calcolatori che si estendeva su una gamma completa che andava da piccole macchine commerciali fino a grandi macchine scientifiche. Per questi sistemi sarebbe stato necessario un solo insieme di programmi; tutti questi sistemi impiegavano lo stesso sistema operativo: l'os/360. Quest'orientamento doveva ridurre i problemi di manutenzione dell'ibm, e permettere agli utenti di spostare liberamente programmi e applicazioni tra i sistemi ibm.

Sfortunatamente con l'os/360 s'è tentato di fare tutto per tutti, con il risultato di non riuscire a svolgere particolarmente bene alcun compito. Il file system comprendeva un campo di tipi che definiva il tipo di ciascun file, ed erano definiti diversi tipi di file con elementi (*record*) a lunghezza fissa o a lunghezza variabile, così come file a blocchi o non a blocchi. Si usava l'allocazione contigua, quindi l'utente doveva predire la dimensione di ogni file impiegato per contenere gli output delle elaborazioni. Il linguaggio Job Control Language (jcl) prevedeva parametri per ogni possibile opzione, diventando incomprensibile all'utente medio.

Le procedure di gestione della memoria erano ostacolate dall'architettura. Sebbene si adoperasse un modo d'indirizzamento con registro di base, il programma poteva accedere al registro base e modificarlo, perciò la cpu generava indirizzi assoluti. Questa situazione ha impedito la rilocazione dinamica: l'associazione degli indirizzi del programma agli indirizzi della memoria fisica si eseguiva nella fase di caricamento. Di questo sistema operativo sono state prodotte due versioni: l'os/mft impiegava regioni fisse, mentre l'os/mvt impiegava regioni variabili.

Il sistema fu scritto in linguaggio assembler da migliaia di programmatori, quindi le righe del codice risultante erano milioni. Lo stesso sistema operativo richiedeva grandi quantità di memoria per il proprio codice e le proprie tabelle. L'overhead del sistema operativo spesso richiedeva metà dei cicli totali della cpu. Con gli anni sono uscite nuove versioni mediante le quali sono state aggiunte nuove caratteristiche e sono stati corretti errori. Tuttavia, la correzione di un errore spesso ne causava un altro in una parte remota del sistema, perciò il numero degli errori noti del sistema era pressoché costante.

Con il passaggio all'architettura ibm 370, all'os/360 è stata aggiunta la memoria virtuale. L'architettura sottostante forniva una memoria virtuale con segmentazione paginata. Le nuove versioni del sistema operativo impiegavano quest'architettura in modi diversi. L'os/vs1 creava un grande spazio d'indirizzi virtuali e in quella memoria virtuale eseguiva l'os/mft. Quindi lo stesso sistema operativo era paginato, così come i programmi utenti. L'os/vs2 Release 1 eseguiva l'os/mvt nella memoria virtuale. Infine, l'os/vs2 Release 2, che ora si chiama mvs, fornisce a ogni utente la sua memoria virtuale.

L'mvs è ancora fondamentalmente un sistema operativo batch. Il sistema ctss veniva eseguito su un ibm 7094, ma il mit decise che lo spazio d'indirizzi del 360, il successore ibm del 7094, era troppo piccolo per il multics, perciò cambiò fornitore. L'ibm decise allora di creare il proprio sistema time sharing, il tss/360. Come il multics, il tss/360 era considerato come una utilità d'elaborazione time sharing. L'architettura di base del 360 fu modificata nel modello 67, per fornire la memoria virtuale. Parecchi enti acquistarono il 360/67 in vista del tss/360.

Il tss/360 subì però alcuni ritardi, perciò furono sviluppati altri sistemi time sharing che servivano come sistemi temporanei finché non fosse stato disponibile il tss/360. All'os/360 è stata aggiunta un'opzione per disporre della opzione time sharing (tso). Il Cambridge Scientific Center dell'ibm sviluppò il cms come sistema per utente singolo e il cp/67 per fornire una macchina virtuale su cui farlo funzionare.

Quando finalmente fu disponibile il tss/360, si dimostrò un fallimento. Era troppo grande e troppo lento. Nessun ente cambiò il proprio sistema temporaneo con il tss/360. Oggi, il time sharing su sistemi ibm si trova diffusamente con l'opzione tso sul sistema mvs o anche con il cms sul cp/67 (ribattezzato vm).

Dato il mancato successo commerciale, è lecito domandarsi quali fossero i problemi principali del tss/360 e del multics. Una parte dei problemi consisteva nel fatto che, pur trattandosi di sistemi progrediti, erano troppo grandi e troppo complessi per essere capitati. Un altro problema derivava dal presupposto che i servizi di calcolo sarebbero stati forniti da un calcolatore grande e remoto. Apparvero poi i minicomputer che diminuirono la necessità di grandi sistemi monolitici. Essi furono seguiti dalle workstation e poi dai personal computer, che portarono la potenza di calcolo sempre più vicina all'utente finale.

A.10 TOPS-20

Nella sua storia dec ha creato diversi importanti sistemi elaborativi. Probabilmente il più famoso sistema operativo associato a dec è vms, un popolare sistema orientato al business tuttora utilizzato nella versione Openvms, un prodotto della Hewlett-Packard. Il più influente sistema operativo della dec è però stato tops-20.

tops-20 è iniziato come un progetto di ricerca alla Bolt, Beranek e Newmann (bbn) intorno al 1970. bbn prese il computer per applicazioni business dec pdp-10, che eseguiva tops-10, aggiunse un sistema hardware per la paginazione della memoria al fine di implementare la memoria virtuale e scrisse un nuovo sistema operativo per la macchina in modo da trarre vantaggio dalle nuove caratteristiche hardware. Il risultato fu tenex, un sistema time sharing general purpose. dec comprò quindi i diritti su tenex e creò un nuovo computer con paginatore hardware incorporato. Ne risultarono il sistema decsystem-20 e il sistema operativo tops-20.

tops-20 disponeva di un interprete avanzato delle riga di comando in grado di offrire un aiuto agli utenti in caso di necessità. Insieme alla potenza del computer e al suo prezzo ragionevole, ciò fece del decsystem-20 il più popolare sistema time sharing del suo tempo. Nel 1984 dec smise di lavorare sulla linea dei pdp-10 a 36 bit per concentrarsi sui sistemi vax a 32 bit con sistema operativo vms.

A.11 CP/M e MS/DOS

I primi computer per hobbisti venivano in genere assemblati a partire da kit, ed eseguivano un solo programma alla volta. I sistemi, con il miglioramento dei componenti dei computer, progredirono verso stadi più evoluti. Uno dei primi sistemi operativi “standard” per tali computer fu cp/m (abbreviazione di Control Program/Monitor), un sistema degli anni Settanta scritto da Gary Kindall della Digital Research. cp/m era prevalentemente installato sul primo “personal computer”, che montava una cpu a 8 bit Intel 8080. In origine cp/m supportava solo 64 kb di memoria ed era in grado di eseguire un solo programma alla volta. Ovviamente era un sistema testuale, dotato di un interprete dei comandi. L’interprete dei comandi era simile a quello di altri sistemi operativi di quei tempi, come il tops-10 della dec.

Quando ibm entrò sul mercato dei personal computer decise di commissionare a Bill Gates e alla sua società il progetto di un nuovo sistema operativo per la cpu a 16 bit che era stata scelta, la Intel 8086. Questo sistema operativo, ms-dos, era simile a cp/m, ma con un insieme più ricco di comandi incorporati, ed era ancora una volta un sistema ispirato a tops-10. ms-dos divenne il sistema operativo per personal computer più popolare dei suoi tempi e dal 1981 fino al 2000 ha continuato la sua evoluzione. ms-dos era in grado di gestire 640 kb di memoria, ma aveva la capacità di indirizzare memoria “estesa” ed “espansa” per andare oltre questo limite. Tuttavia, mancavano alcune funzionalità oggi fondamentali nei sistemi operativi, specialmente la memoria protetta.

A.12 Macintosh OS e Windows

Con l'avvento delle cpu a 16 bit i sistemi operativi diventarono più avanzati, ricchi di funzioni e usabili. Il computer **Apple Macintosh** è stato probabilmente il primo computer disegnato per gli utenti individuali dotato di una interfaccia grafica. A partire dal suo lancio nel 1984, per un certo periodo è stato sicuramente il sistema di maggior successo. Apple Macintosh utilizzava un mouse per il puntamento e la selezione sullo schermo e veniva venduto con molti programmi di utilità che traevano vantaggio da questa nuova interfaccia. I dischi fissi erano abbastanza cari nel 1984; per questa ragione Apple Macintosh veniva offerto con solo una unità floppy da 400 kb.

L'originale Mac os veniva eseguito soltanto sui computer Apple e fu lentamente oscurato da Microsoft Windows (a partire dalla versione 1.0, distribuita nel 1985), che poteva essere installato su molti computer differenti tra loro e prodotti da diverse aziende. Quando le cpu divennero a 32 bit e incorporarono nuove funzionalità, come la memoria protetta e il cambiamento di contesto, questi sistemi operativi videro l'aggiunta di nuove caratteristiche che erano state fino ad allora appannaggio dei mainframe e dei minicomputer.

Con il passare del tempo, i personal computer divennero tanto potenti quanto quei sistemi, e più utili per diversi scopi. I minicomputer scomparvero e vennero rimpiazzati da "server" general purpose o a uso specifico. Anche se i personal computer continuano a migliorare le proprie capacità e prestazioni, i server tendono a restare sempre davanti a loro in fatto di quantità di memoria, capienza del disco, numero e velocità delle cpu disponibili. Al giorno d'oggi i server vengono in genere installati in centri di elaborazione dati e sale macchine, mentre i personal computer sono sulle scrivanie e comunicano tra loro e con i server attraverso la rete.

La rivalità tra Apple e Microsoft per il mercato dei pc continua ancora oggi con le nuove versioni di Windows e Mac os concorrentiali tra loro per funzioni, usabilità e funzionalità delle applicazioni. Altri sistemi operativi, come Amigaos e os/2, sono apparsi negli anni, ma nel lungo periodo non si sono dimostrati avversari temibili dei due sistemi operativi che dominano il mercato dei pc. Nel frattempo, il sistema Linux nelle sue molteplici forme continua a guadagnare popolarità tra gli utenti più esperti tecnicamente – e anche tra i non esperti, come nel caso della rete di computer per bambini **One Laptop per Child (olpc)** (<http://laptop.org/>).

A.13 Mach

Le origini del sistema Mach risalgono al sistema operativo Accent sviluppato alla Carnegie Mellon University (cmu). Il sistema e la filosofia di comunicazione del Mach sono derivati dal sistema Accent, sebbene molte altre parti significative del sistema (per esempio, il sistema di memoria virtuale, la gestione di task e thread) furono sviluppate *ex novo*.

Il progetto di Mach è iniziato intorno alla metà degli anni Ottanta. Il sistema operativo Mach è stato progettato tenendo presenti i seguenti tre obiettivi critici:

1. emulare lo unix 4.3 bsd in modo che i file eseguibili per il sistema unix si potessero eseguire anche nel sistema Mach;
2. essere un sistema operativo moderno che gestisse molti modelli di memoria, e l'elaborazione parallela e distribuita;
3. avere un kernel che fosse più semplice e più facile da modificare di quello del 4.3 bsd.

Lo sviluppo del sistema Mach ha seguito un percorso che parte dal sistema bsd unix. Il codice fu sviluppato inizialmente all'interno del kernel del 4.2bsd, sostituendo i componenti bsd con i componenti Mach appena questi venivano completati; i componenti bsd furono aggiornati alla versione 4.3bsd non appena questa divenne disponibile. Nel 1986 i sottosistemi di memoria virtuale e di comunicazione erano funzionanti per la famiglia di calcolatori dec vax, comprendente versioni multiprocessore. Dopo breve tempo seguirono le versioni per le workstation ibm rt/pc e sun 3. Nel 1987 furono completate sia le versioni multiprocessore Encore Multimax e Sequent Balance, comprendenti la gestione di task e thread, sia le prime versioni ufficiali, Versione 0 e Versione 1.

Dalla Versione 2 il sistema Mach divenne compatibile con i corrispondenti sistemi bsd, poiché comprendeva nel kernel gran parte dello stesso codice bsd. Le nuove caratteristiche e funzioni del sistema Mach fanno sì che il kernel di queste versioni sia più grande dei corrispondenti kernel bsd. Il Mach 3 trasferisce il codice bsd all'esterno del kernel, lasciando un microkernel assai più piccolo. Questo sistema incorpora nel kernel unicamente le funzioni essenziali; tutto il codice specifico dello unix è stato estratto per essere eseguito come server in modalità utente. L'esclusione dal kernel del codice specifico dello unix consente di sostituire il bsd con un altro sistema operativo o di eseguire simultaneamente sopra il microkernel più interfacce di sistemi operativi. Tali interfacce, oltre che per il bsd sono state sviluppate per l'ms-dos, per il sistema operativo Macintosh e per l'osf/1. Questo metodo ha delle similitudini con il concetto di macchina virtuale, anche se in questo caso la macchina virtuale è definita da un livello software (l'interfaccia del kernel del Mach), anziché da una macchina fisica. Per quel che riguarda la versione 3.0, il sistema Mach divenne disponibile per molti sistemi diversi, tra cui i calcolatori con singola cpu della sun Microsystems, Intel, ibm e dec e i sistemi multiprocessore dec, Sequent e Encore.

Il sistema Mach venne spinto all'attenzione dell'industria quando, nel 1989, la Open Software Foundation (osf) annunciò che intendeva impiegarlo come base per il suo nuovo sistema operativo, l'osf/1. L'edizione iniziale dell'osf/1 apparve un anno dopo per competere con lo unix System V, versione 4, il sistema operativo scelto dai membri della unix International (ui). L'osf vantava tra i suoi membri società tecnologicamente importanti come la ibm, dec e hp. L'osf ha nel frattempo cambiato orientamento e solo lo unix dec è basato sul kernel Mach.

Diversamente da unix, che era stato sviluppato senza considerare la multielaborazione, il sistema operativo Mach offre un completa gestione della multielaborazione. Si tratta di una gestione molto flessibile, che varia dai sistemi con memoria condivisa tra le unità d'elaborazione ai sistemi senza memoria condivisa. Il sistema Mach usa i processi leggeri, nella forma di thread d'esecuzione multipli all'interno di un task (o spazio d'indirizzi), per gestire la multielaborazione e l'elaborazione parallela. Il suo esteso uso dei messaggi come unico metodo di comunicazione garantisce che i meccanismi di protezione siano completi ed efficienti. Integrando i messaggi con il sistema della memoria virtuale, il sistema Mach assicura che i messaggi siano gestiti in maniera efficiente. Infine, poiché il sistema di gestione della memoria virtuale impiega i messaggi per comunicare con i processi che gestiscono la memoria ausiliaria, il sistema operativo Mach consente una grande flessibilità nella progettazione e nella realizzazione dei task di gestione degli oggetti di memoria. Offrendo chiamate di sistema di basso livello, o primitive, con le quali si possono costruire funzioni complesse, il sistema Mach riduce le dimensioni del kernel permettendo l'emulazione dei sistemi operativi al livello utente in modo molto simile ai sistemi a macchine virtuali della ibm.

A.14 Sistemi basati su abilitazioni

In questo paragrafo si esaminano due sistemi di protezione basati su abilitazioni. Questi sistemi sono diversi nel grado di complessità e nel tipo di criteri realizzabili su di essi. Nessuno dei due è molto usato, ma entrambi sono interessanti banchi di prova delle teorie sulla protezione.

A.14.1 Un esempio: Hydra

Il sistema Hydra è un sistema di protezione basato sulle abilitazioni, caratterizzato da una notevole flessibilità. Il sistema fornisce un prefissato insieme di possibili diritti d'accesso tra cui sono presenti le principali forme d'accesso come il diritto per lettura, scrittura o esecuzione di un segmento di memoria. Inoltre il sistema offre agli utenti (del sistema di protezione) gli strumenti necessari per dichiarare altri diritti. I diritti definiti dagli utenti sono interpretati esclusivamente dai programmi dell'utente, ma il sistema fornisce la protezione degli accessi nell'uso di questi diritti e dei diritti definiti dal sistema. Queste funzioni costituiscono un notevole passo avanti nella tecnologia della protezione.

Le operazioni sugli oggetti sono definite in modo procedurale, e le procedure che realizzano tali operazioni sono a loro volta oggetti, accessibili indirettamente attraverso abilitazioni. I nomi delle procedure definite dagli utenti si devono comunicare al sistema di protezione, se deve gestire oggetti del tipo definito dagli utenti. Quando si comunica la definizione di un oggetto al sistema Hydra, i nomi delle operazioni sul tipo diventano diritti ausiliari. Questi diritti ausiliari si possono descrivere in un'abilitazione per un'istanza del tipo. Affinché un processo possa eseguire un'operazione su un oggetto tipizzato, l'abilitazione per quell'oggetto deve contenere tra i diritti ausiliari il nome dell'operazione invocata. Questo limite permette di distinguere i diritti d'accesso istanza per istanza e processo per processo.

Il sistema Hydra offre anche l'amplificazione dei diritti. Questo schema permette di certificare che una procedura è *fidata* per agire su un parametro formale di un tipo specificato, per conto di qualsiasi processo che abbia un diritto d'esecuzione della procedura. I diritti posseduti da una procedura fidata sono indipendenti dai diritti posseduti dal processo chiamante e possono anche superarli. Tuttavia tale procedura non deve essere considerata universalmente fidata (per esempio, alla procedura non è permesso di agire su altri tipi), e l'affidabilità non deve essere estesa a qualsiasi altra procedura o altro segmento di programma che può essere eseguito da un processo.

L'amplificazione permette alle procedure di accedere alle variabili che rappresentano il tipo di dato astratto. Se, per esempio, un processo possiede un'abilitazione a un oggetto tipizzato A , quest'abilitazione può comprendere un diritto ausiliario a richiamare una generica operazione P , ma non può comprendere nessuno dei cosiddetti diritti kernel, come i diritti per lettura, scrittura o esecuzione, sul segmento che rappresenta A . Tale abilitazione offre a un processo un mezzo per accedere indirettamente, tramite l'operazione P , alla rappresentazione di A , ma solo per scopi specifici.

D'altra parte, quando un processo invoca l'operazione P su un oggetto A , l'abilitazione d'accesso ad A può essere amplificata quando il controllo passa al corpo del codice di P . Quest'amplificazione può essere necessaria per conferire a P il diritto d'accesso al segmento di memoria che rappresenta A , così da realizzare l'operazione che P definisce sul tipo di dato astratto. Si può permettere al corpo del codice di P di leggere o scrivere direttamente nel segmento di A , anche se il processo chiamante non può farlo. Al termine di P , si riporta l'abilitazione per A al suo stato originale non amplificato. Questo è un tipico caso in cui i diritti posseduti da un processo per accedere a un segmento protetto devono cambiare dinamicamente, secondo il compito da svolgere. La regolazione dinamica dei diritti si esegue per garantire la coerenza delle astrazioni definite dai programmatori. Nel sistema operativo Hydra l'amplificazione dei diritti si può stabilire in modo esplicito nelle dichiarazioni dei tipi di dato astratti.

Quando un utente passa un oggetto come argomento a una procedura, può essere necessario assicurare che la procedura non possa modificare l'oggetto. Questa limitazione si può realizzare facilmente passando un diritto d'accesso senza diritto di modifica (scrittura). Tuttavia, se l'amplificazione è possibile, il diritto di modifica può essere ripristinato, e il requisito di protezione dell'utente può quindi essere aggirato. In generale, naturalmente, un utente può supporre che una procedura esegua correttamente il proprio compito. Tale ipotesi, però, non sempre è corretta, poiché possono verificarsi errori fisici o logici. Il sistema Hydra risolve questo problema limitando le amplificazioni.

Il meccanismo di chiamata di procedura del sistema Hydra è stato progettato come una diretta soluzione al *problema dei sottosistemi mutuamente sospetti*. Questo problema è definito come segue. Si supponga che per un programma possa essere invocato come servizio da diversi utenti (per esempio, una procedura di ordinamento, un compilatore, un videogioco). Quando gli utenti invocano questo programma di servizio, corrono il rischio che il programma non funzioni bene e possa danneggiare i dati forniti, o trattenere, senza averne l'autorità, qualche diritto d'accesso ai dati, da adoperare in seguito. Analogamente, il programma di servizio può avere alcuni file privati, per esempio file di accounting, cui il programma utente chiamante non deve accedere direttamente. Il sistema Hydra fornisce meccanismi per affrontare questo problema in modo diretto.

Un sottosistema di Hydra è costruito sopra un kernel di protezione e può richiedere la protezione dei propri componenti. Un sottosistema interagisce con il kernel tramite un insieme di primitive stabilite dal kernel, che definiscono i diritti d'accesso alle risorse definite dal sottosistema. I criteri relativi all'uso di queste risorse da parte dei processi utenti possono essere definiti da chi progetta il sottosistema, ma si implementano tramite l'uso della protezione degli accessi standard offerta dal sistema di abilitazioni.

Un programmatore può servirsi direttamente del sistema di protezione, dopo aver studiato le sue caratteristiche. Il sistema Hydra offre un'ampia libreria di procedure definite dal sistema che i programmi utenti possono impiegare. Un utente del sistema Hydra può incorporare esplicitamente le chiamate dirette a queste procedure di sistema nel codice del proprio programma, oppure servirsi di un traduttore di programmi interfacciato ad Hydra.

A.14.2 Un esempio: sistema Cambridge CAP

Un diverso orientamento alla protezione basata sulle abilitazioni è stato seguito nella progettazione del sistema Cambridge cap. Il sistema di abilitazioni di cap è più semplice e a prima vista meno potente di quello di Hydra. Tuttavia, con un esame più attento, è

possibile capire che anche questo sistema si può usare per offrire una protezione sicura agli oggetti definiti dagli utenti. cap ha due tipi di abilitazioni. Il tipo ordinario si chiama abilitazione dati, e si può impiegare per fornire l'accesso agli oggetti, ma gli unici diritti forniti sono quelli ordinari di lettura, scrittura o esecuzione dei singoli segmenti di memoria associati all'oggetto. Le abilitazioni dati sono interpretate dal microcodice della macchina cap.

Il secondo tipo è la cosiddetta abilitazione software, che è protetta, ma non interpretata, dal microcodice di cap. L'interpretazione spetta a una procedura *protetta* (cioè privilegiata) che può essere scritta da un programmatore di applicazioni come parte di un sottosistema. A una procedura protetta è associato un particolare tipo di amplificazione di diritti. Quando si esegue il corpo del codice di tale procedura, un processo acquisisce temporaneamente i diritti di lettura o scrittura del contenuto dell'abilitazione software stessa. Questo particolare tipo di amplificazione di diritti corrisponde a una realizzazione delle primitive `seal` e `unseal` sulle abilitazioni. Naturalmente questo privilegio rimane soggetto alla verifica del tipo per assicurare che solo le abilitazioni software per uno specifico tipo astratto possano passare a una particolare procedura. Nessun codice gode di totale fiducia, tranne il microcodice della macchina cap. Per avere riferimenti in merito si vedano le Note bibliografiche alla fine dell'appendice.

L'interpretazione di un'abilitazione software è lasciata esclusivamente al sottosistema, che la esegue per mezzo delle proprie procedure protette. Questo schema permette di realizzare un gran numero di criteri di protezione. Benché un programmatore possa definire le proprie procedure protette, la sicurezza del sistema nel suo complesso non può essere compromessa (anche se tali procedure contengono errori). Il sistema di protezione di base non permetterebbe a una procedura protetta non verificata, definita dall'utente, di accedere a qualsiasi segmento di memoria, o abilitazione, che non appartenga all'ambiente di protezione in cui la procedura stessa risiede. La conseguenza più grave dovuta a una procedura protetta non sicura è la violazione della protezione del sottosistema del quale quella procedura è responsabile.

I progettisti del sistema cap hanno notato che l'uso delle abilitazioni software permette di fare notevoli economie nella formulazione e nella realizzazione di criteri di protezione adeguati ai requisiti delle risorse astratte. Tuttavia un progettista di sottosistemi che voglia usare questa funzione non può semplicemente studiare un manuale, come nel caso del sistema Hydra, ma deve apprenderne i principi e le tecniche di protezione, poiché il sistema non offre alcuna libreria di procedure.

A.15 Altri sistemi

Naturalmente esistono altri sistemi operativi, la maggior parte dei quali ha caratteristiche interessanti. Il sistema operativo mcp, per la famiglia di calcolatori Burroughs è stato il primo a essere scritto in un linguaggio di programmazione di sistema. Impiegava anche la segmentazione e più unità d'elaborazione. Anche il sistema operativo scope per il cdc 6600 era un sistema multiprocessore. Il coordinamento e la sincronizzazione dei processi multipli erano stati progettati sorprendentemente bene.

La storia è disseminata di sistemi operativi che hanno servito uno scopo per un periodo di tempo (lungo o breve che fosse) e poi, una volta scomparsi, sono stati rimpiazzati da sistemi più ricchi di funzioni, in grado di gestire hardware più moderno, e più facili da usare, o anche solamente meglio commercializzati. Siamo sicuri che questa tendenza continuerà in futuro.

Esercizi

A.1 Discutete le considerazioni sulla base delle quali gli operatori decidevano in quale ordine eseguire i programmi sui primi calcolatori gestiti manualmente.

A.2 Quali ottimizzazioni si impiegavano nei primi calcolatori per minimizzare la discrepanza di velocità fra la cpu e l'i/o?

A.3 Per quali aspetti l'algoritmo di sostituzione delle pagine impiegato da Atlas differisce dall'algoritmo a orologio del Paragrafo 10.4.5.2?

A.4 Considerate le code multiple con retroazione impiegate da cts e multics. Supponete che un programma usi regolarmente sette unità di tempo alla volta, prima di eseguire un'operazione di i/o che lo sospenda. Quante unità di tempo sono allocate al programma, nei diversi momenti in cui la sua esecuzione è pianificata?

A.5 Quali sono le conseguenze della funzionalità bsd offerta da server in modalità utente all'interno del sistema operativo Mach?

A.6 Quali conclusioni si possono trarre circa l'evoluzione dei sistemi operativi? Quali fattori hanno portato alcuni sistemi a diventare sempre più popolari e altri a sparire?

APPENDICE B

Windows 7

Microsoft Windows 7 è un sistema operativo client multitasking a 32 o 64 bit con prelazione sviluppato per i microprocessori che implementano i set di istruzioni delle architetture Intel ia-32 e amd64. Il corrispondente sistema operativo server di Microsoft, Windows Server 2008 R2, si basa sullo stesso codice di Windows 7, ma supporta solo le architetture a 64 bit amd64 e ia64 (Itanium). Windows 7 è l'ultimo di una serie di sistemi operativi Microsoft basati sul codice nt, che ha sostituito i precedenti sistemi basati su Windows 95/98. In questa appendice sono trattati gli obiettivi chiave del sistema, la sua architettura a strati che lo rende così facile da usare, il file system, le caratteristiche di networking e l'interfaccia di programmazione.

B.1 Storia

Nella metà degli anni '80 Microsoft e ibm cooperarono per sviluppare il sistema operativo os/2, scritto in linguaggio assembly per sistemi a singola cpu Intel 80286. Nel 1988 Microsoft decise di abbandonare la collaborazione con ibm e sviluppare un proprio sistema operativo facilmente adattabile dalla "nuova tecnologia" (nt), che includesse le interfacce per la programmazione delle applicazioni os/2 e posix. Nell'ottobre del 1988 Dave Cutler, progettista del sistema operativo dec vax/vms, fu assunto e incaricato della progettazione e realizzazione di questo nuovo sistema operativo.

Originariamente per il sistema Windows nt si sarebbe dovuta adottare l'api del sistema operativo os/2 come suo ambiente naturale, ma durante lo sviluppo si scelse di adottare una nuova api a 32 bit (Win32), basata sulla diffusa api a 16 bit di Windows 3.0. Le prime versioni del sistema Windows nt furono Windows nt 3.1 e Windows nt 3.1 Advanced Server (in quel periodo, la più recente versione dell'ambiente Windows a 16 bit era la 3.1). Nella versione 4.0 il sistema Windows nt adottò l'interfaccia utente di Windows 95 e incorporò il software del server Web e il browser; inoltre le procedure dell'interfaccia utente e il codice per la grafica furono spostati all'interno del kernel al fine di migliorare le prestazioni, con l'effetto collaterale di ridurre l'affidabilità del sistema.

Nonostante le prime versioni del sistema nt fossero state adattate per altre architetture, Windows 2000 (uscito nel febbraio 2000) ha limitato la portabilità ai soli processori Intel e compatibili, per ragioni di mercato. Rispetto a Windows nt, Windows 2000 vantava considerevoli innovazioni, quali: aggiunta della Active Directory (un servizio di directory basato sullo standard X.500), migliore gestione dei servizi di rete, supporto ai dispositivi *plug-and-play*, un file system distribuito, la possibilità di aumentare le unità di elaborazione ed espandere la memoria.

Nell'ottobre 2001 l'uscita di Windows xp ha costituito l'evoluzione del sistema operativo desktop Windows 2000 e, nel contempo, come sostituzione di Windows 95/98. Nel 2002 venivano licenziate le versioni server di Windows xp, con il nome di Windows .net Server. Windows xp ha dotato l'interfaccia utente (gui) di un aspetto grafico rinnovato, a cui si aggiungono *nuove funzionalità di uso intuitivo*. Numerose funzionalità sono state introdotte per risolvere in modo automatico i problemi delle applicazioni, e del sistema operativo stesso. Windows xp può contare su una migliore gestione delle connessioni di rete e dei dispositivi (fra cui configurazione automatica delle connessioni wireless, un servizio di messaggistica istantanea, supporto multimediale e applicazioni digitali fotografiche e video), su un notevolissimo aumento delle prestazioni, sia per i pc desktop sia per sistemi multiprocessore più potenti, e su accresciute garanzie di affidabilità e sicurezza.

L'aggiornamento tanto atteso di Windows xp, chiamato Windows Vista, è stato rilasciato nel novembre 2006, ma non è stato ben accolto. Anche se Windows Vista includeva molti miglioramenti che più tardi sono stati riproposti in Windows 7, questi sono stati offuscati dai problemi di lentezza e di compatibilità percepiti nell'utilizzo di Windows Vista. Microsoft ha risposto alle critiche al sistema Vista migliorando i propri processi di ingegnerizzazione e lavorando a stretto contatto con i produttori di hardware e applicazioni per Windows. Il risultato è stato Windows 7, rilasciato nell'ottobre del 2009 insieme alle corrispondenti versioni server di Windows. Tra le modifiche ingegneristiche più significative vi è l'aumento dell'uso della *tracciatura di esecuzione* (*execution tracing*) per l'analisi del comportamento del sistema, al posto dell'utilizzo di contatori o profiling. Il tracing permane in esecuzione nel sistema, monitorando centinaia di scenari di esecuzione. Quando uno di questi scenari fallisce, o quando ha successo ma non ottiene buone prestazioni, è possibile analizzare i dati raccolti per determinare la causa del problema.

Windows 7 si serve di un'architettura client-server (come quella di Mach) per implementare due personalità del sistema operativo, Win32 e posix, tramite processi a livello utente chiamati sottosistemi. Un tempo Windows supportava anche un sottosistema os/2, ma il supporto è stato rimosso in Windows xp a causa della scomparsa di os/2. L'architettura del sottosistema permette di apportare miglioramenti a una singola personalità del sistema operativo senza compromettere la compatibilità con le applicazioni dell'altra. Anche se in Windows 7 continua a essere disponibile il sottosistema posix, l'api Win32 ha visto una grande diffusione e le api posix vengono utilizzate solo in pochi casi. Lo studio dell'approccio a sottosistemi continua a essere interessante dal punto di vista del sistema operativo, ma le tecnologie di virtualizzazione stanno diventando il metodo dominante per eseguire più sistemi operativi su una singola macchina.

Windows 7 è un sistema operativo multiutente, che permette accessi simultanei tramite servizi distribuiti, o tramite istanze multiple dell'interfaccia grafica (gui) gestite dal terminal server del sistema. Le versioni server di Windows 7 permettono più sessioni contemporanee di connessione al server da parte dei sistemi desktop Windows. Le versioni desktop, invece, implementano sessioni virtuali per ogni utente collegato, applicando a tastiera, mouse e monitor la tecnica del multiplex. Questa caratteristica, chiamata *commutazione veloce dell'utente*, permette agli utenti di gestire a turno il controllo del pc senza dover uscire e rientrare nel sistema.

Abbiamo sottolineato in precedenza che qualche applicazione gui in Windows nt 4.0 è stata spostata in modalità kernel. Il ritorno alla modalità utente è iniziato con Windows Vista, che includeva il dwm (desktop window manager) come processo in modalità utente. dwm implementa la composizione del desktop di Windows, fornendo l'aspetto dell'interfaccia Windows Aero, e si colloca al di sopra del software grafico DirectX. DirectX di Windows è ancora eseguito in modalità kernel, così come il codice che implementa la precedente interfaccia e i precedenti modelli grafici di Windows (Win32k e gdi). Windows 7 ha apportato modifiche sostanziali al dwm, riducendo in modo significativo la sua occupazione di memoria e migliorandone le prestazioni.

Windows xp è stata la prima versione di Windows a 64 bit (dal 2001 per ia64 e dal 2005 per amd64). Il file system nativo di nt (ntfs) e molte delle api Win32 hanno sempre usato numeri interi a 64 bit, quando necessario; pertanto, l'estensione a 64 bit di Windows xp serviva principalmente per il supporto di indirizzi virtuali estesi. Le edizioni a 64 bit di Windows, tuttavia, supportano anche memorie fisiche molto più grandi. Quando Windows 7 è stato rilasciato l'isa amd64 era diventato disponibile su quasi tutte le cpu, sia di Intel sia di amd. Inoltre, allo stesso tempo, le memorie fisiche sui sistemi client iniziarono a superare spesso il limite di 4 gb imposto dall'architettura ia-32. Per queste ragioni la versione a 64 bit di Windows 7 è ormai comunemente installata sui sistemi client di buon livello. Poiché l'architettura amd64 è fedelmente compatibile con ia-32 a livello dei singoli processi, in un unico sistema è possibile utilizzare liberamente applicazioni a 32 e 64 bit.

Nel resto della nostra descrizione di Windows 7 non faremo distinzione tra le versioni client e le corrispondenti versioni server. Le diverse versioni si basano sugli stessi componenti di base ed eseguono gli stessi file binari per il kernel e la maggior parte dei driver. Analogamente, anche se Microsoft distribuisce diverse edizioni di ogni release per essere presente sul mercato con diverse fasce di

prezzo, sono poche le differenze che si riflettono sulla parte centrale del sistema. In questa appendice ci concentreremo principalmente sulle componenti fondamentali di Windows 7.

B.2 Princìpi di progettazione

Gli obiettivi di progettazione dichiarati da Microsoft per il sistema operativo Windows includono sicurezza, affidabilità, compatibilità con le applicazioni Windows e posix, alte prestazioni e adattamento a livello internazionale. Alcuni ulteriori obiettivi quali l'efficienza energetica e il supporto dinamico dei dispositivi sono stati recentemente aggiunti a questa lista. Nel seguito discutiamo ciascuno di questi obiettivi e vediamo come essi vengano raggiunti in Windows 7.

B.2.1 Sicurezza

Gli obiettivi di sicurezza di Windows 7 richiedevano qualcosa in più della semplice uniformità agli standard progettuali che hanno permesso a Windows nt 4.0 di ricevere la classificazione di sicurezza C2 dal governo degli Stati Uniti d'America (la classificazione C2 indica un moderato livello di protezione nei confronti di programmi difettosi e di attacchi malevoli; Queste classificazioni sono state definite dal Dipartimento della Difesa nel "Trusted Computer System Evaluation Criteria", noto anche come **Orange Book**. Un'approfondita revisione del codice e attività di testing sono state combinate con strumenti di analisi automatizzati per identificare e studiare potenziali difetti, dietro i quali potrebbero annidarsi falle di sicurezza.

Windows basa la sicurezza sul controllo discrezionale degli accessi. Gli oggetti di sistema, inclusi i file, le impostazioni del registro di sistema e gli oggetti del kernel, sono protetti da **liste di controllo degli accessi** (acl) (si veda il Capitolo 13, Paragrafo 13.4.2). Le acl sono però vulnerabili a errori di utenti e programmatore e ai più comuni attacchi sui sistemi consumer, in cui l'utente viene indotto in maniera ingannevole a eseguire codice, spesso durante la navigazione sul Web. Windows 7 utilizza i livelli di integrità come elementare sistema di capability per il controllo degli accessi. Agli oggetti e ai processi viene assegnata un'integrità bassa, media o alta e Windows non consente a un processo di modificare un oggetto con un livello di integrità più alto, indipendentemente dalle informazioni contenute nella acl.

Tra le altre misure di sicurezza vi sono l'aslr (*address-space layout randomization*), lo stack e l'heap non eseguibili e i servizi di crittografia e firma digitale. aslr ostacola molte forme di attacco impedendo a piccole quantità di codice di essere facilmente iniettate nel codice già caricato in un processo durante la normale esecuzione. Questo meccanismo di protezione fa sì che un sistema sotto attacco sarà probabilmente soggetto a errori o crash, ma non lascerà che il codice malevolo prenda il controllo.

I chip recenti di Intel e amd sono basati sull'architettura amd64 che consente di contrassegnare pagine di memoria in modo che non possano contenere codice eseguibile. Windows tenta di contrassegnare stack e heap di memoria in modo che non possano essere utilizzati per eseguire codice, impedendo attacchi in cui un bug di un programma permette un buffer overflow ed è indotto poi a eseguire il contenuto del buffer. Questa tecnica non può essere applicata a tutti i programmi, perché alcuni si basano sulla modifica dei dati e la loro esecuzione. Una colonna denominata "Data Execution Prevention" nel task manager di Windows mostra quali processi sono contrassegnati per la prevenzione da questi attacchi.

Windows utilizza la crittografia all'interno dei comuni protocolli, per esempio dei protocolli utilizzati per comunicare in modo sicuro con i siti web. La crittografia è usata anche per proteggere da occhi indiscreti i file degli utenti memorizzati su disco. Windows 7 consente agli utenti di crittografare facilmente anche un intero disco, oltre ai dispositivi di memorizzazione rimovibili come le unità usb di memoria flash, per mezzo di una funzione chiamata BitLocker. Se viene rubato un computer con un disco criptato, i ladri avranno bisogno di utilizzare tecnologie molto sofisticate (come un microscopio elettronico) per ottenere l'accesso a qualsiasi file. Windows utilizza firme digitali sui file binari del sistema operativo in modo da poter verificare che i file siano stati prodotti da Microsoft o da un'altra società nota. In alcune edizioni di Windows viene attivato all'avvio un modulo di integrità del codice per garantire che tutti i moduli caricati nel kernel abbiano firme valide, assicurando che questi non siano stati alterati da attacchi off-line.

B.2.2 Affidabilità

Il sistema operativo Windows è cresciuto molto nei suoi primi dieci anni e questa maturazione ha portato al sistema Windows 2000. Allo stesso tempo è aumentata la sua affidabilità, grazie a fattori quali la maturità del codice sorgente, i numerosi test di funzionamento sotto stress, il miglioramento delle architetture delle cpu e il rilevamento automatico di molti gravi errori nei driver di Microsoft e di terze parti. Windows ha successivamente esteso gli strumenti per ottenere maggiore affidabilità includendo l'analisi automatica del codice sorgente per la rilevazione di errori, i test in grado di fornire parametri di ingresso non validi o imprevisti (noti come fuzzing) utili per rilevare gli errori di validazione dell'input e una versione applicativa del verificatore di driver che applica il controllo dinamico su una vasta gamma di errori comuni di programmazione in modalità utente. Ulteriori miglioramenti in termini di affidabilità sono dovuti allo spostamento di una quantità maggiore di codice dal kernel ai servizi in modalità utente. Windows fornisce un ampio supporto per lo sviluppo di driver in modalità utente e diversi servizi di sistema che, una volta, erano nel kernel sono ora eseguiti in modalità utente: tra questi vi sono il dwm e gran parte del software per l'audio.

Uno dei miglioramenti più significativi introdotti grazie all'esperienza nei sistemi Windows consisteva nell'aggiunta dell'opzione di diagnostica della memoria all'avvio, particolarmente preziosa perché ben pochi pc consumer hanno una memoria con correzione d'errore. Quando una ram fallata inizia a perdere bit il risultato è un frustrante comportamento irregolare nel sistema. La disponibilità della diagnostica della memoria ha notevolmente ridotto i livelli di stress degli utenti in possesso di ram scadenti.

Windows 7 ha introdotto una memoria heap con tolleranza agli errori. L'heap apprende dai crash di un'applicazione e inserisce in maniera automatica alcune forme di mitigazione nelle sue future esecuzioni, rendendo così l'applicazione più affidabile anche se contiene bug comuni come l'utilizzo di memoria già liberata o l'accesso alla memoria oltre il limite dell'allocazione.

Il raggiungimento di un'elevata affidabilità in Windows è particolarmente impegnativo, perché quasi un miliardo di computer esegue questo sistema operativo. Persino i problemi di affidabilità che interessano solo una piccola percentuale di utenti possono impattare su un numero altissimo di persone. La complessità dell'ecosistema Windows aggiunge ulteriori ostacoli. Milioni di applicazioni diverse, driver e altri software sono costantemente scaricati ed eseguiti su sistemi Windows e naturalmente c'è anche un flusso costante di

attacchi malware. Dato che Windows è diventato più difficile da attaccare direttamente, gli attacchi prendono sempre più di mira le applicazioni più diffuse.

Per far fronte a queste problematiche, Microsoft fa sempre più affidamento sulle comunicazioni provenienti dalle macchine dei clienti per raccogliere grandi quantità di dati provenienti da tutto l'ecosistema. I dati possono essere campionati per avere informazioni sulle prestazioni che le macchine ottengono, su quali software sono in esecuzione e sui problemi che si incontrano. I clienti possono inviare dati a Microsoft in caso di crash o blocchi delle applicazioni o del sistema. Questo flusso costante di dati proveniente dalle macchine viene raccolto con molta attenzione, con il consenso degli utenti e senza invadere la privacy. Di conseguenza Microsoft va costruendosi un'immagine sempre più chiara di quello che succede nell'ecosistema di Windows e ciò consente continui miglioramenti tramite aggiornamenti software e fornisce le informazioni che guidano lo sviluppo delle versioni future di Windows.

B.2.3 Compatibilità tra applicazioni Windows e posix

Come sottolineato in precedenza, Windows xp non è stato soltanto un aggiornamento di Windows 2000: è il sistema che sostituisce Windows 95/98. Windows 2000 era studiato principalmente per la compatibilità con le applicazioni della clientela professionale, mentre Windows xp prevede una compatibilità molto più elevata con le applicazioni di largo consumo eseguibili in Windows 95/98. La compatibilità delle applicazioni è difficile da ottenere, in quanto molte applicazioni verificano preliminarmente una specifica versione di Windows, risentono delle peculiarità nelle implementazioni delle api, possono nascondere bachi latenti che non si manifestavano nel sistema precedente, e così via. Le applicazioni possono inoltre essere state compilate per un set di istruzioni differente. Windows 7 implementa diverse strategie per eseguire le applicazioni nonostante l'incompatibilità.

Come Windows xp, Windows 7 ha uno strato di compatibilità che si trova fra le applicazioni e le api di Win32: grazie a tale strato, Windows xp appare compatibile con le precedenti versioni di Windows, addirittura quasi **baco per baco** (*bug-for-bug*). Windows xp, come il suo predecessore nt, mantiene il supporto per molte applicazioni a 16 bit usando uno strato di conversione (o *thunking*) che traduce le chiamate delle api a 16 bit in chiamate equivalenti a 32 bit. Analogamente, la versione a 64 bit di Windows xp fornisce uno strato che converte le chiamate delle api a 32 bit in chiamate native a 64 bit.

Il modello a sottosistemi di Windows consente il supporto di personalità multiple del sistema operativo. Come osservato in precedenza, anche se l'api più comunemente usata in Windows è l'api Win32, alcune edizioni di Windows 7 supportano un sottosistema posix. posix è una specifica standard per unix che permette alla maggior parte del software compatibile con unix di essere compilato ed eseguito senza alcuna modifica.

Come ulteriore strumento di compatibilità, varie edizioni di Windows 7 forniscono una macchina virtuale che esegue Windows xp all'interno di Windows 7, in modo da consentire alle applicazioni di ottenere una compatibilità con Windows xp a livello di bug.

B.2.4 Elevate prestazioni

Windows è stato progettato per fornire elevate prestazioni nei sistemi desktop (fortemente condizionati dalle prestazioni dell'i/o), nei sistemi server (dove la cpu è spesso il collo di bottiglia) e nei grandi ambienti multithread e multiprocessore (in cui la gestione dei lock e delle cache è uno snodo cruciale per la scalabilità). Per mantenere alto il rendimento delle prestazioni, nt utilizzava svariate tecniche, quali esecuzione asincrona dell'i/o, protocolli ottimizzati per le reti, grafica basata sul kernel e sofisticate tecniche di gestione della cache del file system. La progettazione degli algoritmi di gestione della memoria e di sincronizzazione è concepita tenendo presenti le prestazioni legate alle cache e ai multiprocessori.

Windows nt è stato progettato per il **multiprocessing simmetrico** (smp). In un computer multiprocessore, più thread possono contemporaneamente essere in esecuzione, anche nel kernel. Windows nt utilizza su ogni cpu uno scheduling dei thread basato su priorità con prelazione. A eccezione dei thread in esecuzione nel dispatcher del kernel o a livello di interrupt, i thread di qualsiasi processo in esecuzione su Windows possono subire la prelazione da thread a più alta priorità. Il sistema, quindi, è in grado di fornire risposte rapide (si veda il Capitolo 5).

I sottosistemi che costituiscono Windows nt comunicano tra loro in modo efficiente mediante le **chiamate di procedura locali** (*local procedure call*, lpc), che garantiscono lo scambio di messaggi ad alte prestazioni. Quando un thread richiede un servizio sincrono da un altro processo attraverso una lpc, il thread che offre il servizio viene contrassegnato come pronto (ready) e la sua priorità viene temporaneamente aumentata per evitare i ritardi di scheduling che si verificherebbero se dovesse attendere thread già in coda.

Windows xp ha ulteriormente migliorato le prestazioni riducendo la lunghezza del codice nelle funzioni critiche, adottando algoritmi migliori e strutture dati dedicate al singolo processore, usando la "colorazione" della memoria per macchine **numa** (*non-uniform memory access*, "accesso non uniforme alla memoria") e implementando protocolli di gestione dei lock scalabili, come le code di spinlock. I nuovi protocolli di locking contribuiscono a ridurre i cicli di bus del sistema; comprendono liste e code prive di lock, l'uso di operazioni atomiche read-modify-write (come l'**incremento interallacciato** – *interlocked increment* –) e altre tecniche avanzate di sincronizzazione.

Quando Windows 7 è stato sviluppato sono state approntate diverse modifiche sul versante della computazione. La computazione client/server aveva nel frattempo assunto maggiore importanza ed è stata dunque introdotta una chiamata di procedura locale avanzata (alpc) per fornire prestazioni più elevate e maggiore affidabilità rispetto alla lpc. Il numero di cpu e la quantità di memoria fisica disponibile sui sistemi più grandi erano aumentati notevolmente: numerosi sforzi sono stati dunque rivolti al miglioramento della scalabilità del sistema operativo.

L'implementazione di smp in Windows nt utilizzava bitmask per rappresentare collezioni di processori e per individuare, per esempio, su quale insieme di processori poteva essere eseguito un particolare thread. Le maschere erano state progettate per occupare una singola parola di memoria, limitando a 64 il numero di processori supportati all'interno di un sistema. Windows 7 ha aggiunto il concetto di gruppi di processori per rappresentare un numero arbitrario di cpu, in modo da poterne ospitare una quantità superiore. Il numero di cpu all'interno dei singoli sistemi ha continuato ad aumentare non solo grazie alla presenza di più core, ma anche per il fatto che i core supportano più di un thread logico in esecuzione alla volta.

Tutte queste cpu aggiuntive hanno portato a una maggior contesa per i lock utilizzati per lo scheduling di cpu e memoria. Windows 7 ha differenziato questi lock. Per esempio, prima di Windows 7 lo scheduler di Windows utilizzava un singolo lock per sincronizzare l'accesso alle code contenenti thread in attesa di eventi. In Windows 7 ogni oggetto ha un proprio lock, permettendo un accesso concorrente alle code. Inoltre, molti percorsi di esecuzione dello scheduler sono stati riscritti e resi privi di lock. Questo cambiamento ha portato a buone prestazioni in termini di scalabilità anche su sistemi con 256 thread hardware.

Altre modifiche sono dovute alla crescente importanza del supporto al calcolo parallelo. Per anni l'industria informatica è stata dominata dalla legge di Moore che ha portato a una maggiore densità dei transistor che a loro volta consentono frequenze di clock più alte per ogni cpu. La legge di Moore continua a valere, ma ormai sono stati raggiunti i limiti che impediscono alla frequenza di clock della cpu di crescere ulteriormente. I transistor vengono ora utilizzati per costruire sempre più cpu all'interno di ogni chip. Nuovi modelli di programmazione per realizzare il parallelismo, come Concurrency RunTime di Microsoft (Concert) e Threading Building Blocks di Intel (tbb), vengono utilizzati per implementare il parallelismo nei programmi C++. Anche se la legge di Moore ha governato l'informatica per 40 anni, sembra ora che la legge di Amdahl, che disciplina il calcolo parallelo, sia destinata a dominare il futuro.

Per supportare il parallelismo a livello di task, Windows 7 fornisce una nuova forma di scheduling in modalità utente (ums) che permette ai programmi di essere scomposti in task poi pianificati sulle cpu disponibili da uno scheduler che opera in modalità utente anziché nel kernel.

L'utilizzo di più cpu sui piccoli computer è solo una parte del cambiamento che sta avvenendo nel calcolo parallelo. Le unità di elaborazione grafica (gpu) accelerano gli algoritmi di calcolo necessari per la grafica utilizzando architetture simd per eseguire una singola istruzione su più dati allo stesso tempo. Tale potenzialità ha condotto all'utilizzo delle gpu per calcoli generici e non solo per la grafica. Il supporto del sistema operativo a software come Opencl e cuda sta permettendo ai programmi di sfruttare le gpu. Windows supporta l'uso delle gpu attraverso il software presente nel suo supporto grafico DirectX. Questo software, chiamato - DirectCompute, permette ai programmi di specificare nuclei computazionali utilizzando lo stesso modello di programmazione hlsl (*high-level shader language*) utilizzato per programmare l'hardware simd per shader grafici. I nuclei computazionali vengono eseguiti molto velocemente sulla gpu e restituiscono i loro risultati alla computazione principale in esecuzione sulla cpu.

B.2.5 Estendibilità

L'estendibilità si riferisce alla capacità di un sistema operativo di tenere il passo con gli sviluppi della tecnologia informatica. Il sistema Windows ha una struttura stratificata che facilita eventuali cambiamenti nel corso del tempo. Il suo modulo executive, che esegue in modalità kernel, fornisce i servizi di sistema fondamentali e le astrazioni a supporto di un uso condiviso del sistema; sopra l'executive operano molti sottosistemi server eseguiti in modalità utente, fra i quali i sottosistemi d'ambiente che simulano differenti sistemi operativi: in questo modo, un programma scritto per Win32 o posso può essere eseguito dal sistema operativo nell'ambiente appropriato. Grazie alla struttura modulare è possibile aggiungere nuovi sottosistemi d'ambiente senza che ciò abbia ripercussioni sull'executive; inoltre, il sistema Windows all'interno del sistema di i/o usa driver caricabili, cosicché si possano aggiungere nuovi file system e nuovi tipi di dispositivi di i/o o di rete mentre il sistema è attivo. Il sistema Windows, come il Mach, adotta un modello client-server e gestisce l'elaborazione distribuita per mezzo di chiamate di procedure remote (rpc) secondo le specifiche dell'Open Software Foundation.

B.2.6 Portabilità

Un sistema operativo è **portabile** (*portable*) se per poter essere eseguito in un'altra architettura richiede un numero di modifiche relativamente piccolo; Windows è progettato per essere portabile. Così come per il sistema operativo unix, la maggior parte del sistema Windows è scritta in linguaggio C o in C++.

Il codice sorgente specifico per l'architettura è relativamente piccolo e si fa molto poco uso di codice assembly. Il porting di Windows su una nuova architettura riguarda soprattutto il kernel, dato che il codice in modalità utente viene scritto quasi esclusivamente per essere indipendente dall'architettura. Per il porting di Windows deve essere effettuato il porting del codice del kernel specifico per l'architettura e, a causa dei cambiamenti nelle principali strutture dati, come il formato della tabella delle pagine, è talvolta necessaria una compilazione condizionale in altre parti del kernel. L'intero sistema Windows deve poi essere ricompilato per il nuovo set di istruzioni della cpu.

I sistemi operativi sono sensibili non solo all'architettura della cpu, ma anche ai chip di supporto alla cpu e ai programmi di boot. L'insieme formato da cpu e chip di supporto è noto complessivamente come chipset. I chipset e il codice di avvio a essi associato determinano la modalità di emissione degli interrupt, descrivono le caratteristiche fisiche di ogni sistema e forniscono interfacce per gli aspetti più profondi dell'architettura della cpu, come la gestione degli errori e dell'alimentazione. Sarebbe davvero oneroso effettuare il porting di Windows su ciascun tipo di chip di supporto e su ogni architettura di cpu. Per ovviare a questo problema, Windows isola la maggior parte del codice dipendente dal chipset in una libreria dinamica (dll) che costituisce lo strato di astrazione dell'hardware (hal) e che viene caricata con il kernel. Il kernel di Windows dipende dalle interfacce hal piuttosto che dai dettagli del - chipset sottostante. Ciò permette di utilizzare il singolo insieme di file binari del kernel e dei driver per una particolare cpu con diversi chipset semplicemente caricando una versione diversa dell'hal.

Nel corso degli anni Windows è stato portato su una serie di diverse architetture di cpu: cpu a 32 bit compatibili con ia-32 di Intel, cpu a 64 bit compatibili con ia64 o amd64, dec Alpha, mips e Powerpc. La maggior parte di queste architetture è scomparsa dal mercato e quando Windows 7 è stato rilasciato solo le architetture ia-32 e amd64 sono state supportate sui computer client e le architetture amd64 e ia64 sui server.

B.2.7 Funzionalità di adattamento internazionale

Windows è anche stato concepito per un'utenza internazionale. Si adatta alle lingue locali grazie all'api per la **gestione delle lingue nazionali** (*national language support, nls*), che fornisce procedure specializzate per trattare date, orari e valute in accordo agli usi dei diversi paesi. I confronti fra sequenze di caratteri tengono conto dei diversi alfabeti. Il codice dei caratteri proprio di Windows è

unicode, i caratteri ansi sono convertiti in unicode (da 8 a 16 bit) prima della manipolazione. Le stringhe contenenti informazioni di sistema si trovano in file sostituibili per adattare il sistema a lingue diverse. È anche possibile impiegare contemporaneamente versioni per lingue differenti, il che è importante per gli utenti e le aziende multilingue.

B.2.8 Efficienza energetica

Aumentare l'efficienza energetica dei computer fa sì che le batterie di computer portatili e netbook durino più a lungo, permette di risparmiare notevolmente sui costi di esercizio e di raffreddamento dei data center e contribuisce a iniziative ecologiche volte a ridurre il consumo di energia da parte delle imprese e dei consumatori. Da qualche tempo Windows ha messo in atto diverse strategie per ridurre il consumo di energia. Quando possibile, le cpu vengono impostate in uno stato di potenza inferiore, per esempio abbassando la frequenza di clock. Inoltre, quando un computer non viene attivamente utilizzato Windows può mettere l'intero sistema in uno stato di basso consumo (sleep) o persino salvare tutta la memoria su disco e spegnere il computer (sospensione). Al ritorno dell'utente, il computer si accende e continua dal suo stato precedente, senza necessità di riavviare sistema e applicazioni.

Windows 7 ha aggiunto alcune nuove strategie per il risparmio energetico, che cresce proporzionalmente al tempo in cui la cpu resta inutilizzata: poiché i computer sono molto più veloci degli esseri umani, può essere risparmiata una notevole quantità di energia mentre gli esseri umani sono fermi a pensare. Il problema è che diversi programmi interrogano costantemente il sistema (polling) per vedere che cosa stia succedendo e tali interrogazioni periodiche impediscono alla cpu di restare inattiva abbastanza a lungo da permettere un risparmio effettivo. Windows 7 è in grado di aumentare il tempo di inattività della cpu ignorando gli impulsi di clock, accordando i timer dei software in un numero inferiore di eventi e "parcheggiando" intere cpu quando il sistema non è troppo carico.

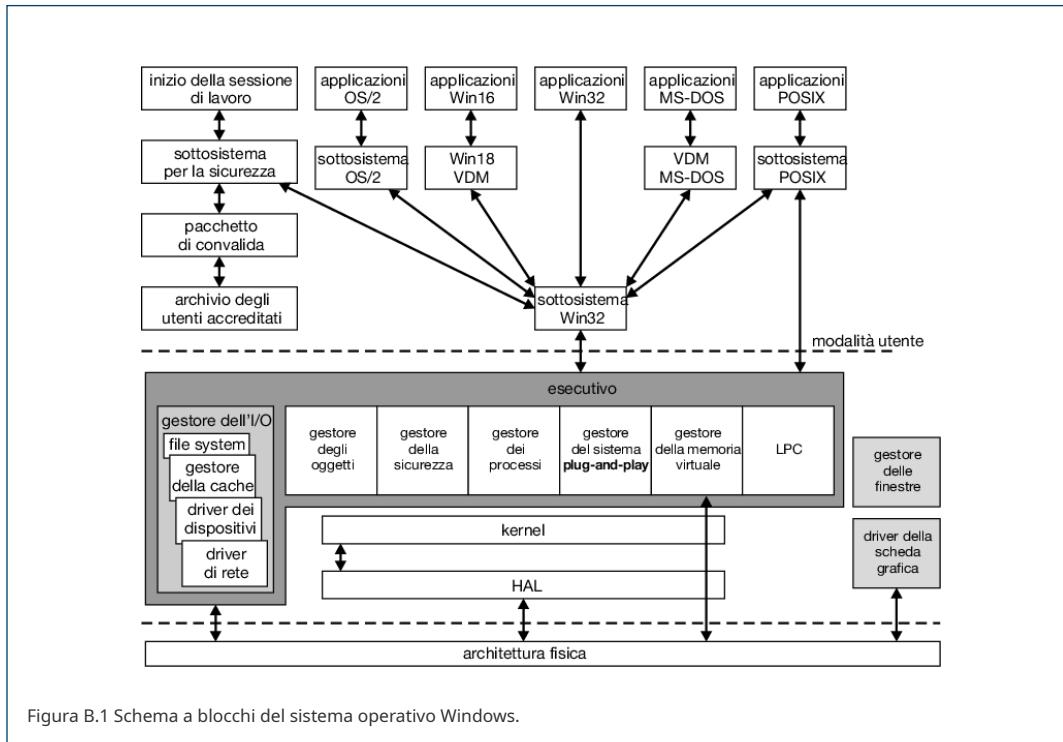
B.2.9 Supporto dinamico dei dispositivi

All'inizio della storia dell'industria dei pc le configurazioni di un computer erano abbastanza statiche; al massimo potevano essere occasionalmente collegati dei nuovi dispositivi alle porte seriali o alle porte per stampante o di gioco presenti sul retro del pc. I primi passi verso la configurazione dinamica dei pc furono i dock per computer portatili e le schede pcmcia: un pc poteva improvvisamente essere collegato o scollegato da tutta una serie di periferiche. In un pc moderno la situazione è completamente diversa. I pc sono progettati per consentire agli utenti di collegare e scollegare una vasta gamma di periferiche in qualsiasi momento. Dischi esterni, chiavette usb, fotocamere e dispositivi simili vengono continuamente inseriti e disinseriti.

Nel sistema Windows il supporto per la configurazione dinamica dei dispositivi è in continua evoluzione. Il sistema è in grado di riconoscere automaticamente i dispositivi quando sono collegati e può trovare, installare e caricare i driver appropriati spesso senza l'intervento dell'utente. Quando i dispositivi vengono scollegati i driver sono automaticamente scaricati e l'esecuzione del sistema continua senza interrompere altri software.

B.3 Componenti del sistema

La struttura del sistema operativo Windows è un sistema stratificato di moduli (Figura B.1). Gli strati principali sono l'hal, il kernel e l'executive, operanti in modo kernel, e un'ampia raccolta di sottosistemi eseguiti in modalità utente; questi ultimi si dividono in due categorie: i sottosistemi d'ambiente, che simulano sistemi operativi diversi, e i sottosistemi di protezione, che forniscono servizi di sicurezza. Uno dei vantaggi principali di questo tipo d'organizzazione è che le interazioni fra i moduli sono semplificate. Il resto di questo paragrafo descrive i diversi strati e sottosistemi.



B.3.1 Strato di astrazione dell'hardware

Lo **strato di astrazione dell'hardware** (hal) è uno strato software che nasconde agli strati superiori le differenze presenti nel chipset, contribuendo alla portabilità del sistema operativo. Lo hal realizza una macchina virtuale usata come interfaccia dal dispatcher del kernel, dall'executive e dai driver dei dispositivi; è sufficiente una sola versione di ogni driver per ogni architettura della cpu, indipendentemente dai chip di supporto utilizzati. I driver mappano i dispositivi e vi accedono direttamente, ma i dettagli specifici del chipset legati alla mappatura in memoria, alla configurazione dei bus per l'i/o, al dma, alla gestione di funzionalità specifiche della scheda madre, sono tutti curati dalle interfacce hal.

B.3.2 Kernel

Il livello kernel di Windows ha quattro compiti principali: lo scheduling dei thread, la gestione delle interruzioni e delle eccezioni, la sincronizzazione a basso livello della cpu e il passaggio tra la modalità utente e la modalità kernel. Il kernel è implementato in linguaggio C, con l'utilizzo del linguaggio assembly solo nei casi in cui è assolutamente necessario per interfacciarsi con i livelli più bassi dell'architettura.

Il kernel è orientato agli oggetti: un tipo di oggetto è un tipo di dato definito dal sistema che ha un insieme di attributi (valori dei dati) e possiede un insieme di metodi (cioè, funzioni o operazioni). Un oggetto è semplicemente un'istanza di uno specifico tipo d'oggetto. Il kernel svolge i suoi compiti usando un insieme d'oggetti i cui attributi memorizzano i dati necessari al kernel, e i cui metodi eseguono le attività del kernel.

B.3.2.1 Dispatcher del kernel

Il dispatcher del kernel costituisce il fondamento per l'executive e i sottosistemi. La gran parte del dispatcher non è mai rimossa dalla memoria centrale, e la sua esecuzione non è mai soggetta a prelazione. I suoi compiti principali sono lo scheduling dei thread, il

cambio di contesto, l'implementazione delle primitive di sincronizzazione, la gestione del timer, delle interruzioni software (chiamate di procedura asincrone e differite), e la consegna (*dispatching*) delle eccezioni.

B.3.2.2 Thread e scheduling

Come molti altri moderni sistemi operativi, il sistema Windows associa le nozioni di processo e thread al codice eseguibile. Ogni processo ha uno o più thread, cioè unità d'esecuzione elaborate dal kernel; ogni thread possiede un suo stato, che comprende una priorità, dei gradi di affinità relativi alle unità d'elaborazione, e informazioni sull'uso della cpu.

I sei possibili stati di un thread sono: pronto, in standby, in esecuzione, in attesa, in transizione, e terminato. Un thread è pronto quando attende d'essere eseguito. Il thread con priorità più alta è posto in standby e sarà il thread successivo a essere eseguito; in un sistema multiprocessore c'è un thread in standby per ogni unità d'elaborazione. Un thread è in esecuzione quando un'unità d'elaborazione lo sta eseguendo; rimane in questo stato finché non viene interrotto da un thread con priorità più alta, o termina, o il suo quanto di tempo scade, o si blocca in attesa su un oggetto dispatcher – per esempio, un evento che segnala la terminazione di un'operazione di i/o. Un thread è in attesa se attende la segnalazione di un oggetto dispatcher. Un thread è in stato di transizione quando attende le risorse necessarie all'esecuzione (per esempio, potrebbe essere in attesa che il suo stack venga caricato dal disco alla memoria). Un thread è terminato quando ha completato l'esecuzione.

Il dispatcher adotta uno schema con 32 livelli di priorità per determinare l'ordine d'esecuzione dei thread; le priorità si dividono in due classi: la classe a priorità variabile, contenente i thread le cui priorità sono comprese fra 1 e 15, e la classe per l'elaborazione in tempo reale, contenente i thread le cui priorità vanno da 16 a 31. Il dispatcher usa una coda per ogni livello di priorità, e percorre l'insieme delle code partendo dalla priorità più alta finché non incontra un thread pronto per l'esecuzione. Se un thread ha una particolare affinità con un certo processore non disponibile in quel momento, il dispatcher lo ignora e continua a cercare un thread pronto per l'esecuzione; se nessun thread è pronto, il dispatcher esegue un thread speciale detto *idle thread*. La classe di priorità 0 è riservata per l'*idle thread*.

Quando scade il suo quanto di tempo, l'interruzione sollevata dal clock accoda al processore una chiamata di procedura differita (dpc) relativa al quanto scaduto. L'accodamento della dpc provoca un interrupt software quando il processore torna alla priorità d'interruzione normale. L'interrupt fa sì che il dispatcher riprogrammi il processore affinché esegua il successivo thread disponibile al livello di priorità del thread che subisce la prelazione.

La priorità di quest'ultimo thread può essere modificata prima che sia rimesso nelle code del dispatcher. Se il thread che subisce prelazione ha priorità di classe variabile, la sua priorità viene ridotta, anche se mai sotto la priorità di base. Il fatto di ridurre la priorità tende a limitare l'impiego di tempo di cpu da parte dei thread con prevalenza d'elaborazione rispetto a quelli con prevalenza di i/o. Quando un thread a priorità variabile abbandona lo stato d'attesa, il dispatcher aumenta la priorità in funzione dell'evento atteso dal thread: un thread che attende i/o dalla tastiera otterrà un notevole aumento di priorità, mentre un thread che attende un'operazione di i/o da un disco otterrà un aumento più moderato. Questa strategia tende a garantire brevi tempi di risposta ai thread interattivi che usano mouse e finestre, e permette ai thread con prevalenza di i/o di tenere occupati i dispositivi di i/o mentre i thread con prevalenza d'elaborazione possono utilizzare in background i cicli di cpu disponibili. Inoltre, la finestra con cui l'utente sta interagendo riceverà un aumento di priorità in modo da migliorare i tempi di risposta.

Lo scheduling può avvenire quando un thread è pronto o è posto nello stato d'attesa, quando termina, o quando un'applicazione ne cambia la priorità o il grado di affinità con un'unità d'elaborazione. Se un thread a più alta priorità diviene pronto durante l'esecuzione di un thread a priorità più bassa, quest'ultimo sarà interrotto; ciò fornisce al thread a priorità maggiore un accesso preferenziale alla cpu. Il sistema Windows, tuttavia, non è un sistema operativo per l'elaborazione in tempo reale stretto (*hard*) perché non garantisce che un thread real time sia eseguito entro limiti di tempo fissati; i thread sono bloccati a tempo indeterminato mentre le dpc e le routine di interrupt di servizio (isr) sono in esecuzione (come verrà spiegato più avanti).

Tradizionalmente gli scheduler del sistema operativo si servivano del campionamento per misurare l'utilizzo della cpu da parte dei thread. Il timer di sistema emetteva un segnale periodico e il gestore degli interrupt del timer prendeva nota di quali thread erano in esecuzione al momento dell'interrupt e di quali tra questi erano eseguiti in modalità kernel o utente. Questa tecnica di campionamento era necessaria perché la cpu non aveva un clock ad alta risoluzione, oppure perché l'accesso al clock era troppo costoso o troppo inaffidabile per poter essere effettuato frequentemente. Anche se efficiente, il campionamento era impreciso e portava ad anomalie come il conteggio del tempo di gestione di un interrupt come tempo del thread e la conseguente interruzione di thread che avevano utilizzato solo una frazione del loro quanto di tempo. A partire da Windows Vista l'uso del tempo di cpu è stato monitorato utilizzando il contatore timestamp hardware (tsc), presente nei processori recenti. L'utilizzo del tsc conferisce una maggiore accuratezza ai calcoli relativi all'utilizzo della cpu e lo scheduler non interrompe più i thread prima che questi abbiano utilizzato completamente il loro quanto di tempo.

B.3.2.3 Realizzazione delle primitive di sincronizzazione

Le principali strutture dati del sistema operativo sono gestite come oggetti che usano funzionalità comuni per l'allocazione, il conteggio dei riferimenti e la sicurezza. Gli oggetti dispatcher controllano, nel sistema, il dispatching e la sincronizzazione.

- L'oggetto evento è usato per registrare il verificarsi di un evento e per sincronizzarlo con una qualche azione. Gli eventi di notifica segnalano tutti i thread in attesa, mentre gli eventi di sincronizzazione segnalano un singolo thread in attesa.
- L'oggetto mutante fornisce la mutua esclusione in modalità kernel o utente, associata con la nozione di proprietà.
- Il mutex, disponibile unicamente in modalità kernel, garantisce mutua esclusione senza pericolo di stallo.
- Un oggetto semaforo opera come un contatore, per controllare il numero di thread che accede alla risorsa.
- L'oggetto thread è l'entità schedulata dal dispatcher del kernel ed è associata con un oggetto processo, che incapsula uno spazio di indirizzamento virtuale. L'oggetto thread riceve un segnale all'uscita del thread e l'oggetto processo all'uscita del processo.
- Gli oggetti timer sono usati per tenere traccia del tempo e per segnalare timeout quando le operazioni impiegano troppo tempo e devono essere interrotte, oppure quando deve essere schedulata un'attività ripetitiva.

A molti degli oggetti del dispatcher si può accedere in modalità utente tramite un'operazione `open`, che restituisce un handle (o riferimento); il codice in modalità utente controlla periodicamente e/o attende in maniera bloccante gli handle per sincronizzarsi con gli altri thread, così come con il sistema operativo (si veda il Paragrafo B.7.1).

B.3.2.4 Interruzioni software: chiamate di procedura asincrona e differita

Il dispatcher implementa due tipi di interruzioni software: la **chiamata di procedura asincrona** (*asynchronous procedure call*, apc) e la chiamata di procedura differita (*deferred procedure call*, dpc). La chiamata di procedura asincrona interrompe un thread in esecuzione e richiama una procedura. Le apc servono per iniziare l'esecuzione di un nuovo thread, per sospendere o riattivare un thread esistente, per terminare i thread e i processi, per consegnare la notifica relativa a un'operazione di i/o asincrona che è stata completata e per estrarre il contenuto dei registri della cpu da un thread in esecuzione. Le apc sono accodate a specifici thread e permettono al sistema di eseguire sia il codice di sistema sia quello utente entro il contesto del processo. L'esecuzione di una apc in modalità utente non può avvenire in qualsiasi momento, ma solamente quando il thread è in attesa nel kernel ed è contrassegnato come *alertable* (allertabile).

Le chiamate di procedura differite sono finalizzate a posporre l'elaborazione delle interruzioni. Dopo aver gestito tutti i processi urgenti bloccati da interruzioni dei dispositivi, la **procedura di servizio delle interruzioni** (*interrupt service routine*, isr) pianifica i lavori di elaborazione rimasti in sospeso accodando una dpc. Il dispatcher pianifica le interruzioni software con una priorità più bassa di quella delle interruzioni dei dispositivi, in modo che le dpc non blocchino altre isr, ma più alta della priorità con cui il thread è in esecuzione. Oltre che per rinviare l'elaborazione delle interruzioni dei dispositivi, il dispatcher usa le dpc per elaborare le interruzioni generate dal timer e per interrompere l'esecuzione dei thread il cui quanto di tempo è scaduto.

L'esecuzione delle dpc impedisce ai thread di essere schedulati nel processore corrente, e alle apc di segnalare il completamento di un'operazione di i/o; ciò affinché le procedure dpc non impieghino quantità di tempo eccessive per essere completate. In alternativa, il dispatcher mantiene un gruppo di "thread di lavoro". Le isr e le dpc possono accodare i lavori da svolgere ai thread di lavoro e questi saranno eseguiti utilizzando il normale scheduling dei thread. Le dpc sono limitate in modo che non possano incorrere in page fault, richiamare servizi del sistema o compiere qualsiasi altra azione che potrebbe sfociare nel tentativo di sospendere l'esecuzione in attesa di un oggetto dispatcher. A differenza delle apc, le routine dpc non fanno ipotesi in merito a quale contesto di processo il processore stia eseguendo.

B.3.2.5 Eccezioni e interruzioni

Il dispatcher del kernel fornisce anche la gestione delle eccezioni e delle interruzioni; Windows definisce diverse eccezioni indipendenti dall'architettura, tra cui:

- accesso illegale alla memoria;
- overflow in operazioni su interi;
- overflow e underflow in operazioni in virgola mobile;
- divisone intera per zero;
- divisone in virgola mobile per zero;
- istruzione illegale;
- dati non allineati;
- istruzione privilegiata;
- errore di lettura della pagina;
- violazione d'accesso;
- quota di paginazione ecceduta;
- punto d'arresto (*breakpoint*) del debugger;
- singolo passo del debugger.

Le eccezioni semplici possono essere trattate dai trap handler; le altre sono gestite dal dispatcher delle eccezioni del kernel, il quale annota la causa dell'eccezione per individuarne l'appropriato gestore.

Quando si verifica un'eccezione in modalità kernel, il dispatcher delle eccezioni invoca semplicemente una procedura che ne individua il gestore; nel caso in cui non se ne trovi alcuno avviene un errore di sistema fatale che lascia l'utente davanti al famigerato "schermo blu della morte", il quale sta a significare l'arresto totale del sistema.

La gestione delle eccezioni è più complessa nel caso dei processi eseguiti in modalità utente, perché un sottosistema d'ambiente (per esempio il posix) può specificare una porta di debugging e una porta per le eccezioni per ogni processo da esso creato (per maggiori dettagli sulle porte si veda il Paragrafo B.3.3.4). Se è registrata una porta di debugging, il dispatcher delle eccezioni trasmette l'eccezione a tale porta; se non si trova una porta di debugging o se essa non è in grado di gestire l'eccezione, il dispatcher tenta di individuare un appropriato gestore dell'eccezione; se non riesce a trovarlo, richiama di nuovo il debugger in modo che esso possa rilevare l'errore. Se tale programma non è in esecuzione il dispatcher manda un messaggio alla porta per le eccezioni del processo per dare al sottosistema d'ambiente la possibilità di tradurre l'eccezione: per esempio l'ambiente posix traduce messaggi d'eccezione in segnali posix prima di mandarli al thread che ha causato l'eccezione. In ultima analisi, se tutto il resto non ha funzionato il kernel termina forzatamente il processo contenente il thread che ha causato l'eccezione.

Quando Windows non riesce a gestire un'eccezione, può costruire una descrizione dell'errore che si è verificato e richiedere l'autorizzazione da parte dell'utente per inviare le informazioni a Microsoft per ulteriori analisi. In alcuni casi, l'analisi automatica di Microsoft è in grado di riconoscere immediatamente l'errore e suggerire una correzione o una soluzione alternativa.

Il dispatcher delle interruzioni del kernel gestisce le interruzioni richiamando una **procedura di servizio delle interruzioni** (*interrupt service routine*, isr) o una procedura interna del kernel. Un segnale d'interruzione è rappresentato da un oggetto interruzione contenente tutte le informazioni necessarie per gestire l'interruzione, il che rende facile associare procedure di servizio a un segnale d'interruzione senza dover accedere direttamente al dispositivo in questione.

Diverse architetture hanno tipi e numeri d'interruzione diversi; per garantire la portabilità il dispatcher delle interruzioni associa le interruzioni a un insieme convenzionale. Ai segnali d'interruzione è assegnata una priorità e sono serviti secondo questa priorità in ordine decrescente: nel sistema Windows ci sono 32 livelli d'interruzione (irq): otto sono riservati all'uso del kernel; gli altri ventiquattro rappresentano segnali d'interruzione provenienti dai dispositivi tramite la mediazione dello hal (sebbene la maggior parte dei sistemi ia-32 ne usi solo 16). L'elenco delle interruzioni di Windows è riportato nella Figura B.2.

Livelli di interruzione	Tipi di interruzione
31	controllo di macchina o errore sul bus
30	calo di tensione
29	comunicazione fra unità d'elaborazione (richiesta d'azione a un'altra unità d'elaborazione, ad esempio, avvio di un processo o aggiornamento dei TLB)
28	orologio
27	profile
3-26	ordinarie interruzioni IRQ dei PC
2	dispatch e chiamata di procedura differita (DPC) (kernel)
1	chiamata di procedura asincrona (APC)
0	passiva

Figura B.2 Segnali d'interruzione del sistema Windows.

Il kernel usa una tabella di dispatch delle interruzioni (idt) per associare a ogni livello delle interruzioni una procedura di servizio; nel caso di un multiprocessore, il kernel di Windows mantiene idt separate per ogni processore, e l'irql di ogni processore si può regolare in modo indipendente per mascherare determinate interruzioni. Tutte le interruzioni che occorrono a un livello pari o inferiore dell'irql di un processore sono bloccate finché l'irql non si sia abbassato da un thread a livello del kernel o dal completamento di esecuzione di una isr. Il sistema Windows sfrutta questa caratteristica impiegando le eccezioni per eseguire funzioni di sistema: per esempio, il kernel usa eccezioni per avviare il dispatch di un thread, per gestire i timer e sincronizzare i thread con il completamento degli i/o.

B.3.2.6 Commutazione tra thread in modalità utente e thread in modalità kernel

Quando un programmatore Windows pensa a un thread in realtà sta pensando a due thread: un thread in modalità utente (ut) e un thread in modalità kernel (kt). Ognuno di questi thread ha il proprio stack, i propri valori dei registri e il proprio contesto di esecuzione. Un ut richiede un servizio di sistema eseguendo un'istruzione che provoca una trap alla modalità kernel. Lo strato kernel esegue quindi un gestore di trap che passa dal thread ut al corrispondente thread kt. Quando un kt ha completato l'esecuzione nel kernel ed è pronto a tornare al corrispondente ut, viene invocato lo strato kernel per effettuare il passaggio all'ut, che continua la sua esecuzione in modalità utente.

Windows 7 modifica il comportamento dello strato kernel per supportare lo scheduling in modalità utente dei thread ut. Lo scheduler in modalità utente di Windows 7 supporta lo scheduling cooperativo. Da un ut si può passare esplicitamente a un altro ut chiamando lo scheduler in modalità utente, senza intervento del kernel. Lo scheduling in modalità utente è descritto in dettaglio nel Paragrafo B.7.3.7.

B.3.3 Executive

L'executive del sistema Windows fornisce una serie di servizi utilizzabili da tutti i sottosistemi d'ambiente. I servizi sono raggruppati come segue: gestore degli oggetti, gestore della memoria virtuale, gestore dei processi, servizi relativi alle chiamate di procedura locali avanzate, gestore dell'i/o, della cache, monitor della sicurezza dei riferimenti, gestori del *plug-and-play* e dell'alimentazione, registri (*registry*) e avvio del sistema.

B.3.3.1 Gestore degli oggetti

Per la gestione di entità in modalità kernel, Windows usa un insieme di interfacce manipolate dai programmi in modalità utente. In Windows queste entità sono chiamate oggetti, mentre il componente di codice eseguibile che li gestisce è detto gestore degli oggetti (*object manager*). Esempi di oggetti sono: semafori, mutex, eventi, processi e thread. Gli oggetti citati sono tutti *oggetti dispatcher*. I thread possono bloccarsi nel dispatcher del kernel, nell'attesa che uno di questi oggetti sia segnalato. Processi, thread e api della memoria virtuale usano gli handle per identificare il processo o il thread su cui operare. Altri esempi di oggetti includono file, sezioni, porte e vari oggetti interni di i/o. Gli oggetti file sono usati per mantenere lo stato aperto dei file e dei dispositivi. Le sezioni sono usate per mappare i file. I punti terminali locali delle comunicazione sono implementati come oggetti porta.

Il codice in modalità utente accede a questi oggetti utilizzando un valore opaco chiamato handle, che viene restituito da molte api. Ogni processo ha una tabella degli handle contenente le voci che tengono traccia degli oggetti utilizzati dal processo. Il processo di sistema, che contiene il kernel, ha la sua tabella degli handle protetta dal codice utente. Le tabelle degli handle sono rappresentate in Windows da una struttura ad albero che può contenere da 1.024 fino a oltre 16 milioni di handle. Il codice in modalità kernel può accedere a un oggetto utilizzando un **handle** o un **puntatore di riferimento** (*referenced pointer*).

Un processo ottiene un handle creando un oggetto, aprendo un oggetto esistente, ricevendo un handle duplicato da un altro processo o ereditando un handle dal processo padre. Quando un processo termina tutti i suoi handle ancora aperti vengono implicitamente chiusi. Dal momento che il gestore degli oggetti è l'unica entità che genera handle di oggetti, si tratta del luogo naturale per controllare la sicurezza. Il gestore degli oggetti verifica se un processo ha il diritto di accedere a un oggetto quando tenta di aprirlo. Il gestore degli oggetti applica inoltre delle quote, come la quantità massima di memoria che un processo può utilizzare, tenendo in conto della memoria occupata da tutti gli oggetti referenziati dal processo e rifiutando di allocare ulteriore memoria quando la quantità di memoria accumulata eccede la quota assegnata al processo.

Il gestore degli oggetti tiene traccia di due conteggi per ciascun oggetto: il numero di handle e il numero di puntatori. Il primo è il numero di handle che fanno riferimento all'oggetto nelle tabelle degli handle di tutti i processi, compreso il processo di sistema che contiene il kernel. Il conteggio dei puntatori che fanno riferimento all'oggetto viene incrementato ogni volta che il kernel ha bisogno di un nuovo puntatore e decrementato quando il kernel ha finito di utilizzare un puntatore. Lo scopo del conteggio di questi riferimenti è quello di garantire che un oggetto non venga liberato mentre vi fa ancora riferimento un handle o un puntatore interno al kernel.

Il gestore degli oggetti mantiene lo spazio dei nomi interno di Windows. A differenza di unix, che radica lo spazio dei nomi del sistema nel file system, Windows utilizza uno spazio dei nomi astratto e collega i file system come dispositivi. L'assegnazione di un nome a un oggetto di Windows dipende da chi ha creato l'oggetto. Processi e thread vengono creati senza nome e vi si fa riferimento sia mediante handle sia tramite un identificatore numerico separato. Gli eventi di sincronizzazione hanno di solito dei nomi, in modo che possano essere aperti da processi indipendenti. Un nome può essere permanente o temporaneo: un nome permanente rappresenta un'entità, come un disco rigido, che rimane anche se nessun processo vi sta accedendo; un nome temporaneo esiste solo fino a quando un processo contiene un handle all'oggetto. Il gestore degli oggetti supporta le directory e i link simbolici nello spazio dei nomi. A titolo di esempio, le lettere delle unità disco in ms-dos vengono implementate usando collegamenti simbolici: \Global??\C: è un collegamento simbolico all'oggetto dispositivo \Device\HarddiskVolume2 che rappresenta un volume del file-system montato nella directory \Device.

Ogni oggetto, come accennato in precedenza, è un'istanza di un tipo di oggetto. Il tipo di oggetto specifica come devono essere allocate le istanze, come devono essere definiti i campi di dati e come deve essere implementato il set standard di funzioni virtuali utilizzate per tutti gli oggetti. Le funzioni standard implementano operazioni come la mappatura dei nomi agli oggetti, la chiusura, l'eliminazione e l'applicazione di controlli di sicurezza. Le funzioni specifiche per un particolare tipo di oggetto sono implementate dai servizi di sistema progettati per funzionare su quel particolare tipo di oggetto, non da metodi specificati nel tipo di oggetto.

La funzione `parse()` è la più interessante tra le funzioni standard e consente l'implementazione di un oggetto. I file system, l'archivio di configurazione del registro di sistema e gli oggetti gui utilizzano le funzioni di parse per estendere lo spazio dei nomi di Windows, e ne rappresentano i principali utilizzatori.

Tornando al nostro esempio di assegnazione dei nomi in Windows, gli oggetti dispositivo utilizzati per rappresentare i volumi del file system forniscono una funzione di parse per permettere a un nome come \Global??\C:\foo\bar.doc di essere interpretato come il file \foo\bar.doc del volume rappresentato dall'oggetto dispositivo HarddiskVolume2. Possiamo osservare come l'assegnazione dei nomi, le funzioni di parse, gli oggetti e gli handle lavorano insieme illustrando i passi che permettono di aprire un file in Windows.

1. Un'applicazione richiede l'apertura di un file denominato C:\foo\bar.doc.
2. Il gestore degli oggetti trova l'oggetto dispositivo HarddiskVolume2, cerca la procedura di parse `IopParseDevice` dal tipo di oggetto e la richiama con il nome del file relativo alla radice del file system.
3. `IopParseDevice()` alloca un oggetto file e lo passa al file system che fornisce i dettagli su come accedere a C:\foo\bar.doc sul volume.
4. Quando il file system ha terminato, `IopParseDevice()` assegna una voce nella tabella degli handle per l'oggetto file per il processo corrente e restituisce l'handle all'applicazione.

Se il file non può essere aperto correttamente, `IopParseDevice()` cancella l'oggetto file che ha allocato e restituisce all'applicazione una segnalazione di errore.

B.3.3.2 Gestore della memoria virtuale

Il componente dell'executive che gestisce lo spazio degli indirizzi virtuali, l'allocazione della memoria fisica e la paginazione è il **gestore della memoria virtuale** (*virtual memory manager*, vm). Il gestore della vm è concepito assumendo che l'architettura sottostante sia in grado di associare indirizzi virtuali a indirizzi fisici, possieda un meccanismo di paginazione, realizzzi trasparentemente la coerenza della cache nei sistemi multiprocessore, e permetta l'associazione di più elementi della tabella delle pagine alla stessa pagina fisica.

Il gestore della vm di Windows adotta uno schema di gestione basato su pagine della dimensione di 4 kb e 2 mb su processori amd64 e ia-32 compatibili, e di 8 kb su ia64. Le pagine di dati allocate a un processo, ma non residenti nella memoria fisica, sono memorizzate nel file di paginazione in un disco o mappate direttamente su un file ordinario di un file system locale o remoto. Le pagine possono anche essere marcate come *zero-fill-on-demand*, il che ha l'effetto di inizializzarle a zero prima dell'allocazione, cancellandone i contenuti precedenti.

Sui processori ia-32, i processi hanno 4 gb di memoria virtuale a testa. I 2 gb superiori sono quasi identici per tutti i processi, e servono a Windows per accedere in modalità kernel alle strutture dati e al codice del sistema operativo. Nel caso dell'architettura amd64 Windows fornisce 8 tb di spazio di indirizzamento virtuale per la modalità utente nell'ambito dei 16 eb supportati dall'hardware per ogni processo.

Alcune aree cruciali della regione di memoria del kernel che variano da processo a processo sono la mappa della page table, l'iperspazio e lo spazio della sessione. L'hardware denota le tabelle delle pagine di un processo per mezzo dei numeri dei frame fisici e la mappa (*self-map*) della page table rende i contenuti della page table dei processi accessibili tramite indirizzi virtuali. L'iperspazio mappa le informazioni relative all'insieme di lavoro corrente del processo nello spazio degli indirizzi accessibile in modalità kernel. Lo spazio della sessione serve per condividere un'istanza di Win32 e altri driver legati alla specifica sessione fra tutti i processi della stessa sessione del terminal server (ts). Sessioni ts distinte condividono istanze differenti di questi driver, anche se questi sono mappati agli stessi indirizzi virtuali. La regione, più bassa, della modalità utente dello spazio degli indirizzi virtuali è specifica per ogni processo e accessibile sia dai thread utente che dai thread del kernel.

Per assegnare memoria virtuale il gestore della vm procede in due passi: innanzitutto *riserva* una o più pagine di indirizzi virtuali dello spazio d'indirizzi virtuali del processo; quindi *impegna* l'assegnazione dello spazio per la memoria virtuale (spazio in memoria fisica o nel file di paginazione). Il sistema operativo Windows può limitare lo spazio del file di paginazione impiegato da un processo imponendo una quota massima d'assegnazione; un processo può rilasciare parti di memoria non più in uso a beneficio di altri processi. Le api che riservano gli indirizzi virtuali e impegnano la memoria virtuale ricevono come parametro l'handle di un oggetto processo. In questo modo, un dato processo è in grado di controllare la memoria virtuale di un altro processo. I sottosistemi d'ambiente gestiscono la memoria dei loro processi client in questo modo.

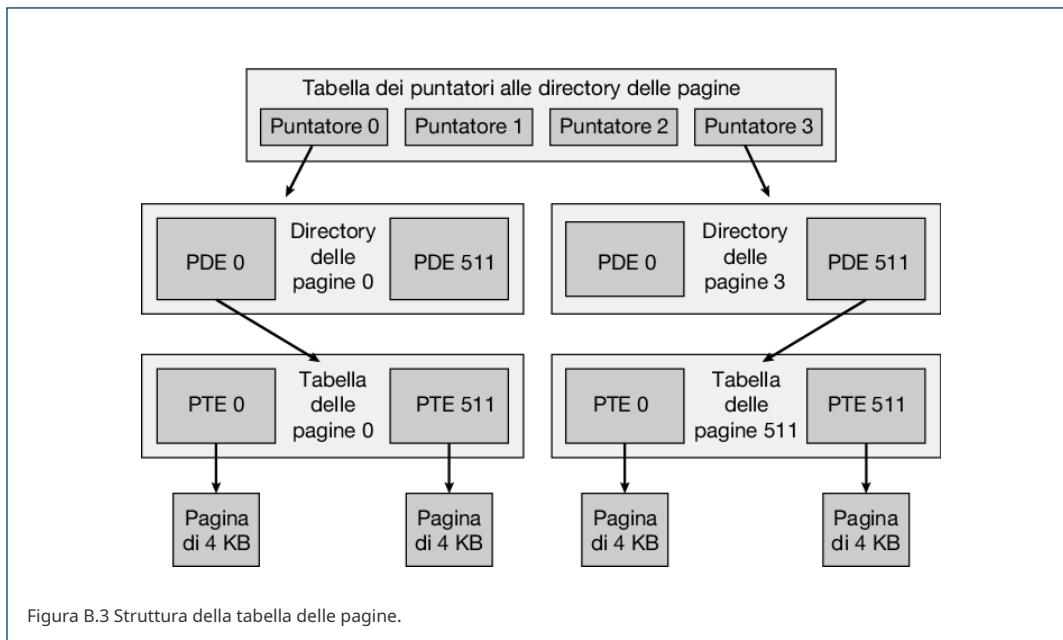
Windows implementa la memoria condivisa tramite gli oggetti sezione. Un processo, dopo aver ottenuto un handle dell'oggetto sezione, mappa la porzione di memoria della sezione su un intervallo di indirizzi, chiamato vista (*view*). Un processo può stabilire una vista dell'intera sezione o solo della parte di sezione di cui ha bisogno. Windows consente alle sezioni di essere mappate non solo nel processo corrente, ma in un qualunque processo per cui il chiamante possiede un handle.

Le sezioni sono utilizzabili in molti modi. Una sezione può utilizzare come memoria ausiliaria il file di paginazione di sistema o un file ordinario (un file mappato in memoria). Una sezione può essere *basata* (*based*), nel senso che appare allo stesso indirizzo virtuale a ogni processo che vi accede. Le sezioni possono anche rappresentare la memoria fisica, consentendo così a un processo a 32 bit di accedere a una maggior quantità di memoria fisica di quanta ne possa essere contenuta nel suo spazio di indirizzamento virtuale. Infine la protezione delle pagine della sezione si può impostare per la sola lettura, lettura e scrittura, sola esecuzione, non accessibile, e copiatura su scrittura.

Le ultime due impostazioni di protezione richiedono alcune spiegazioni.

- Una *pagina non accessibile* solleva un'eccezione qualora si tenti di accedervi; l'eccezione si può usare, fra l'altro, per controllare se un programma difettoso esegue iterazioni che vanno oltre la dimensione di un array o semplicemente per rilevare se un programma tenta di accedere a indirizzi virtuali non presenti in memoria. Gli stack in modalità utente e kernel utilizzano le pagine non accessibili come sentinelle (*guard page*) per rilevare eventuali stack overflow. Un altro possibile uso è il controllo del superamento delle dimensioni del buffer (*buffer overrun*) nello heap. Sia l'allocatore della memoria utente sia l'allocatore del kernel usato dal verificatore del dispositivo possono essere configurati per mappare ogni allocazione richiesta fino alla fine di una pagina seguita da una pagina non accessibile; ciò permette di rilevare errori di programmazione che causano accessi oltre il limite dell'allocazione.
- Il meccanismo di copiatura su scrittura permette al gestore della vm di risparmiare memoria: quando due processi vogliono copie indipendenti dei dati dello stesso oggetto, il gestore pone solo una copia condivisa nella memoria fisica e attribuisce a quella sezione la proprietà di copiatura su scrittura; se uno dei processi tenta di scrivere dati in una delle pagine caratterizzate da tale proprietà, il gestore fornisce al processo una copia privata della pagina che può poi essere modificata.

La traduzione degli indirizzi virtuali in Windows sfrutta una tabella delle pagine a più livelli. Nel caso di processori ia-32 e amd64, ogni processo ha una directory delle pagine contenente 512 elementi della directory delle pagine (*page-directory entries*, pde), da 8 byte l'uno; ognuno di loro punta a una tabella delle pagine, che a sua volta contiene 512 elementi della tabella delle pagine (*page-table entries*, pte) di 8 byte, che infine puntano a frame da 4 kb nella memoria fisica. Per una serie di motivi, l'hardware richiede che le directory delle pagine o le tabelle dei pte a ogni livello di una tabella delle pagine multilivello occupino una sola pagina. In tal modo, il numero di pde o pte contenuti in una pagina determina quanti indirizzi virtuali sono tradotti da quella pagina. Per uno schema di questa struttura si veda la Figura B.3.



La struttura fin qui descritta può essere utilizzata per rappresentare la traduzione di solo 1 gb di indirizzi virtuali. Nel caso di ia-32 è necessario un secondo livello di directory delle pagine contenente solo quattro voci, come mostrato nel diagramma. Su processori a 64 bit sono necessari più livelli. Per amd64 Windows utilizza in totale quattro livelli completi. La dimensione totale di tutte le pagine della tabella delle pagine necessarie per rappresentare pienamente anche solo uno spazio di indirizzamento virtuale a 32 bit per un processo è di 8 mb. Il gestore della vm alloca pagine di pde e pte secondo le necessità e sposta le pagine della tabella su disco quando non sono utilizzate. Le pagine della tabella sono riportate in memoria quando vi si fa riferimento.

Descriviamo ora come gli indirizzi virtuali siano tradotti in indirizzi fisici su processori ia-32. Un valore di 2 bit può rappresentare i valori 0, 1, 2, 3, un valore di 9 bit può rappresentare valori nell'intervallo 0-511 e un valore di 12 bit valori da 0 a 4095. Un valore di 12 bit può pertanto selezionare qualsiasi byte all'interno di una pagina di 4 kb di memoria. Un valore di 9 bit può rappresentare uno qualsiasi dei 512 pde o pte in una directory delle pagine o in una pagina della tabella dei pte. Come mostrato nella Figura B.4, la traduzione di un puntatore di indirizzo virtuale nell'indirizzo di un byte nella memoria fisica si effettua suddividendo il puntatore a 32 bit in quattro valori. A partire dai bit più significativi.



Figura B.4 Composizione di un indirizzo di memoria virtuale di un'architettura ia32.

- Due bit vengono utilizzati per indicizzare i quattro pde al livello superiore della tabella delle pagine. Il pde selezionato conterrà il numero di pagina fisica per ciascuna delle quattro pagine della directory delle pagine che mappano 1 gb di spazio di indirizzi.
- Nove bit vengono utilizzati per selezionare un altro pde, questa volta da una directory delle pagine di secondo livello. Questo pde conterrà i numeri di pagina fisica di un massimo di 512 pagine della tabella dei pte.
- Nove bit vengono utilizzati per selezionare uno dei 512 pte dalla pagina selezionata della tabella dei pte. Il pte selezionato conterrà il numero di pagina fisica per il byte a cui stiamo accedendo.
- Dodici bit sono usati come offset all'interno della pagina. L'indirizzo fisico del byte a cui stiamo accedendo è costruito aggiungendo i 12 bit più bassi dell'indirizzo virtuale alla fine del numero di pagina fisica che abbiamo trovato nel pte selezionato.

Il numero di bit in un indirizzo fisico può essere diverso dal numero di bit in un indirizzo virtuale. Nell'architettura ia-32 originale, il pte e il pde erano strutture a 32 bit che facevano spazio a 20 bit di numero di pagina fisica: la dimensione dell'indirizzo fisico e la dimensione degli indirizzi virtuali erano quindi le stesse e tali sistemi potevano indirizzare solo 4 gb di memoria fisica. Più tardi, la ia-32 fu estesa con l'introduzione dei pte a 64 bit utilizzati ancora oggi e l'hardware iniziò a supportare numeri di pagina fisica a 24 bit. Questi sistemi sono in grado di gestire 64 gb di memoria ed erano utilizzati su sistemi server. Oggi tutti i server Windows si basano su amd64 o ia64 e supportano indirizzi fisici molto grandi, molto più grandi di quanto sia possibile utilizzare: ricordate il fatto che una volta 4 gb di memoria fisica sembravano ottimisticamente troppi.

Per migliorare le prestazioni il gestore della vm mappa la directory delle pagine e le pagine della tabella dei pte nella stessa regione contigua di indirizzi virtuali in ogni processo. Questa auto-mappa (self-map) permette al gestore di utilizzare lo stesso puntatore per accedere al pde o al pte corrente, corrispondente a un particolare indirizzo virtuale, indipendentemente dal processo che è in esecuzione. In ia-32 l'auto-mappa utilizza una regione contigua da 8 mb dello spazio di indirizzamento virtuale del kernel, mentre in amd64 la mappa occupa 512 gb. Nonostante il significativo spazio di indirizzamento occupato, questa mappa non richiede alcuna pagina di memoria virtuale aggiuntiva; inoltre, consente alle pagine della tabella delle pagine di essere automaticamente spostate dentro e fuori dalla memoria fisica.

Nella creazione di un'auto-mappa, uno dei pde della directory delle pagine di livello superiore fa riferimento alla pagina stessa, formando un "ciclo" nella traduzione della tabella delle pagine. Quando non si dà percorre il ciclo si effettua un accesso alle pagine virtuali, quando il ciclo viene percorso una volta si accede alle pagine della tabella dei pte, quando il ciclo viene percorso due volte si accede alle pagine della directory delle pagine di livello più basso, e così via.

I livelli aggiuntivi delle directory delle pagine utilizzati per la memoria virtuale a 64 bit sono tradotti nello stesso modo, eccetto per il fatto che il puntatore all'indirizzo virtuale viene suddiviso in ancor più valori. Nel caso di amd64, Windows utilizza quattro livelli pieni, ognuno dei quali mappa 512 pagine, o $9+9+9+12 = 48$ bit di indirizzamento virtuale.

Per evitare i rallentamenti dovuti alla traduzione degli indirizzi virtuali tramite la ricerca dei pde e pte appropriati, i processori adottano uno speciale hardware detto **tlb** (*translation look-aside buffer*), contenente una memoria cache associativa per mappare pagine virtuali sui pte. Il tlb fa parte dell'unità di gestione della memoria (mmu) all'interno di ciascun processore. La mmu ha bisogno di percorrere la tabella delle pagine (le sue strutture dati) in memoria solo quando una traduzione è necessaria, perché manca nel tlb.

I pde e i pte non contengono solo i numeri di pagina fisica, ma hanno anche alcuni bit riservati per l'uso del sistema operativo e alcuni bit per controllare come l'hardware utilizzi la memoria, per esempio per controllare se la cache hardware debba essere utilizzata per ogni pagina. Inoltre le voci specificano il tipo di accesso consentito per le modalità kernel e utente.

Un pde può essere contrassegnato per segnalare che dovrebbe funzionare come pte piuttosto che come pde. Su ia-32 i primi 11 bit del puntatore di indirizzo virtuale selezionano un pde nei primi due livelli di traduzione. Se il pde selezionato è contrassegnato per fungere da pte i rimanenti 21 bit del puntatore sono usati come offset del byte. Ciò porta a una dimensione di pagina di 2 mb. Mischiare e far corrispondere pagine di 4 kb e 2 mb all'interno della tabella delle pagine è facile per il sistema operativo e può

migliorare significativamente le prestazioni di alcuni programmi, riducendo la frequenza con cui la mmu ha bisogno di ricaricare le voci nel tlb, visto che un pde di 2 mb sostituisce 512 pte di 4 kb.

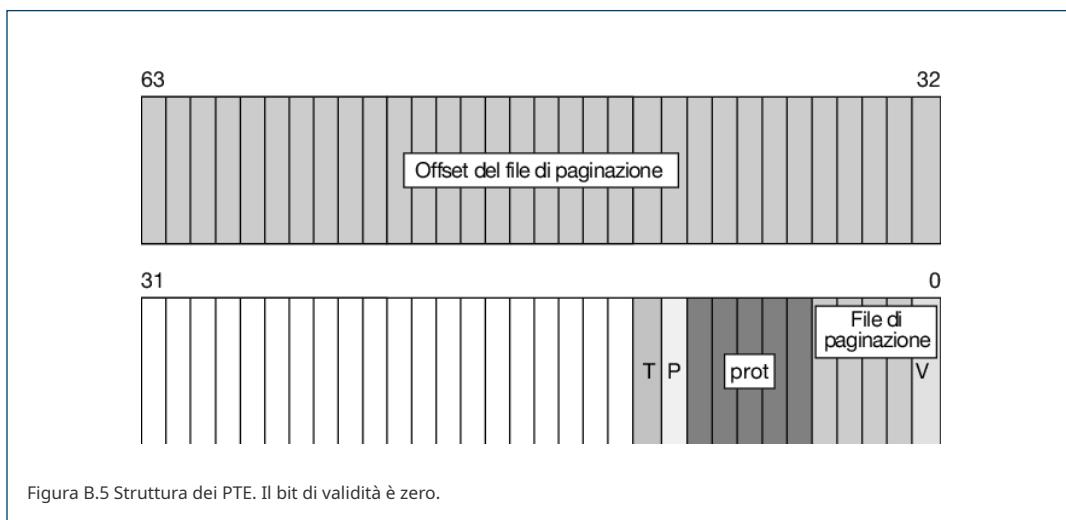
Gestire la memoria fisica in modo che le pagine di 2 mb siano disponibili al bisogno è tuttavia difficile, in quanto le pagine possono continuamente essere suddivise in pagine di 4 kb, provocando frammentazione esterna della memoria. Inoltre, le pagine grandi possono causare una frammentazione interna piuttosto significativa. A causa di questi problemi, sono in genere esclusivamente lo stesso Windows e le applicazioni per server di grandi dimensioni a utilizzare pagine grandi per migliorare le prestazioni del tlb. Essi sono più adatti a farlo, perché il sistema operativo e le applicazioni server iniziano l'esecuzione all'avvio del sistema, prima che la memoria sia diventata frammentata.

Windows gestisce la memoria fisica associando ciascuna pagina fisica a uno dei sette stati seguenti: libera, azzerata, modificata, in attesa, guasta, in transizione o valida.

- Una pagina libera non ha un contenuto particolare.
- Una pagina azzerata è libera, i suoi contenuti sono stati inizializzati a zero ed è disponibile per soddisfare richieste di pagine *zero-on-demand*.
- Una pagina modificata è stata modificata (tramite scrittura) da un processo e deve essere trasferita su disco prima di essere allocata a un altro processo.
- Una pagina in attesa è una copia d'informazioni già presenti su disco. Si può trattare di una pagina non ancora modificata, di una pagina già scritta su disco o di una pagina caricata anticipatamente in memoria perché ci si aspetta di utilizzarla al più presto.
- Una pagina guasta è inutilizzabile a causa di un errore hardware.
- Una pagina in transizione è in corso di trasferimento dal disco a un frame allocato nella memoria fisica.
- Una pagina valida è parte del working set di uno o più processi ed è contenuta all'interno delle tabelle delle pagine di questi processi.

Mentre le pagine valide sono contenute nelle tabelle delle pagine dei processi, le pagine in altri stati sono mantenute in liste distinte in base al tipo di stato. Le liste sono costruite associando le voci corrispondenti nel database pfn (page frame number, *numero di frame di pagina*), che contiene una voce per ogni pagina di memoria fisica. Le voci pfn includono anche informazioni come il numero di riferimenti, i lock e le informazioni numa. Si noti che il database pfn rappresenta pagine di memoria fisica, mentre il pte rappresenta pagine di memoria virtuale.

Se il bit di validità di un pte è a zero, l'hardware ignora gli altri bit e il gestore della vm può definirne il formato per il proprio uso. Le pagine non valide si possono trovare in un certo numero di stati rappresentati da bit nel pte. Le pagine dei file di paginazione mai caricati in seguito a un fault sono marcate *zero-on-demand*; le pagine mappate tramite oggetti sezione codificano un puntatore all'oggetto in questione; le pagine scritte sul file di paginazione contengono informazioni sufficienti per reperire la pagina sul disco, e così via. La struttura dei pte è mostrata nella Figura B.5. Per questo tipo di pte i bit T, P e V sono tutti a zero. Il pte contiene 5 bit dedicati alla protezione della pagina, 32 bit per l'offset all'interno del file di paginazione e 4 bit per selezionare il file di paginazione. Ci sono anche 20 bit riservati per ulteriori operazioni.



Windows utilizza una politica di sostituzione lru sui working set per prelevare le pagine dai processi in maniera appropriata. Quando si avvia un processo, gli viene assegnata una dimensione minima predefinita del working set. Il working set di ogni processo può crescere fino a quando la quantità residua di memoria fisica comincia a scarseggiare; a quel punto il gestore della vm inizia a tracciare l'età delle pagine in ogni working set. Infine, se la memoria disponibile è quasi esaurita, il gestore della vm taglia il working set rimuovendo le pagine più vecchie.

L'età di una pagina non dipende da quanto tempo è stata in memoria, ma dall'ultimo riferimento alla pagina, e viene determinata scandendo periodicamente il working set di ogni processo e incrementando l'età delle pagine che dall'ultima passata non sono state

evidenziate nel pte come referenziate. Quando diventa necessario tagliare il working set, il gestore della vm utilizza euristiche per decidere quanto tagliare da ogni processo e quindi rimuove le pagine a partire dalle meno recenti.

È possibile che il working set di un processo venga tagliato anche quando resta disponibile molta memoria. Ciò avviene se è stato impostato un limite rigido sulla quantità di memoria fisica che il processo può usare. In Windows 7 il gestore della vm è in grado di tagliare anche i processi che stanno crescendo rapidamente, anche se la memoria è abbondante. Questa politica migliora significativamente la capacità di risposta del sistema sugli altri processi.

Windows tiene traccia dei working set non solo per i processi in modalità utente, ma anche per il processo di sistema, che comprende tutte le strutture dati paginabili e il codice che viene eseguito in modalità kernel. Windows 7 crea working set supplementari per il processo di sistema e li associa a particolari categorie di memoria del kernel. La cache dei file, l'heap del kernel e il codice del kernel hanno in Windows 7 i loro working set. La presenza di working set distinti consente al gestore della vm di utilizzare diverse politiche per tagliare le diverse categorie di memoria del kernel.

A seguito di un errore per page fault, il gestore della vm non carica solo la pagina immediatamente necessaria, perché vi è evidenza a supporto della tesi che i riferimenti in memoria seguano un principio di località: se una pagina è usata è probabile che lo siano anche le pagine adiacenti nel prossimo futuro. (Si pensi alle iterazioni per scandire un array, o al reperimento sequenziale delle istruzioni che costituiscono il codice di un thread). Per questi motivi, il gestore carica in memoria la pagina richiesta, insieme però a un certo numero di pagine adiacenti; ciò tende a ridurre il numero totale di page fault. Inoltre, le operazioni di lettura sono accoppiate per migliorare le prestazioni.

Oltre a occuparsi della memoria impegnata, il gestore della vm si prende anche cura della memoria riservata da ogni processo, ossia del suo spazio degli indirizzi virtuali. Ciascun processo ha un albero associato che specifica la gamma di indirizzi virtuali usati, e la loro finalità. Su questa base, il gestore della vm può caricare le pagine mancanti man mano che servono: se il pte non è inizializzato, il gestore cerca l'indirizzo interessato all'interno dell'albero dei descrittori di indirizzi virtuali (*virtual address descriptor*, vad), e usa l'informazione così reperita per creare il pte mancante e individuare la pagina. In qualche caso potrebbe non esistere la pagina della tabella dei pte, quindi il gestore dovrà allocarla e inizializzarla in maniera trasparente. In altri casi, la pagina potrebbe essere condivisa come parte di un oggetto sezione e il vad conterrà un puntatore a quell'oggetto sezione. L'oggetto sezione contiene informazioni su come trovare la pagina virtuale condivisa in modo che il pte possa essere inizializzato per puntare direttamente a questa.

B.3.3.3 Gestore dei processi

Il gestore dei processi del sistema Windows fornisce i servizi necessari a creazione, eliminazione e uso dei thread e dei processi. Esso non possiede alcuna informazione sulle relazioni parentali o sulle gerarchie fra i processi; questi dettagli sono lasciati al particolare sottosistema d'ambiente relativo al processo. Il gestore dei processi non è coinvolto neppure nello scheduling dei processi, fuorché per determinare le priorità e le affinità dei processi e dei thread, quando essi sono creati. Lo scheduling dei thread ha luogo nel dispatcher del kernel.

Ciascun processo contiene uno o più thread. I processi medesimi possono essere raccolti in grosse unità, chiamate oggetti job; il ricorso a oggetti job permette di imporre limiti all'utilizzo della cpu e alla dimensione dei working set e di definire affinità per i processore che controllano più processi in una volta. Gli oggetti job sono usati per gestire le potenti macchine dei centri di elaborazione dati.

I seguenti passi fungono da esempio per la creazione di un processo in Win32.

1. Un'applicazione di Win32 richiama `CreateProcess()`.
2. Viene inviato un messaggio al sottosistema Win32 per notificare la creazione del processo.
3. `CreateProcess()`, nel processo originale, richiama una api nel gestore dei processi dell'executive per creare effettivamente il processo.
4. Il gestore dei processi richiama il gestore degli oggetti per creare un oggetto processo, e restituisce l'handle a Win32;
5. Win32 richiama di nuovo il gestore del processo al fine di costituire un thread per il processo e restituisce gli handle relativi al processo e al thread.

Le api di Windows che manipolano la memoria virtuale e i thread, e che duplicano gli handle, accettano come parametro in ingresso un handle del processo, in modo che i sottosistemi possano eseguire operazioni per conto di un nuovo processo senza dover eseguire direttamente nel contesto del nuovo processo. Non appena è creato un nuovo processo, si genera il thread iniziale; una chiamata di procedura asincrona è recapitata al thread per avviare l'esecuzione del caricatore dell'immagine in modalità utente. Il caricatore è nella `ntdll.dll`, una libreria di collegamento dinamico mappata automaticamente su ogni processo creato. Per la generazione dei processi, Windows fornisce altresì una chiamata `fork()`, di derivazione unix, allo scopo di supportare il sottosistema d'ambiente posix. Sebbene l'ambiente Win32 contatti il gestore dei processi dal processo client, la natura inter-processo delle api di Windows consente a posix la creazione del nuovo processo, direttamente dal processo del sottosistema in cui agisce.

Il gestore dei processi si basa sulle chiamate di procedura asincrone (apc) implementate dal livello kernel. Le apc vengono utilizzate per iniziare l'esecuzione dei thread, per sospendere e riprendere thread, per accedere ai registri dei thread, per terminare thread e processi e per il supporto dei debugger.

Il supporto del debugger da parte del gestore dei processi include la possibilità di sospendere e riavviare i thread, e di creare thread che iniziano l'esecuzione sospendendosi. Vi sono anche api del gestore dei processi in grado di leggere e impostare il contesto del registro di un thread e accedere alla memoria virtuale di un altro processo. I thread possono essere creati nel processo corrente, ma possono anche essere iniettati in un processo diverso. Il debugger fa uso di quest'ultimo meccanismo per eseguire codice all'interno di un processo in fase di debug.

Nell'ambito dell'executive, i thread esistenti possono aggregarsi temporaneamente a un altro processo: questo metodo è usato dai thread di lavoro che devono operare nel contesto del processo da cui proviene una richiesta di lavoro. Per esempio, il gestore di vm potrebbe utilizzare un thread aggregato quando è necessario l'accesso al working set o alla tabella delle pagine di un processo e il gestore di i/o potrebbe utilizzarlo per aggiornare la variabile di stato in un processo per le operazioni di i/o asincrono.

Il gestore prevede anche la impersonazione. A ogni thread è associato un **contrassegno di sicurezza** (*security token*). Quando il processo di login autentica un utente il security token viene collegato al processo dell'utente ed ereditato dai suoi processi figlio. Il token contiene il sid (*security identity*, identità di sicurezza) dell'utente, i sid dei gruppi cui appartiene l'utente, i privilegi dell'utente e il livello di integrità del processo. Per impostazione predefinita tutti i thread all'interno di un processo condividono un token comune che rappresenta l'utente e l'applicazione che ha avviato il processo. Tuttavia un thread in esecuzione in un processo con un security token appartenente a un utente può attivare un token specifico appartenente a un altro utente per impersonare tale utente.

La personificazione è fondamentale per il modello rpc client-server, dove i servizi devono agire per conto di una varietà di client con diversi id di sicurezza. Il diritto di rappresentare un utente è spesso fornito come parte di una connessione rpc da un processo client a un processo server. La impersonazione consente al server di accedere ai servizi di sistema come se fosse il client in modo da poter accedere a oggetti e file o creare per conto del client. Il processo server deve essere affidabile e deve essere scritto con molta attenzione per poter essere robusto contro gli attacchi. In caso contrario un client potrebbe subentrare a un processo server e impersonare quindi qualsiasi utente che in seguito effettuerà una richiesta.

B.3.3.4 Servizi per la computazione client-server

L'implementazione di Windows utilizza un modello client-server. I sottosistemi d'ambiente sono server che implementano specifiche personalità del sistema operativo. Oltre a ciò, il medesimo modello è applicato alla realizzazione di tutta una gamma di servizi del sistema: l'autenticazione degli utenti, i servizi di rete, la gestione delle code di stampa, i servizi web, i file system di rete, il *plug-and-play*. Per moderare la memoria necessaria, molti servizi sono spesso accoppati all'interno di pochi processi che eseguono il programma `svchost.exe`. Ogni servizio viene caricato come una libreria dinamica (dll) che realizza il servizio sfruttando le potenzialità dei pool di thread in modalità utente per condividere thread e ricevere messaggi (si veda il Paragrafo B.3.3.3).

Il paradigma normale per realizzare una computazione client-server consiste nell'utilizzo di rpc per comunicare le richieste. L'api Win32 supporta un protocollo rpc standard, come descritto nel Paragrafo B.6.2.7. rpc utilizza diversi meccanismi di trasporto (per esempio, NamedPipes e tcp/ip) e può essere utilizzato per implementare rpc tra sistemi diversi. Quando una rpc viene sempre eseguita tra un client e un server sullo stesso sistema locale, per il trasporto può essere utilizzata la chiamata di procedura locale avanzata (alpc). Al livello più basso del sistema, nell'implementazione dei sistemi di ambiente, e per i servizi che devono essere disponibili nelle fasi di avvio, rpc non è disponibile. I servizi nativi di Windows utilizzano in questi casi direttamente le alpc.

Una alpc è un meccanismo per lo scambio dei messaggi. Il processo server pubblica un oggetto globalmente visibile che rappresenta una porta di connessione: quando un client necessita di un servizio da un sottosistema, apre un handle per l'oggetto porta di connessione del sottosistema e trasmette una richiesta di connessione alla porta in questione; il server crea un canale e restituisce un handle al client. Il canale consiste in una coppia di porte di comunicazione private: una per i messaggi dal client al server, l'altra per quelli dal server al client. I canali di comunicazione forniscono un meccanismo di richiamata che permette al client e al server di accettare richieste anche quando attendono una risposta.

Al momento della creazione di un canale alpc è necessario specificare una fra tre possibili tecniche di scambio dei messaggi.

1. La prima tecnica è adatta a messaggi brevi o di lunghezza media (fino a 63 kb): in questo caso, si usa la coda dei messaggi della porta come mezzo di memorizzazione intermedio, e si copiano i messaggi da un processo all'altro.
2. La seconda tecnica è adatta a messaggi più lunghi: in questo caso, si crea un oggetto sezione di memoria condivisa per il canale; i messaggi trasmessi attraverso la coda dei messaggi della porta contengono un puntatore alla sezione, oltre a informazioni sulla dimensione dell'oggetto sezione. In questo modo si evita la necessità di copiare lunghi messaggi: il mittente inserisce dati nella sezione condivisa e il ricevente può vederli immediatamente.
3. La terza tecnica sfrutta le api che leggono e scrivono direttamente nello spazio degli indirizzi di un processo. Il meccanismo alpc mette a disposizione dei server funzioni e strumenti di sincronizzazione che permettono di accedere ai dati del client. Questa tecnica è solitamente utilizzata dalle rpc per ottenere prestazioni migliori in alcuni scenari specifici.

Il gestore delle finestre di Win32 gestisce la trasmissione dei messaggi in modo diverso e indipendente dall'alpc. Se un client richiede una connessione con trasmissione di messaggi basata sul gestore delle finestre, il server predispone tre oggetti: (1) un thread del server dedicato alla gestione delle richieste, (2) un oggetto sezione da 64 kb, e (3) un cosiddetto *oggetto coppia di eventi*. Quest'ultimo è un oggetto per la sincronizzazione che il sottosistema Win32 usa per notificare che il thread client ha copiato un messaggio verso il server Win32, o viceversa. L'oggetto sezione viene utilizzato per passare i messaggi, e l'oggetto coppia di eventi esegue la sincronizzazione.

Questo meccanismo per lo scambio di messaggi del gestore delle finestre offre molti vantaggi.

- L'oggetto sezione elimina la necessità di copiare messaggi, poiché rappresenta una regione di memoria condivisa.
- L'oggetto coppia di eventi elimina i rallentamenti legati all'uso di oggetti porta per passare i messaggi contenenti puntatori e lunghezze.
- Il thread server dedicato elimina i rallentamenti legati all'identificazione del thread client che richiede il servizio, visto che vi è un thread server per ciascun thread client.
- Il kernel avvantaggia lo scheduling di questi thread server dedicati, per migliorare le prestazioni del sistema.

B.3.3.5 Gestore dell'i/o

Il gestore dell'i/o è responsabile della gestione dei file system, dei driver dei dispositivi e dei driver di rete: tiene traccia dei driver e dei file system caricati e gestisce i buffer per le richieste di i/o; collabora con il gestore della vm per eseguire l'i/o dei file mappati in memoria, e controlla il gestore della cache di Windows che, a sua volta, gestisce i servizi di caching per l'intero sistema di i/o. Il gestore di i/o è sostanzialmente asincrono; l'i/o sincrono è realizzato tramite l'attesa esplicita del completamento dell'operazione. Il gestore di i/o fornisce numerosi modelli di completamento asincrono dell'i/o, compresa l'impostazione di eventi, l'aggiornamento di una variabile di stato nel processo chiamante, l'invio di apc al thread che ha avviato l'operazione, e le porte di completamento di i/o, che permettono a un singolo thread di gestire il completamento dell'i/o di molti thread.

I driver dei dispositivi sono organizzati, per ciascun dispositivo, in una lista detta stack di i/o del dispositivo. Un driver viene rappresentato nel sistema come oggetto driver. Visto che un singolo driver può operare su più dispositivi, i driver vengono rappresentati sullo stack dell'i/o mediante un oggetto dispositivo che contiene un link all'oggetto driver. Il gestore di i/o converte le richieste ricevute in una forma standard, detta **pacchetto di richiesta di i/o** (*i/o request packet*, irp), che inoltra al primo driver nello

stack affinché sia elaborato. Dopo che un driver ha elaborato l'irp, richiama il gestore di i/o per inoltrare il pacchetto al driver successivo nello stack o, se l'elaborazione è terminata, per completare l'operazione sull'irp.

Il completamento può avvenire in un contesto differente dalla richiesta originale di i/o. Per esempio, se un driver sta eseguendo la propria parte di un'operazione di i/o ed è obbligato a bloccarsi per lungo tempo, può accadere l'irp a un thread di lavoro per proseguire l'elaborazione nel contesto del sistema. Al thread originale il driver restituisce un indicatore di stato per comunicare che la richiesta di i/o è in corso, in modo che possa continuare l'esecuzione in parallelo con l'operazione di i/o. I pacchetti irp possono anche essere elaborati da procedure di servizio delle interruzioni; l'elaborazione può terminare in un contesto di processo arbitrario. Poiché una qualche elaborazione finale potrebbe essere necessaria nel contesto che ha iniziato l'i/o, il gestore di i/o usa una apc per eseguire l'elaborazione finale, a completamento dell'i/o, nel contesto del thread che ha dato l'avvio all'operazione.

Questo modello a stack è molto flessibile. Quando si costruisce uno stack, vari driver hanno l'opportunità di inserirsi nello stack come driver filtro. I driver filtro sono in grado di esaminare e potenzialmente modificare ciascuna operazione di i/o. La gestione del montaggio, delle partizioni, le operazioni di scomposizione in sezioni e mirroring del disco sono altrettanti esempi di funzionalità implementate usando i driver filtro che operano nello stack, al di sotto del file system. I driver filtro del file system agiscono al di sopra del file system e sono stati usati per realizzare funzionalità quali la gestione della memorizzazione gerarchica, la creazione di singole istanze di file per l'avvio remoto e la conversione dinamica dei formati. Terze parti usano anche i driver filtro del file system per implementare il rilevamento dei virus.

I driver dei dispositivi di Windows sono scritti in conformità alle specifiche del "modello di driver di Windows" (*Windows driver model*, wdm). Esso stabilisce i requisiti del driver del dispositivo, incluse le modalità di stratificazione dei driver filtro, la condivisione del codice comune per gestire l'alimentazione e le richieste *plug-and-play*, la costruzione della corretta logica di cancellazione, e così via.

A causa della ricchezza del modello, scrivere un driver wdm completo per ogni periferica aggiunta può rivelarsi un compito improbo. Fortunatamente ciò non è necessario, grazie al cosiddetto modello porta/miniporta. Ciascuna istanza di un dispositivo che rientri in una classe di dispositivi simili, quali i driver audio, i dispositivi sata e i controllori Ethernet, condivide un driver comune per quella classe, chiamato driver della porta. Il driver della porta implementa le operazioni standard per la classe di appartenenza e richiama poi le procedure specifiche del dispositivo nel driver della miniporta di quel dispositivo, al fine di realizzare le proprie funzionalità peculiari. Lo stack tcp/ip è implementato in questo modo, con il driver `ndis.sys` che implementa molte funzionalità dei driver di rete e chiama i driver delle miniporte di rete per lo specifico hardware.

Le versioni recenti di Windows, tra cui Windows 7, forniscono ulteriori semplificazioni per la scrittura di driver per dispositivi hardware. I driver in modalità kernel possono essere scritti utilizzando l'ambiente kmdf (*kernel-mode driver framework*) che fornisce un modello di programmazione semplificato per i driver su wdm. Un'altra opzione è l'ambiente umdf (*user-mode driver framework*). Molti driver non hanno bisogno di operare in modalità kernel ed è più facile sviluppare e distribuire i driver in modalità utente. L'utilizzo di driver in modalità utente rende inoltre il sistema più affidabile, in quanto un errore di un driver in modalità utente non causa un crash in modalità kernel.

B.3.3.6 Gestore della cache

In molti sistemi operativi la funzionalità di cache è di pertinenza del file system. Windows, invece, offre un servizio di cache centralizzata in cui il gestore della cache opera in stretta collaborazione con il gestore della vm, per fornire servizi di cache a tutti i componenti del sistema sotto il controllo del gestore di i/o. Il caching in Windows è basata sui file, e non sui blocchi di basso livello. La dimensione della cache cambia dinamicamente, in base alla quantità di memoria libera disponibile nel sistema. Il gestore della cache mantiene un proprio working set piuttosto che condividere quello del processo di sistema. Il gestore della cache mappa i file nella memoria del kernel e quindi utilizza interfacce speciali verso il gestore della vm per aggiungere o tagliare pagine da questo working set privato.

La cache è divisa in blocchi da 256 kb, ognuno dei quali può contenere una vista di un file, cioè una parte del file mappata in memoria. Ciascun blocco della cache è descritto da un blocco di controllo dell'indirizzo virtuale (vacb), che memorizza l'indirizzo virtuale e l'offset della vista nel file, così come il numero di processi che stanno usando la vista. I vacb risiedono in un singolo array mantenuto dal gestore della cache.

Quando il gestore dell'i/o riceve una richiesta di lettura di un file a livello utente invia un irp allo stack di i/o per il volume su cui risiede il file. Per i file contrassegnati come memorizzabili nella cache, il file system chiama il gestore della cache per cercare i dati richiesti nelle sue viste dei file nella cache. Il gestore della cache calcola quale elemento dell'array vacb di quel file corrisponda all'offset in byte codificato nella richiesta. O l'elemento in questione punta alla vista interessata nella cache, oppure è nullo; in tal caso, il gestore della cache alocca un blocco della cache, insieme all'elemento corrispondente dell'array vacb, e mappa la vista nel nuovo blocco della cache. Esso tenta poi di copiare i dati dal file mappato al buffer del chiamante; se la copia riesce, l'operazione è completata.

Se la copia fallisce, ciò avviene a causa di una pagina mancante, che spinge il gestore della vm a inviare al gestore di i/o una richiesta di lettura con esplicita esclusione della cache; questi trasmette un'ulteriore richiesta allo stack dei driver del dispositivo per ottenere, questa volta, un'operazione di *paginazione*, che esclude il gestore della cache e legge i dati direttamente dal file nella pagina allocata per il gestore della cache: completata l'operazione, il vacb è impostato in modo da puntare a quella pagina. I dati, ora nella cache, sono copiati nel buffer del chiamante, cosa che completa l'originaria richiesta di i/o. La Figura B.6 riassume graficamente queste operazioni.

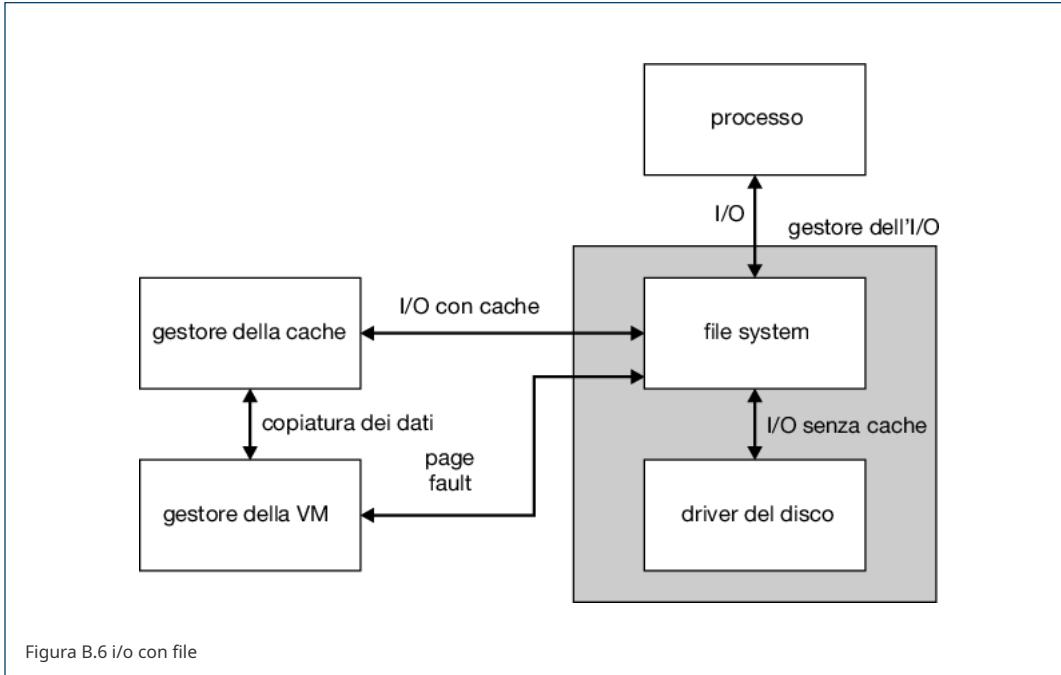


Figura B.6 i/o con file

Un'operazione di lettura a livello del kernel si svolge in modo simile, a eccezione del fatto che si può accedere ai dati direttamente dalla cache, anziché copiarli in un buffer nello spazio dell'utente. Per leggere i metadati del file system – le strutture dati che descrivono il file system – il kernel si serve dell'interfaccia del gestore della cache dedicata alla mappatura; per modificarli, usa invece l'interfaccia del gestore della cache per ancorare le pagine. Il **pinning** (cioè l'*ancoraggio*) di una pagina significa associarla a un frame della memoria fisica, in modo che il gestore della vm non possa spostarla o scaricarla. Dopo l'aggiornamento dei metadati, il file system richiede al gestore della cache di liberare la pagina ancorata. Una pagina modificata viene etichettata come *dirty*, sicché il gestore della vm ricopia la pagina su disco.

Per migliorare le prestazioni, il gestore della cache mantiene un piccolo archivio delle richieste di lettura, con cui cerca di predire le richieste successive. Se il gestore rileva uno schema predicable nelle tre precedenti richieste, per esempio un accesso sequenziale in avanti o all'indietro, trasferisce i dati nella cache in anticipo, prima che giunga la successiva richiesta da parte dell'applicazione. In questo modo l'applicazione può trovare i propri dati già nella cache senza dover aspettare l'i/o del disco.

Altro compito del gestore della cache è di ordinare al gestore della vm lo svuotamento del contenuto della cache. Per default, il gestore della cache opera secondo la tecnica della scrittura differita (*write-back caching*): accumula scritture per 4-5 secondi e poi risveglia il thread di trasferimento su disco. Quando l'operazione deve invece seguire la **tecnica della scrittura diretta** (*write-through policy*), un processo può segnalarlo, impostando un flag all'apertura del file, o può richiamare un'esplicita funzione di svuotamento della cache.

Un processo di scrittura veloce potrebbe arrivare a riempire tutte le pagine libere della cache prima che il thread di trasferimento abbia la possibilità di attivarsi per scaricare le pagine su disco; il thread previene tale eventualità nel modo seguente. Quando la quantità di memoria libera nella cache si riduce, il gestore della cache sospende temporaneamente i processi che tentano di scrivere dati e risveglia il thread di trasferimento per scaricare le pagine su disco. Se il processo di scrittura implementa in realtà un reindirizzamento per un file system di rete, congelarlo troppo a lungo potrebbe far scadere i trasferimenti di rete e indurre la ritrasmissione dei dati; pertanto, si sprecherebbe banda di rete. Per impedire tale spreco, i reindirizzatori di rete possono chiedere al gestore della cache di limitare le scritture accumulate nella cache.

Poiché un file system di rete necessita di trasferire i dati fra un disco e l'interfaccia di rete, il gestore della cache fornisce anche un'interfaccia dma per spostare i dati direttamente. Il trasferimento diretto evita la duplicazione dei dati attraverso un buffer intermedio.

B.3.3.7 Monitor della sicurezza dei riferimenti

Grazie alla centralizzazione della gestione delle entità del sistema nel gestore degli oggetti, Windows può realizzare un meccanismo uniforme per la validazione run-time degli accessi e per le verifiche di sicurezza per ogni elemento del sistema accessibile dagli utenti. Ogniqualvolta un processo apre un oggetto, il **monitor della sicurezza dei riferimenti** (*security reference monitor*, srm) controlla il token di sicurezza del processo e la lista di controllo degli accessi dell'oggetto al fine di determinare se il processo ha i diritti necessari.

L'srm è anche responsabile della manipolazione dei privilegi nei token di sicurezza. Gli utenti che debbano eseguire backup o operazioni di ripristino del file system, che debbano eseguire il debug di processi, e così via, necessitano di privilegi speciali. È anche possibile impostare un token di modo che neghi l'accesso a elementi solitamente disponibili per la gran parte degli utenti; ciò è utile soprattutto per limitare i danni che l'esecuzione di codice di provenienza incerta potrebbe causare.

Il livello di integrità del codice in esecuzione in un processo è rappresentato anche da un token. I livelli di integrità sono un meccanismo per gestire le capability di un processo, come accennato in precedenza. Un processo non può modificare un oggetto con un livello di integrità superiore a quello del codice in esecuzione nel processo, indipendentemente dalle altre autorizzazioni concesse. I livelli di integrità sono stati introdotti per rendere più difficile l'acquisizione del controllo sul sistema da parte di codice che attacca con successo il software in grado di collegarsi verso l'esterno, come Internet Explorer.

Un altro compito dell'srm è la registrazione su log degli eventi di verifica della sicurezza. Il Common Criteria del Dipartimento della Difesa americano (il successore dell'Orange Book, pubblicato nel 2005) richiede che un sistema sicuro debba poter rilevare e registrare tutti i tentativi d'accesso alle proprie risorse, al fine di facilitare l'identificazione dei tentativi d'accesso non autorizzato. Poiché l'srm è responsabile dei controlli d'accesso, esso genera la gran parte delle informazioni di verifica nel log della sicurezza.

B.3.3.8 Gestore del plug-and-play

Il sistema operativo usa il gestore plug-and-play (PnP) per riconoscere eventuali cambiamenti nella configurazione hardware del calcolatore e adattarvi opportunamente il sistema. I dispositivi PnP utilizzano protocolli standard per farsi identificare dal sistema. Il gestore PnP riconosce automaticamente i dispositivi installati e mentre il sistema è funzionante rileva eventuali cambiamenti nei dispositivi. Il gestore tiene anche traccia delle risorse impiegate da ciascun dispositivo, insieme con le risorse che potenzialmente potrebbero essere utilizzate e si occupa del caricamento dei driver necessari. Questa gestione delle risorse fisiche (che sono principalmente le interruzioni e gli intervalli di memoria per l'i/o) ha l'obiettivo d'identificare una configurazione hardware che permetta a tutti i dispositivi di funzionare correttamente.

Il gestore PnP opera la riconfigurazione dinamica come segue. Innanzitutto ottiene una lista di dispositivi da ciascun driver di bus (per esempio, pci, usb). Successivamente carica il driver installato (se è necessario, dopo averne trovato uno) e invia un comando `add-device` al driver appropriato per ciascun dispositivo. Il gestore PnP trova poi le assegnazioni ottimali delle risorse e invia un comando `start-device` a ciascun driver per specificare l'assegnamento delle risorse per quel dispositivo. Se un dispositivo deve essere riconfigurato il gestore PnP invia un comando `query-stop` che richiede al driver se il dispositivo può essere temporaneamente disabilitato. Se il driver può disabilitare il dispositivo, allora tutte le operazioni sospese vengono completate e non si può avviare alcuna nuova operazione. Successivamente, il gestore PnP invia un comando `stop`, dopodiché può riconfigurare il dispositivo con un altro comando `start-device`.

Il gestore PnP prevede altri comandi, come `query-remove`, che si usa quando l'utente sta per rimuovere un dispositivo removibile, come una memoria usb, e che opera in modo simile a `query-stop`. Il comando `surprise-remove` si usa quando c'è un malfunzionamento di dispositivo, oppure, più comunemente, quando un utente estra un dispositivo senza prima comunicarlo al sistema. Il comando `remove`, infine, richiede che il driver smetta di usare il dispositivo in maniere permanente.

Molti programmi presenti nel sistema sono interessati all'aggiunta o alla rimozione dei dispositivi e per questa ragione il gestore PnP offre funzionalità di notifica. Per esempio, una notifica fornisce ai menu dei file della gui le informazioni di cui hanno bisogno per aggiornare l'elenco dei volumi del disco quando un nuovo dispositivo di archiviazione viene collegato o rimosso. L'installazione di periferiche si traduce spesso in aggiunta di nuovi servizi ai processi `svchost.exe` nel sistema. In molti casi questi servizi vengono eseguiti ogni volta che il sistema si avvia e continuano a funzionare anche se il dispositivo originale non viene mai inserito nel sistema. Windows 7 ha introdotto un meccanismo di attivazione dei servizi nel gestore scm (*service control manager*), che gestisce i servizi di sistema. Mediante questo meccanismo i servizi possono essere configurati per essere eseguiti solo quando scm riceve una notifica dal gestore PnP che segnala che il dispositivo di interesse è stato aggiunto al sistema.

B.3.3.9 Gestore dell'alimentazione

Windows collabora con l'hardware per implementare sofisticate strategie per l'efficienza energetica, come descritto nel Paragrafo B.2.8. Le politiche che guidano queste strategie sono attuate dal **gestore dell'alimentazione** (*power manager*). Il gestore dell'alimentazione rileva le condizioni del sistema, come il carico della cpu o dei dispositivi di i/o, e migliora l'efficienza energetica riducendo le prestazioni e la reattività del sistema quando le richieste sono basse. Il gestore dell'alimentazione può anche mettere l'intero sistema nella modalità *sleep*, molto efficiente, o anche scrivere tutti i contenuti della memoria su disco e spegnere l'alimentazione per consentire al sistema di passare in uno stato di *sospensione*, o *ibernazione* (*hyibernation*).

Il vantaggio principale dello *sleep* è la rapidità con cui è possibile passare in questo stato: spesso sono sufficienti pochi secondi dopo aver chiuso il coperchio di un computer portatile. Anche il ripristino dallo *sleep* è abbastanza veloce. Nello stato di *sleep* viene tolta l'alimentazione a cpu e dispositivi di i/o, ma la memoria continua a essere alimentata a sufficienza per non perderne il contenuto.

La sospensione richiede molto più tempo, perché l'intero contenuto della memoria deve essere trasferito sul disco prima che il sistema venga spento. Tuttavia, il fatto che il sistema sia realmente spento costituisce un vantaggio significativo. Se l'alimentazione del sistema viene persa, per esempio quando viene sostituita la batteria su un computer portatile o quando un sistema desktop viene scollegato dalla rete elettrica, i dati memorizzati non verranno persi. A differenza dello spegnimento, la sospensione salva il sistema attualmente in uso, in modo che un utente possa riprendere da dove aveva lasciato. Poiché la sospensione non richiede alimentazione, un sistema è in grado di rimanere in questo stato per un tempo indefinito.

Come il gestore PnP, il gestore dell'alimentazione fornisce notifiche al resto del sistema sui cambiamenti nello stato dell'alimentazione. Alcune applicazioni vogliono sapere quando il sistema è in procinto di essere chiuso in modo che possano iniziare a memorizzare su disco il proprio stato.

B.3.3.10 Registro di sistema

Windows mantiene molte delle sue informazioni di configurazione nei database interni, chiamati hive, che sono gestiti dal gestore della configurazione di Windows, comunemente noto come il registro di sistema (*registry*). Vi sono hive separati per le informazioni di sistema, le preferenze predefinite dell'utente, l'installazione del software, la sicurezza e le opzioni di avvio. Le informazioni nell'hive di sistema servono all'avvio: per questo motivo, il gestore del registry è implementato come componente dell'executive.

Il registry rappresenta lo stato di configurazione in ogni hive come uno spazio dei nomi gerarchico di chiavi (directory), ognuna delle quali può contenere un insieme di valori tipizzati, come una stringa unicode, una stringa ansi, un intero o dati binari non tipizzati. In teoria, le nuove chiavi e i valori vengono creati e inizializzati all'installazione di un nuovo software e vengono poi modificati per riflettere le modifiche nella configurazione di tale software. In pratica, il registry viene spesso usato come un database generico, come un meccanismo di comunicazione tra processi e per molti altri tali scopi.

Riavviare le applicazioni, o addirittura l'intero sistema, ogni volta che viene apportata una modifica nella configurazione sarebbe una seccatura. I programmi si basano allora su vari tipi di notifiche, come quelli forniti dal gestore PnP e dal gestore dell'alimentazione, per venire a conoscenza dei cambiamenti nella configurazione del sistema. Anche il registry fornisce notifiche che permettono ai thread di essere avvisati quando vengono apportate modifiche a una parte del registro di sistema. I thread possono così rilevare i cambiamenti di configurazione apportati al registry e adattarvisi.

Qualora vengano apportate modifiche significative al sistema, per esempio quando vengono installati aggiornamenti del sistema operativo o dei driver, vi è il pericolo che i dati di configurazione siano danneggiati (per esempio, se un driver funzionante viene sostituito da un driver non funzionante o se l'installazione di un'applicazione non riesce correttamente e lascia informazioni parziali nel registry). Prima di apportare questo genere di modifiche Windows crea un punto di ripristino. Il punto di ripristino contiene una copia degli hive prima del cambiamento e può essere utilizzato per ritornare alla vecchia versione degli hive permettendo a un sistema danneggiato di funzionare di nuovo.

Per migliorare la stabilità della configurazione del registry Windows ha aggiunto, a partire da Windows Vista, un meccanismo di transazione che può essere utilizzato per prevenire che il registry venga aggiornato parzialmente con un insieme di modifiche di configurazione correlate. Le transazioni del registry possono essere parte di transazioni più generali amministrate dal **gestore delle transazioni del kernel** (*kernel transaction manager*, ktm), che può anche includere transazioni del file system. Le transazioni ktm non hanno la semantica completa delle transazioni nei normali database e non hanno soppiantato il ripristino del sistema nel recupero da danni alla configurazione del registry causati dall'installazione del software.

B.3.3.11 Avvio

L'avvio di un pc Windows inizia quando si fornisce alimentazione all'hardware e il firmware contenuto nella rom è eseguito. In macchine più vecchie questo firmware era conosciuto come il bios, ma i sistemi più moderni utilizzano uefi (Unified Extensible Firmware Interface), più veloce e più generale e consente un migliore utilizzo delle caratteristiche dei processori moderni. Il firmware esegue la diagnostica post (*power-on self-test*), identifica molti dei dispositivi collegati al sistema e li inizializza a uno stato di accensione e poi costruisce la descrizione utilizzata dalla interfaccia avanzata di configurazione e alimentazione (acpi). Successivamente, il firmware trova il disco di sistema, carica il programma `bootmgr` di Windows e ne inizia l'esecuzione.

Su una macchina che è stata sospesa viene successivamente caricato il programma `winresume` che ripristina dal disco il sistema in esecuzione, in modo che il sistema possa continuare dal punto in cui era prima della sospensione. Su una macchina che è stata arrestata, il `bootmgr` esegue un'ulteriore inizializzazione del sistema e quindi carica `WinLoad`, che a sua volta carica `hal.dll`, il kernel (`ntoskrnl.exe`), tutti i driver necessari all'avvio, e l'hive di sistema. `WinLoad` trasferisce poi l'esecuzione al kernel.

Il kernel si inizializza e crea due processi: il processo di sistema contiene tutti i thread di lavoro interni al kernel e non esegue mai in modalità utente; smss (*session manager subsystem*, ossia "sottosistema per la gestione della sessione"), il primo processo utente creato, è simile al processo init (da *inizializzazione*) di unix. Il processo smss prosegue l'inizializzazione del sistema, tra l'altro instaurando i file di paginazione, caricando altri driver dei dispositivi e gestendo le sessioni di Windows. Ogni sessione è utilizzata per rappresentare un utente connesso a eccezione della sessione 0, utilizzata per eseguire alcuni servizi di sistema in background, come lsass e services. Una sessione viene ancorata da un'istanza del processo csrss. Ogni sessione diversa dalla sessione 0 esegue inizialmente il processo winlogon, che registra un utente e poi avvia il processo explorer, che implementa la gui di Windows. Il seguente elenco riporta alcuni degli aspetti della fase di avvio.

- smss completa l'inizializzazione del sistema e quindi avvia la sessione 0 e la prima sessione di login.
- wininit viene eseguito nella sessione 0 per inizializzare la modalità utente e avviare lsass, services e lsm, il gestore della sessione locale.
- lsass, il sottosistema di sicurezza, implementa servizi quali l'autenticazione degli utenti.
- services contiene scm, che supervisiona tutte le attività in background del sistema, compresi i servizi in modalità utente. Una serie di servizi saranno registrati per essere eseguiti all'avvio del sistema, mentre altri saranno avviati solo su richiesta o quando innescati da un evento come l'inserimento di un dispositivo.
- csrss è il processo del sottosistema dell'ambiente Win32. Viene avviato in ogni sessione, a differenza del sottosistema posix che viene avviato solo su richiesta quando viene creato un processo posix.
- winlogon viene eseguito in ogni sessione di Windows diversa dalla sessione 0 per registrare l'accesso di un utente.

Il sistema ottimizza le procedure d'avvio caricando preventivamente da disco alcuni file sulla base delle sessioni precedenti. La storia degli accessi al disco durante l'avvio serve anche per collocare i file di sistema su disco in modo da ridurre le operazioni di i/o richieste. Il numero di processi necessari all'avvio è ridotto raggruppando più servizi in un numero minore di processi. Tutti questi accorgimenti conferiscono un notevole contenimento dei tempi d'avvio del sistema, ma va anche sottolineato che i tempi d'avvio sono oggi meno importanti a causa delle funzionalità di sospensione e di sleep di Windows.

B.4 Terminal services e cambio rapido utente

Windows fornisce una console con interfaccia grafica che si interfaccia con l'utente tramite tastiera, mouse e monitor. La maggior parte dei sistemi supporta anche l'audio e il video. L'ingresso audio è utilizzato dal software di riconoscimento vocale di Windows, che rende il sistema più comodo e agevola la sua accessibilità per gli utenti disabili. Windows 7 ha aggiunto il supporto per l'hardware multi-touch, consentendo agli utenti di inserire dati toccando lo schermo e mediante i movimenti di uno o più dita. Infine, l'ingresso video attualmente utilizzato per applicazioni di comunicazione è suscettibile di utilizzo per l'interpretazione dei movimenti del viso, come Microsoft ha dimostrato con il suo prodotto Kinect per Xbox 360. Altre esperienze di input possono nascere in futuro dall'evoluzione del surface computer di Microsoft. Il surface computer, spesso installato in luoghi pubblici come alberghi e centri congressi, è una superficie piana dotata di speciali telecamere ed è in grado di monitorare le azioni di più utenti contemporaneamente e riconoscere gli oggetti che vengono posti sulla parte superiore.

Il pc è stato, ovviamente, pensato come un computer a uso personale, una macchina intrinsecamente a singolo utente. Le moderne versioni di Windows, tuttavia, supportano la condivisione di un pc tra più utenti. Ogni utente che è connesso mediante una gui possiede una sessione creata per rappresentare l'ambiente gui che verrà utilizzato e per contenere tutti i processi creati per eseguire le applicazioni. Windows consente la presenza di sessioni multiple allo stesso tempo su una singola macchina, ma supporta solo una console, costituita da tutti i monitor, le tastiere e i mouse collegati al pc. Una sola sessione alla volta può essere collegata alla console. Dalla schermata di accesso visualizzata sulla console gli utenti possono creare nuove sessioni o connettersi a una sessione esistente creata in precedenza. Ciò consente a più utenti di condividere un singolo pc senza doversi disconnettere e riconnettere a ogni cambio di utente. Microsoft chiama questo uso delle sessioni **cambio rapido utente** (*fast user switching*).

Gli utenti possono anche creare nuove sessioni o connettersi a sessioni esistenti su un pc da una sessione in esecuzione su un altro pc Windows. Il terminal server (ts) collega una delle finestre gui nella sessione locale di un utente alla sessione nuova o esistente, chiamata desktop remoto, sul computer remoto. L'uso più comune del desktop remoto è la connessione a una sessione sul proprio pc di lavoro dal proprio pc di casa.

Molte società utilizzano sistemi terminal server aziendali mantenuti in data center per eseguire tutte le sessioni utente che accedono alle risorse aziendali, piuttosto che permettere agli utenti di accedere a tali risorse dai pc presenti nell'ufficio di ciascun utente. Ogni computer server può gestire molte decine di sessioni di desktop remoto. In questi casi si parla di elaborazione thin-client, in cui i singoli computer si basano su un server per svolgere molte funzioni. Affidandosi ai terminal server dei data center si migliora l'affidabilità, la gestibilità e la sicurezza delle risorse informatiche aziendali.

Il ts è anche utilizzato da Windows per mettere in atto l'assistenza remota. Un utente remoto può essere invitato a condividere una sessione con l'utente connesso alla sessione sulla console. L'utente remoto può osservare le azioni dell'utente e ottenere il controllo del desktop per aiutarlo a risolvere eventuali problemi.

B.5 File system

Il file system nativo di Windows è ntfs, usato per tutti i volumi locali. Tuttavia, le memorie usb, le memorie flash delle fotocamere e i dischi esterni possono essere formattati con il file system fat a 32 bit per questioni di portabilità. fat è un formato di file system molto più vecchio ed è riconosciuto da molti sistemi diversi da Windows, come il software in esecuzione sulle macchine fotografiche. Uno degli svantaggi è che il file system fat non limita l'accesso ai file agli utenti autorizzati. L'unica soluzione per la protezione dei dati con fat consiste nell'eseguire un programma per crittografare i dati prima di memorizzarli nel file system.

Al contrario, ntfs utilizza acl per controllare l'accesso ai singoli file e supporta la crittografia implicita di singoli file o di interi volumi (utilizzando la funzionalità BitLocker di Windows). ntfs implementa molte altre caratteristiche, tra cui il recupero dei dati, la tolleranza ai guasti, i file e i file system di grandi dimensioni, i flussi multipli di dati, i nomi Unicode, i file sparsi, il journaling, le copie ombra dei volumi e la compressione dei file.

B.5.1 Struttura interna di ntfs

L'entità fondamentale dell'ntfs è il volume: si crea con l'utilità di amministrazione dei dischi del sistema operativo ed è basato su una partizione logica del disco. Un volume può occupare parte di un disco, un intero disco, o anche più dischi.

L'ntfs non tratta direttamente i singoli settori dei dischi, ma usa **cluster** (*unità di assegnazione*) costituiti da un numero di settori pari a una potenza di 2. La dimensione di un cluster si stabilisce nella fase di formattazione di un file system. La dimensione di default di un cluster si basa sulla dimensione del volume ed è pari a 4 kb per volumi più grandi di 2 gb. Vista la dimensione degli attuali dischi, ha senso utilizzare una dimensione di cluster maggiore di quella di default per raggiungere prestazioni migliori, anche se il vantaggio che si ottiene va a scapito di una maggior frammentazione interna.

Come indirizzi relativi al disco l'ntfs usa **numeri di cluster logici** (*logical cluster number*, lcn), che assegna numerando i cluster dall'inizio alla fine del disco. Il sistema può quindi calcolare un offset (espresso in byte) relativo al disco fisico moltiplicando l'lcn per la dimensione del cluster.

Un file dell'ntfs non è una semplice sequenza di byte come in unix; è invece un oggetto strutturato costituito da attributi. Ciascun attributo di un file è una sequenza indipendente di byte che può essere creata, eliminata, letta e scritta. Alcuni attributi sono comuni a tutti i file, per esempio il nome (o i nomi, se il file ha degli alias come ad esempio un nome ms-dos), l'ora di creazione, e il descrittore di sicurezza che stabilisce il controllo degli accessi. I dati dell'utente sono memorizzati in *attributi di dati*.

Molti dei tradizionali file di dati posseggono attributi di dati privi di nome contenenti tutti i dati del file. Si possono però creare altri flussi di dati con nomi espliciti. Per esempio, nei file Macintosh memorizzati sui server Windows, la *resource fork* è un flusso di dati con nome. Le interfacce IProp del modello com impiegano un flusso di dati con nome per memorizzare proprietà su file ordinari, comprese le anteprime delle immagini. In generale, si possono aggiungere attributi secondo necessità; vi si accede seguendo la sintassi *nameoffile:attributo*. ntfs restituisce solo la dimensione dell'attributo privo di nome in risposta a richieste di informazioni sui file, per esempio quando si esegue il comando `dir`.

Ogni file dell'ntfs è descritto da uno o più elementi contenuti in un array memorizzato in un file speciale detto **tabella principale dei file** (*master file table*, mft); la dimensione di uno di questi elementi è determinata al momento della creazione del file system, e varia da 1 kb a 4 kb. Gli attributi di piccole dimensioni sono memorizzati nella mft stessa, e sono detti attributi residenti; gli attributi più grandi, per esempio la massa anonima dei dati, sono invece attributi non residenti e sono memorizzati in una o più estensioni contigue (*extent*) nei dischi, e un puntatore a ogni estensione è memorizzato nell'mft. Se un file è molto piccolo anche l'attributo dei dati si può memorizzare nell'elemento dell'mft, mentre se un file possiede molti attributi o se è molto frammentato, e richiede quindi molti puntatori per individuare tutti i frammenti, un solo elemento nella mft potrebbe essere insufficiente. In questo caso il file è descritto da un elemento di base del file che punta a elementi aggiuntivi contenenti altri puntatori e attributi.

Ogni file in un volume ntfs possiede un unico identificatore detto riferimento al file (*file reference*); si tratta di 64 bit che consistono di un numero del file di 48 bit e di un numero di sequenza di 16 bit. Il numero del file è il numero dell'elemento (cioè l'indice dell'array) nella mft che descrive il file; il numero di sequenza viene incrementato ogni volta che si riutilizza un elemento della mft. Ciò permette all'ntfs di eseguire controlli interni di coerenza, per esempio per rilevare un riferimento a un file cancellato dopo che l'elemento della mft è stato riusato per un nuovo file.

B.5.1.1 Albero B+ di ntfs

Lo spazio dei nomi dell'ntfs, come in unix, è organizzato come una gerarchia di directory. Ogni directory usa una struttura dati detta albero B+ per memorizzare un indice dei nomi dei file contenuti nella directory; in un albero B+ la lunghezza di ogni percorso dalla radice a una foglia è la stessa e viene eliminato il costo della riorganizzazione dell'albero. La radice indice di una directory contiene il livello più alto di un albero B+; nel caso di una directory di grandi dimensioni, questo livello contiene i puntatori alle estensioni del disco nelle quali è memorizzato il resto dell'albero. Ogni elemento della directory contiene il nome del file e il suo riferimento, così come una copia dell'ora dell'ultimo aggiornamento e della dimensione del file presa dagli attributi residenti nella mft; il motivo per cui le copie di queste informazioni sono memorizzate nella directory è che in questo modo è più efficiente elencarne i contenuti: tutti i nomi, le dimensioni e gli orari sono disponibili all'interno della directory stessa, cosicché non c'è bisogno di recuperare questi attributi dagli elementi della mft per ognuno dei file.

B.5.1.2 Metadati di ntfs

I metadati di ogni volume dell'ntfs sono tutti memorizzati in alcuni file, il primo dei quali è proprio la mft. Il secondo, usato durante la procedura di recupero dei dati a seguito del danneggiamento della mft, contiene una copia dei primi 16 elementi della mft. Anche un certo numero di file successivi è dedicato a compiti specifici, come descritto di seguito.

- Il file di log registra tutte le modifiche ai metadati del sistema.

- Il file del volume contiene il nome del volume, la versione di ntfs con cui il volume è stato formattato, e un bit che indica se il volume richiede un controllo di coerenza, con l'utilizzo del programma `chkdsk`, a seguito di un possibile guasto.
- La tabella di definizione degli attributi indica i tipi di attributi usati nel volume, e le operazioni da eseguire per ognuno di loro.
- La directory radice è la directory di livello più alto nella gerarchia del file system.
- Il file bitmap tiene traccia dei cluster allocati ai file e dei cluster liberi.
- Il file d'avvio contiene il codice d'avvio di Windows e deve risiedere a uno specifico indirizzo del disco in modo che sia facilmente reperibile da un semplice caricatore d'avvio collocato su rom. Esso contiene inoltre l'indirizzo fisico della mft.
- Il file dei cluster guasti tiene traccia delle aree del volume non funzionanti; ntfs sfrutta queste informazioni per il ripristino a seguito di errori.

Il mantenimento di tutti i metadati ntfs in veri e propri file offre una proprietà utile. Come discusso nel Paragrafo B.3.3.6, il gestore della cache memorizza nella cache i dati contenuti nei file. Poiché tutti i metadati ntfs risiedono in file, questi dati possono essere memorizzati nella cache utilizzando gli stessi meccanismi che si impiegano per i dati comuni.

B.5.2 Ripristino

In molti semplici file system un calo di tensione al momento sbagliato può danneggiare le strutture dati del file system così gravemente da compromettere un intero volume. Molti file system di unix, incluso ufs, ma non zfs, memorizzano nei dischi metadati ridondanti e tentano il ripristino dei dati a seguito di un crash del sistema usando il programma `fsck` per controllare tutte le strutture dati del file system e ripristinarne forzatamente uno stato coerente; si tratta di una procedura che spesso comporta l'eliminazione dei file danneggiati e il rilascio di cluster di dati su cui erano stati scritti dati degli utenti, ma che non erano stati correttamente registrati nelle strutture di metadati del file system: è un processo che può essere lento e può portare a significative perdite di dati.

Per conferire robustezza al file system l'ntfs adotta un orientamento differente: tutti gli aggiornamenti delle strutture dati del file system sono eseguiti all'interno di transazioni. Prima che una struttura dati sia modificata, la transazione scrive nel log le informazioni necessarie per ripetere o annullare l'operazione; dopo che la struttura dati è stata cambiata, la transazione scrive un'annotazione di conferma nel log per indicare il successo della transazione.

A seguito di un crash, il sistema è in grado di riportare le strutture dati del file system a uno stato coerente elaborando le informazioni contenute nel log, prima ripetendo le operazioni confermate, poi annullando le operazioni che non erano state confermate prima del crash. Periodicamente (di solito ogni 5 secondi) si scrive un elemento di controllo nel log: il sistema non fa uso degli elementi che precedono l'elemento di controllo al fine di ripristinare i dati dopo un crash; tali elementi si possono quindi eliminare in modo che il file contenente il log non cresca eccessivamente. La prima volta che si accede a un volume dopo l'avviamento del sistema, l'ntfs esegue automaticamente la procedura di ripristino.

Questa strategia non garantisce l'integrità di tutti i contenuti dei file degli utenti dopo un crash; assicura solo che le strutture dati del file system (cioè i file di metadati) siano integre e riflettano uno stato coerente precedente al crash. Sarebbe possibile estendere il metodo delle transazioni ai file degli utenti e Microsoft ha mosso i primi passi in questa direzione con Windows Vista.

Il log è memorizzato nel terzo file di metadati all'inizio del volume; viene creato nella fase di formattazione del file system e ha una dimensione massima fissata. È costituito da due parti: l'area per il log, che è una coda circolare di elementi, e l'area di riavvio, contenente informazioni contestuali quali il punto dell'area per il log dal quale l'ntfs dovrebbe cominciare la lettura durante una procedura di ripristino. In effetti l'area di riavvio contiene due copie di queste informazioni, in modo che il ripristino sia possibile se una copia è stata danneggiata durante il crash.

La funzione di logging è fornita dal servizio di gestione del file di log; oltre a scrivere gli elementi nel file di log e a eseguire operazioni di ripristino, questo servizio tiene traccia dello spazio libero nel file di log: quando esso si riduce eccessivamente il servizio di gestione del file di log accoda le transazioni in evase, e l'ntfs sospende tutte le nuove operazioni di i/o. Una volta che tutte le operazioni in corso sono state completate, l'ntfs attiva il gestore della cache per trasferire tutti i dati nei dischi, reimposta il file di log e, infine, esegue le transazioni in coda.

B.5.3 Sicurezza

La sicurezza di un volume dell'ntfs deriva dal modello a oggetti del sistema Windows. Ogni file ntfs fa riferimento a un descrittore di sicurezza, che specifica il proprietario del file, e a una lista di controllo degli accessi, che contiene le autorizzazioni di accesso concesse o negate a ciascun utente o gruppo elencato. Le prime versioni di ntfs utilizzavano un attributo descrittore di sicurezza separato per ciascun file. A partire da Windows 2000, l'attributo descrittore di sicurezza punta a una copia condivisa, con un notevole risparmio di spazio su disco e sulla cache, poiché numerosissimi file hanno descrittori di sicurezza identici.

Nel suo normale funzionamento, ntfs non fa rispettare i permessi nell'attraversamento delle directory dei nomi di percorso del file. Tuttavia, per garantire compatibilità con posix, questi controlli possono essere abilitati. I controlli sull'attraversamento sono intrinsecamente più onerosi, visto che la moderna analisi dei nomi di percorso dei file è basata sul matching dei prefissi anziché sull'analisi, directory per directory, dei nomi di percorso. Il matching dei prefissi è un algoritmo che cerca le stringhe in una cache e trova la voce con la corrispondenza più lunga. Per esempio, la voce `\foo\bar\dir` è una corrispondenza per `\foo\bar\dir2\dir3\myfile`. La cache di matching dei prefissi permette di iniziare l'attraversamento della struttura da una posizione più profonda, risparmiando diversi passaggi. Il rispetto dei controlli di attraversamento richiede che l'accesso dell'utente sia controllato a ogni livello di directory. Per esempio, un utente potrebbe non avere il permesso di attraversare `\foo\bar`, quindi iniziare dall'accesso a `\foo\bar\dir` sarebbe un errore.

B.5.4 Gestione dei volumi e tolleranza ai guasti

Il driver di disco fault tolerant del sistema Windows si chiama `FtDisk`: una volta installato mette a disposizione molti modi di combinare diverse unità a disco in un unico volume logico, in modo da migliorare prestazioni, capacità di memorizzazione e affidabilità.

B.5.4.1 Insiemi di volumi e insiemi RAID

Uno dei possibili modi di combinare dischi diversi è di concatenarli logicamente in modo da formare un solo volume logico, com'è illustrato dalla Figura B.7. In Windows questo volume logico si chiama **insieme di volumi** (*volume set*) e consiste al massimo di 32 partizioni fisiche. Un insieme di volumi che contenga un volume dell'ntfs può essere esteso senza modificare i dati già memorizzati all'interno del file system: i metadati del file della bitmap del volume dell'ntfs sono semplicemente modificati al fine di includere lo spazio che si desidera aggiungere. L'ntfs continua a usare lo stesso meccanismo lcn che usa per un unico disco fisico, e il driver `FtDisk` fornisce le conversioni degli offset riferiti a un volume logico in offset riferiti a uno specifico disco.

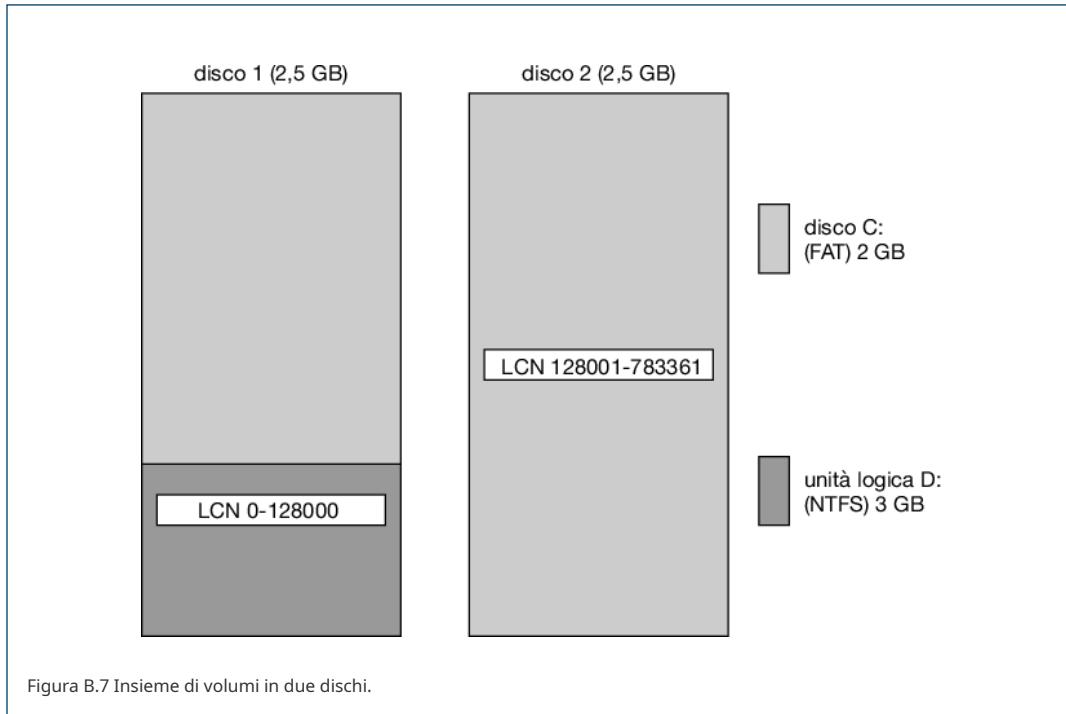


Figura B.7 Insieme di volumi in due dischi.

Un altro modo di combinare più partizioni fisiche è di intercalare i loro blocchi per formare un cosiddetto insieme di sezioni (*stripe set*). Si tratta di un metodo noto anche come raid di livello 0 o sezionamento del disco (per maggiori informazioni sul raid si veda il Paragrafo 11.8 del Capitolo 11). Il driver `FtDisk` usa sezioni di 64 kb: i primi 64 kb del volume logico sono memorizzati nella prima partizione fisica, i secondi 64 kb nella seconda partizione, e così via finché ogni partizione non abbia contribuito con 64 kb di spazio; a questo punto il processo di allocazione riprende dal primo disco, che contribuisce con un secondo blocco di 64 kb. Un insieme di sezioni costituisce un solo ampio volume logico, ma l'effettiva configurazione fisica può migliorare l'ampiezza di banda dell'i/o, perché per trasferimenti ingenti tutti i dischi possono trasferire dati in parallelo. Windows supporta anche il raid di livello 5, gli insiemi di sezioni con parità e il raid di livello 1 (copia speculare o mirroring).

B.5.4.2 Accantonamento dei settori e rimappatura dei cluster

Per gestire il malfunzionamento dei settori `FtDisk` si serve di una funzione hardware detta accantonamento di settori, e l'ntfs utilizza una tecnica software detta *rimappatura dei cluster*. L'accantonamento di settori è una funzionalità hardware fornita da molte unità disco: quando si formatta un disco, l'unità crea una corrispondenza fra i numeri dei blocchi logici e i settori fisici funzionanti del disco; essa accantona alcuni settori lasciandoli volutamente inutilizzati, in modo che se un settore diviene difettoso, `FtDisk` può richiedere all'unità di sostituirlo con uno di loro. La rimappatura dei cluster è una procedura messa in atto dal file system: se un blocco del disco diventa difettoso, l'ntfs lo sostituisce con un blocco libero differente cambiando i puntatori interessati nell'mft; l'ntfs inoltre annota che il blocco malfunzionante non dovrà più essere allocato.

L'usuale conseguenza del malfunzionamento di un blocco è la perdita di dati; la rimappatura dei cluster e l'accantonamento di settori, però, si possono combinare con i volumi tolleranti i guasti per mascherare il guasto di un blocco. Se una lettura non riesce, il sistema ricostruisce i dati mancanti leggendo l'immagine speculare o calcolando la parità nel caso di un insieme di sezioni con parità; i dati così ricostruiti si memorizzano in una nuova locazione del disco ottenuta grazie alla rimappatura dei cluster o all'accantonamento di settori.

B.5.5 Compressione

L'ntfs è in grado di eseguire la compressione dei dati di un singolo file o di tutti i file di dati di un'intera directory. Per comprimere un file, divide i dati in unità di compressione, cioè blocchi di 16 cluster contigui. Al momento della scrittura di ciascuna unità di

compressione si applica un algoritmo di compressione dei dati; se il risultato occupa meno di 16 cluster, si scrive nei dischi la versione compressa. Durante la lettura l'ntfs è in grado di determinare se i dati sono stati compressi; in questo caso la lunghezza dell'unità di compressione memorizzata è inferiore ai 16 cluster; per migliorare le prestazioni durante la lettura di unità di compressione contigue, l'ntfs legge e decomprime in anticipo rispetto alle richieste dell'applicazione.

Nel caso di file sparsi o file contenenti prevalentemente zeri, l'ntfs adotta un'altra tecnica per risparmiare spazio: i cluster che contengono solo zeri non sono realmente assegnati o memorizzati nei dischi, ma si lasciano dei **buchi** (*gap*) nella sequenza dei numeri virtuali dei cluster memorizzati nell'elemento dell'mft relativo al file. Se nella lettura di un file ritrova un buco nella sequenza in questione, l'ntfs riempie semplicemente di zeri la parte interessata del buffer per l'i/o del chiamante. Questa tecnica è adottata anche nel sistema operativo unix.

B.5.6 Punti di montaggio, collegamenti simbolici e collegamenti fisici

I punti di montaggio sono una forma di collegamento simbolico peculiare delle directory di ntfs, introdotti con Windows 2000. Essi offrono agli amministratori la possibilità di organizzare i volumi del disco in maniera più flessibile, rispetto all'utilizzo di nomi globali (quali le lettere delle unità). I punti di montaggio sono realizzati come collegamento simbolico a dati associati contenenti il vero nome del volume. È presumibile che i punti di montaggio soppiantino completamente le lettere delle unità, ma solo in seguito a una lunga transizione dovuta alla dipendenza di molte applicazioni dalle lettere delle unità.

Windows Vista ha introdotto il supporto a una forma più generale di collegamenti simbolici, simili a quelli che si trovano in unix. I collegamenti possono essere assoluti o relativi, possono puntare a oggetti che non esistono e possono puntare a file e directory, anche attraversando i volumi. ntfs supporta anche i collegamenti fisici (hard link), in cui un singolo file ha una voce in più di una directory sullo stesso volume.

B.5.7 Giornale delle modifiche

L'ntfs mantiene un giornale in cui descrive tutti i cambiamenti avvenuti nel file system. I servizi in modalità utente possono ricevere notifica delle modifiche intervenute e individuare in un secondo momento i file modificati, leggendo dal giornale. Il servizio di indicizzazione del contenuto usa il giornale delle modifiche per individuare i file che devono essere nuovamente indicizzati, mentre il servizio di replicazione dei file lo utilizza per identificare i file che devono essere replicati in rete.

B.5.8 Copie ombra dei volumi

Windows ha la capacità di portare un volume in uno stato conosciuto per poi creare una copia ombra (*shadow*), utilizzabile per creare copie di backup coerenti del volume. La stessa tecnica viene chiamata istantanea (*snapshot*) in altri file system. Si tratta di una variante della copiatura su scrittura: i blocchi modificati dopo la creazione di una copia ombra sono mantenuti nella forma originale su quest'ultima. Affinché la copia raggiunga uno stato coerente con il volume è richiesta la cooperazione delle applicazioni, dal momento che il sistema non può sapere quando i dati usati dall'applicazione siano in uno stato stabile a partire dal quale l'applicazione stessa può essere riavviata in modo sicuro.

La versione server di Windows usa copie ombra per rendere prontamente disponibili le vecchie versioni dei file memorizzate sui file server. Gli utenti possono così usufruire di documenti conformi alla versione originale memorizzata sui file server. Gli utenti possono avvalersi di questa caratteristica per recuperare file cancellati per errore, o semplicemente per consultare una versione precedente dei file, il tutto senza la necessità di un nastro contenente copie di riserva.

B.6 Servizi di rete

I servizi di rete di Windows operano sia secondo il modello client-server sia secondo il modello peer-to-peer; il sistema operativo fornisce anche strumenti per la gestione della rete. I componenti del sottosistema di networking permettono la trasmissione dei dati, la comunicazione fra processi, la condivisione dei file attraverso la rete e la possibilità di stampare impiegando stampanti remote.

B.6.1 Interfacce di rete

Nella descrizione dei servizi di rete di Windows si fa riferimento a due interfacce di rete interne, dette ndis (*network device interface specification*) e tdi (*transport driver interface*). L'interfaccia ndis fu sviluppata nel 1989 da Microsoft e da 3Com al fine di separare gli adattatori di rete dai protocolli di trasporto, in modo che gli uni potessero essere modificati senza che ciò avesse effetto sugli altri. L'ndis costituisce l'interfaccia fra lo strato di collegamento dei dati e lo strato di rete nel modello osi e permette a molti protocolli di funzionare alla presenza di diversi adattatori di rete. Analogamente, la tdi è l'interfaccia fra lo strato di trasporto (strato 4) e lo strato di sessione (strato 5) del modello osi: essa permette a ogni componente dello strato di sessione di adoperare ogni meccanismo di trasporto disponibile. (Necessità simili hanno portato al cosiddetto meccanismo *stream* di unix.) L'interfaccia tdi può gestire sia il trasporto privo di connessione sia quello basato sulla connessione, ed è dotata di funzioni per la trasmissione di tutti i tipi di dati.

B.6.2 Protocolli

Nel sistema Windows i protocolli di trasporto sono realizzati come driver che si possono caricare e scaricare dinamicamente, anche se in pratica il sistema deve di solito essere riavviato a seguito di una di queste modifiche. Il sistema Windows dispone di diversi protocolli di networking. Di seguito se ne illustrano alcuni.

B.6.2.1 Protocollo smb

Il protocollo smb (*server message-block*) fu introdotto per la prima volta con l'ms-dos 3.1 e si usa per trasmettere richieste di i/o sulla rete. Esso tratta quattro tipi di messaggi. I messaggi *Session control* sono comandi per l'instaurazione e la chiusura della connessione di un oggetto di ridirezione (*redirector*) con una risorsa condivisa del server. I messaggi *File* vengono usati da un redirector per accedere a file del server. I messaggi *Printer* sono usati per trasmettere dati a una coda di stampa remota e per ricevere informazioni sullo stato della stampa, mentre i messaggi *Message* si usano per comunicare con un altro sistema in rete. Una versione del protocollo smb è stato pubblicato come Common Internet File System (cifs) ed è disponibile su una serie di sistemi operativi.

B.6.2.2 Protocollo tcp/ip

La pila di protocolli tcp/ip che si usa nella rete Internet è diventata lo standard *de facto* per la comunicazione in rete. Il sistema operativo Windows usa i protocolli tcp/ip per mettere in comunicazione un'ampia gamma di sistemi operativi e di piattaforme. Il pacchetto tcp/ip di Windows comprende un protocollo per la gestione di rete snmp (*simple network-management protocol*), il protocollo dhcp (*dynamic host-configuration protocol*) per la configurazione dinamica dei calcolatori che si connettono alla rete e il vecchio servizio wins (*Windows Internet name service*) di risoluzione dei nomi. Windows Vista ha introdotto una nuova implementazione di tcp/ip che supporta sia IPv4 sia IPv6 nello stesso stack di rete. Questa nuova implementazione supporta anche l'offloading dello stack di rete su hardware avanzato, per ottenere prestazioni molto elevate sui server.

Windows fornisce un firewall software che limita le porte tcp utilizzabili dai programmi per la comunicazione di rete. I firewall di rete sono comunemente implementati nei router e costituiscono una misura di sicurezza molto importante. Avere un firewall integrato nel sistema operativo rende un router hardware inutile e fornisce una gestione più integrata e una maggior facilità di utilizzo.

B.6.2.3 Protocollo pptp

Il protocollo pptp (*point-to-point tunneling protocol*) è un protocollo messo a disposizione dal sistema Windows per la comunicazione fra moduli server di accesso remoto residenti in calcolatori server Windows e altri sistemi client connessi alla rete Internet; i server possono cifrare i dati trasmessi, e gestiscono reti private virtuali (*virtual private network*, vpn) multiprotocollo nella rete Internet.

B.6.2.4 Protocollo HTTP

Il protocollo http è utilizzato per avere (*get*) o fornire (*put*) informazioni utilizzando il World Wide Web. Windows implementa http utilizzando un driver in modalità kernel, quindi i server web sono in grado di funzionare con un basso overhead di connessione allo stack di rete. http è un protocollo abbastanza generale che Windows rende disponibile come opzione di trasporto per l'implementazione di rpc.

B.6.2.5 Protocollo per la realizzazione e lo sviluppo distribuiti di contenuti

Webdav (*web distributed authoring and versioning*) è un protocollo, basato su http, per la realizzazione e lo sviluppo coordinato di contenuti in rete. Windows installa all'interno del file system un oggetto di ridirezione Webdav; ciò permette al protocollo in questione di cooperare con altre funzionalità del file system, quali la crittografia. I file personali possono in tal modo essere memorizzati con sicurezza in un luogo pubblico. Visto che Webdav utilizza http, che è un protocollo *get/put*, Windows deve memorizzare localmente sulla cache i file in modo che i programmi possano usare operazioni di *read* e *write* su parti di essi.

B.6.2.6 Pipe con nome

Le pipe con nome sono un meccanismo di trasmissione dei messaggi. Un processo può utilizzare le pipe con nome per comunicare con altri processi sulla stessa macchina. Dal momento che le pipe con nome sono accessibili tramite l'interfaccia del file system, i

meccanismi di sicurezza utilizzati per oggetti file si applicano anche alle pipe con nome. Il protocollo smb supporta le pipe con nome, che possono dunque essere utilizzate anche per la comunicazione tra processi su sistemi diversi.

Il nome di una pipe con nome segue una convenzione detta unc (*uniform naming convention*). Un nome unc si presenta in modo simile a un tipico nome di file remoto.

Il suo formato è `\\\nome_del_server\nome_della_condivisione\x\y\z`, dove `nome_del_server` identifica un server della rete; `nome_della_condivisione` specifica qualsiasi risorsa resa disponibili agli utenti della rete, per esempio directory, file, pipe con nome e stampanti, e la parte terminale `\x\y\z` è un ordinario nome di percorso di un file.

B.6.2.7 *rpc*

Una rpc (*remote procedure call*) è un meccanismo di comunicazione client-server che permette a un'applicazione residente in una certa macchina di eseguire una chiamata di procedura relativa a codice residente in un'altra macchina. Quando il client invoca una procedura remota, il sistema delle rpc richiama una procedura locale detta stub che impacca i suoi argomenti in un messaggio e li invia tramite la rete a un determinato processo server, dopo di che si blocca. Nel frattempo, il server estrae gli argomenti dal messaggio, richiama la procedura, impacca i risultati in un messaggio, e li spedisce allo stub del client; quest'ultimo riprende l'elaborazione, riceve il messaggio, estrae i risultati della rpc e li restituisce al chiamante. Impaccamento ed estrazione degli argomenti sono a volte chiamati marshaling. Il codice dello stub e i descrittori necessari per impacchettare e spacchettare gli argomenti per una rpc sono compilati da una specifica scritta in midi (Microsoft Interface Definition Language).

La funzione rpc di Windows aderisce al diffuso standard rpc Distributed Computing Environment, cosicché i programmi che usano la rpc di questo sistema sono facilmente adattabili ad altri sistemi. Lo standard rpc è dettagliato: nasconde molte differenze esistenti fra le architetture dei calcolatori, per esempio la dimensione dei numeri binari e l'ordine dei bit e dei byte nelle parole, specificando formati convenzionali dei dati per i messaggi rpc.

B.6.2.8 *Modello com*

Il modello com (*component object model*) è un meccanismo per la comunicazione fra processi originariamente sviluppato per Windows. Gli oggetti com mettono a disposizione una precisa interfaccia per la manipolazione dei loro dati. A titolo d'esempio, com funge da infrastruttura per la realizzazione della tecnologia Microsoft ole (*object linking and embedding*, ossia "collegamento e integrazione degli oggetti"), grazie alla quale è possibile inserire fogli di calcolo nei documenti Word. Diversi servizi Windows forniscono interfacce com. Windows è dotato di un'estensione distribuita del modello com nota come dcom, che permette lo sviluppo trasparente di applicazioni distribuite su una rete grazie all'uso di rpc.

B.6.3 Redirector e server

Nel sistema operativo Windows un'applicazione può usare l'api di i/o per accedere ai file di un altro calcolatore come se essi fossero locali, ammesso che sul computer remoto giri un server cifs come quelli forniti da Windows. Un oggetto di ridirezione (*redirector*) è un oggetto lato client che trasmette richieste di i/o per file remoti, poi soddisfatte da un server. Per motivi legati alle prestazioni e alla sicurezza, il redirector e i server sono eseguiti in modalità kernel.

Più in dettaglio, l'accesso a un file remoto avviene come segue:

1. l'applicazione richiama il gestore dell'i/o per richiedere l'apertura di un file, fornendo un nome nel formato standard unc;
2. il gestore dell'i/o costruisce un pacchetto di richiesta di i/o com'è descritto nel Paragrafo B.3.3.5;
3. il gestore dell'i/o rileva che la richiesta si riferisce a un file remoto, e chiama un driver detto mup (*multiple universal-naming-convention provider*);
4. il mup invia in modo asincrono l'irp a tutti i redirector registrati;
5. un redirector capace di soddisfare la richiesta risponde al mup; per evitare di porre in futuro la stessa domanda a tutti i redirector, il mup usa una cache per annotare l'identità del redirector capace di trattare questo file;
6. il redirector trasmette la richiesta sulla rete al sistema remoto;
7. i driver di rete del sistema remoto ricevono la richiesta e la passano al driver server;
8. il driver server passa la richiesta al driver appropriato del file system locale;
9. l'appropriato driver del dispositivo è chiamato per accedere ai dati;
10. i risultati sono restituiti al driver server, che li rispedisce al redirector richiedente, il quale li restituisce all'applicazione chiamante tramite il gestore dell'i/o.

Un processo simile avviene nel caso delle applicazioni che usano l'api di rete Win32 anziché i servizi unc, in questo caso invece di un mup si usa un modulo detto instradatore multiplo.

Per motivi legati alla portabilità il redirector e i server usano le api tdi per il trasporto di rete; le richieste stesse sono espresse per mezzo di un protocollo di più alto livello, che di solito è il protocollo smb menzionato nel Paragrafo B.6.2. L'elenco dei redirector è contenuto nell'hive di sistema del registro.

B.6.3.1 *File system distribuito*

I nomi unc, che fanno esplicito riferimento al nome del server, non si dimostrano sempre appropriati, poiché vi può essere disponibilità di numerosi file server per gli stessi contenuti, e i nomi unc includono esplicitamente il nome del server. Windows fornisce un **protocollo di file system distribuito** (*distributed file system*, dfs), grazie al quale un amministratore di rete può smistare i file da più server utilizzando un solo spazio dei nomi distribuito.

B.6.3.2 *Reindirizzamento delle cartelle e caching lato client*

Al fine di migliorare l'esperienza degli utenti che operano spesso su pc diversi per motivi professionali, Windows permette agli amministratori di assegnare agli utenti un profilo mobile, che mantiene su server le preferenze dell'utente e altre impostazioni. Per memorizzare automaticamente i documenti e gli altri file dell'utente su un server, si applica quindi il reindirizzamento delle cartelle.

Questa tecnica funziona a dovere finché uno dei calcolatori non è più collegato alla rete, come nel caso di un portatile su un aereo. Per fornire agli utenti un accesso non in linea ai loro file reindirizzati, Windows utilizza il **caching lato client** (*client-side caching, csc*). Questo meccanismo è anche impiegato quando il computer è in linea per mantenere copie dei file del server sulla macchina locale, al fine di ottenere prestazioni migliori. I file sono inviati al server non appena cambiano e, se il calcolatore si disconnette, i file sono ancora disponibili; l'aggiornamento del server è rinviato alla volta successiva in cui il calcolatore tornerà in linea.

B.6.4 Domini

Per molti ambienti di rete esistono gruppi naturali di utenti, per esempio gli studenti di un laboratorio scolastico o gli impiegati di un'azienda. Molto spesso si vuole che tutti i membri di un gruppo possano accedere a risorse condivise messe a disposizione dai calcolatori del gruppo. Per gestire i diritti d'accesso globale all'interno di un tale gruppo, il sistema Windows fa uso del concetto di dominio. In precedenza, questi domini non avevano alcuna relazione con il dns (*domain name system*) che trasformava nomi di calcolatori collegati alla rete Internet in indirizzi ip; ora invece sono strettamente correlati.

In particolare, un dominio Windows è un gruppo di stazioni di lavoro e server che condivide le politiche di sicurezza e ha una comune database degli utenti. Poiché il sistema Windows impiega per l'autenticazione e la fidatezza il protocollo Kerberos, un dominio Windows è quello che nel Kerberos si chiama *realm* (realm). Nel sistema Windows si segue un approccio gerarchico per stabilire relazioni di fiducia tra i domini collegati. Le relazioni di fiducia si basano sul dns e sono relazioni transitive che si possono trasmettere in entrambe le direzioni nella gerarchia. Questo metodo riduce il numero di relazioni di fiducia richieste da $n * (n - 1)$ a $O(n)$. Le stazioni di lavoro nel dominio si fidano del controllore di dominio affinché dia informazioni corrette sui diritti d'accesso di ciascun utente (caricati nel token di accesso dell'utente da lsass). Ogni utente si riserva comunque la possibilità di limitare l'accesso alle proprie stazioni di lavoro, indipendentemente dalle disposizioni del controllore il dominio.

B.6.5 Active Directory

Active Directory è lo strumento con cui Windows implementa i servizi del protocollo ldap (*lightweight directory-access protocol*). Active Directory memorizza le informazioni di topologia relative ai domini, gestisce i profili individuali e di gruppo degli utenti di un dominio, con le relative parole d'ordine e offre una memorizzazione nel dominio per funzionalità Windows che ne hanno bisogno, come i **criteri di gruppo** (*Windows group policy*).

Gli amministratori impiegano i criteri di gruppo per stabilire standard uniformi per le preferenze e il software dei desktop. Per molte aziende commerciali che si occupano di tecnologia dell'informazione, l'uniformità consente di ridurre drasticamente il costo dell'elaborazione.

B.7 Interfaccia di programmazione

L'api Win32 è l'interfaccia fondamentale ai servizi del sistema operativo Windows. Questo paragrafo descrive cinque aspetti chiave dell'api Win32: l'accesso agli oggetti del kernel, la condivisione degli oggetti fra i processi, la gestione dei processi, la comunicazione fra i processi e la gestione della memoria.

B.7.1 Accesso agli oggetti del kernel

Il kernel del sistema operativo Windows mette a disposizione dei programmi applicativi molti servizi: i programmi usufruiscono di questi servizi manipolando oggetti del kernel. Un processo accede a un oggetto del kernel di nome `xxx` chiamando la funzione `CreateXXX()` per ottenere un handle di un'istanza di `xxx`; questo handle è unico per il processo. Se la funzione `Create` non va a buon fine riporta il valore `0` o una costante specifica detta `INVALID_HANDLE_VALUE`, secondo l'oggetto che si sta apendo. Un processo può chiudere un handle richiamando la funzione `CloseHandle()`, e il sistema può eliminare l'oggetto se il contatore che totalizza il numero degli handle che fanno riferimento all'oggetto in tutti i processi arriva a zero.

B.7.2 Condivisione degli oggetti tra processi

Il sistema Windows fornisce ai processi tre modi di condividere un oggetto. Il primo consiste nel fatto che un processo figlio può ereditare un handle dell'oggetto: quando invoca la funzione `CreateXXX()`, il genitore fornisce una struttura `SECURITIES_ATTRIBUTES` il cui campo `bInheritHandle` ha valore `TRUE`; questo campo crea un handle ereditabile. A questo punto si può creare il processo figlio, passando il valore `TRUE` all'argomento `bInheritHandle` della funzione `CreateProcess()`. Nella Figura B.8 è mostrato un esempio di codice che crea l'handle di un semaforo ereditato dal processo figlio.

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
HANDLE a_semaphore = CreateSemaphore(&sa,1,1,NULL);
char command_line[132];
ostrstream ostring(command_line,sizeof(command_line));
ostring << a_semaphore << ends;
CreateProcess("another_process.exe",command_line,
NULL,NULL,TRUE, ... );
```

Figura B.8 Codice che permette a un figlio di condividere un oggetto ereditandone l'handle.

Nell'ipotesi che il processo figlio sappia quali handle sono condivisi, la condivisione degli oggetti permette di realizzare la comunicazione fra genitore e figlio; nell'esempio della Figura B.8 il processo figlio otterrebbe il valore dell'handle dal primo argomento della riga di comando e potrebbe quindi condividere il semaforo con il processo genitore.

Il secondo metodo di condivisione degli oggetti presuppone che l'oggetto abbia ricevuto un nome dal processo che lo ha creato, cosicché un altro processo possa richiedere l'apertura dell'oggetto riferendosi al suo nome. Questa tecnica ha due svantaggi: il primo è che il sistema operativo Windows non fornisce un modo di controllare se un oggetto con un certo nome già esiste; il secondo è che lo spazio dei nomi degli oggetti è globale, senza alcuna distinzione relativa ai tipi di oggetti. Due applicazioni potrebbero per esempio creare un oggetto denominato `foo`, mentre ciò che si voleva erano due oggetti distinti e forse anche di diverso tipo.

Il vantaggio degli oggetti con nome è che processi non correlati possono facilmente condividerli: un primo processo richiamerebbe una delle funzioni `CreateXXX()`, e fornirebbe un nome come parametro; un secondo processo potrebbe ottenere un handle di questo oggetto chiamando `OpenXXX()` (o `CreateXXX()`) con lo stesso nome, com'è mostrato dall'esempio nella Figura B.9.

```

// processo A
...
HANDLE a_semaphore = CreateSemaphore(NULL,1,1,"MySEM1");
...
// processo B
...
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
                                    FALSE, "MySEM1");
...

```

Figura B.9 Codice per la condivisione di un oggetto per ricerca del nome.

Il terzo modo di condividere oggetti si basa sulla funzione `DuplicateHandle()`; esso richiede altri metodi di comunicazione fra processi per trasmettere l'handle duplicato. Se un processo possiede un handle con un certo valore, un altro processo può ottenerne uno dello stesso oggetto, e pertanto condividerlo; un esempio di questo metodo è mostrato nella Figura B.10.

```

// il processo A vuole fornire al processo B
// l'accesso a un semaforo
// processo A
HANDLE a_semaphore = CreateSemaphore(NULL,1,1,NULL);
// invia il valore del semaforo al processo B
// usando un messaggio o la condivisione della memoria
...
// processo B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS,FALSE,
                                 process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a,a_semaphore,
                GetCurrentProcess(),&b_semaphore,0,FALSE,
                DUPLICATE_SAME_ACCESS);
// usa b_semaphore per accedere al semaforo

```

Figura B.10 Codice per la condivisione di un oggetto tramite il passaggio di un handle.

B.7.3 Gestione dei processi

Nel sistema Windows un processo è un'istanza caricata di un'applicazione e un thread è un'unità eseguibile di codice che può essere sottoposta a scheduling dal dispatcher del kernel: un processo contiene uno o più thread. Un processo viene creato quando un thread in qualche altro processo invoca l'api `CreateProcess()`, che carica le librerie dinamiche usate dal processo e crea un thread iniziale nel processo; tramite la funzione `CreateThread()` si possono creare thread aggiuntivi: ogni thread possiede il suo stack, la cui dimensione predefinita è di 1 mb se non è altrimenti specificato da un argomento di `CreateThread()`.

B.7.3.1 Regola di scheduling

Le priorità nell'ambiente Win32 sono basate sul modello di scheduling del kernel nativo (nt), ma non tutti i valori di priorità possono essere scelti. L'ambiente Win32 usa quattro classi di priorità:

1. IDLE_PRIORITY_CLASS (livello di priorità 4)
2. NORMAL_PRIORITY_CLASS (livello di priorità 8)
3. HIGH_PRIORITY_CLASS (livello di priorità 13)
4. REALTIME_PRIORITY_CLASS (livello 24).

Ordinariamente i processi appartengono alla classe `NORMAL_PRIORITY_CLASS`, sempre che il genitore del processo non sia di classe `IDLE_PRIORITY_CLASS`, oppure non sia stata specificata un'altra classe al momento della chiamata alla funzione `CreateProcess()`. La

classe di priorità di un processo diventa la classe predefinita per tutti i thread eseguiti nel processo. La classe di priorità si può modificare tramite la funzione `SetPriorityClass()`, o passando un argomento al comando `START`. Si noti che solo gli utenti che godono del privilegio che consente d'incrementare le priorità di scheduling possono assegnare un processo alla classe `REALTIME_PRIORITY_CLASS`; gli amministratori e gli appartenenti al gruppo *Power users* godono automaticamente di questo privilegio.

Quando un utente esegue un programma interattivo il sistema deve schedulare i thread del processo per fornire buone prestazioni: è per questo motivo che Windows ha una regola di scheduling speciale per i processi di classe `NORMAL_PRIORITY_CLASS`. Windows distingue tra il processo associato alla finestra in primo piano e gli altri processi, in background. Quando un processo si sposta in primo piano il sistema operativo incrementa il quanto di tempo di tutti i suoi thread di un fattore 3. Questo aumento ha l'effetto di concedere ai thread con prevalenza d'elaborazione (cpu-bound) in primo piano un tempo d'esecuzione tre volte più lungo rispetto ai thread simili dei processi in background.

B.7.3.2 Priorità dei thread

La priorità iniziale di un thread è determinata dalla sua classe, ma si può modificare tramite la funzione `SetThreadPriority()`; essa accetta un argomento che specifica una priorità relativa alla priorità di base della classe del thread:

- `THREAD_PRIORITY_LOWEST`: base - 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base - 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

Per regolare la priorità si usano altri due valori convenzionali. Il kernel ha due classi di priorità (Paragrafo B.3.2.2): la classe per le elaborazioni real time (livelli 16-31) e la classe a priorità variabile (livelli 1-15); `THREAD_PRIORITY_IDLE` rappresenta il livello di priorità 16 per i thread d'elaborazione real time, e il livello di priorità 1 per i thread a priorità variabile.

`THREAD_PRIORITY_TIME_CRITICAL` rappresenta il livello di priorità 31 per i thread d'elaborazione real time, e il livello 15 per i thread a priorità variabile.

Come si descrive nel Paragrafo B.3.2.2, il kernel regola dinamicamente la priorità di un thread in funzione del fatto che si tratti di un thread con prevalenza di i/o o con prevalenza d'elaborazione; l'api Win32 fornisce un modo di disattivare questo processo di regolazione tramite le funzioni `SetProcessPriorityBoost()` e `SetThreadPriorityBoost()`.

B.7.3.3 Sospensione e ripristino dei thread

Un thread può essere creato in uno stato sospeso o può essere collocato in uno stato sospeso in seguito utilizzando la funzione `SuspendThread()`. Prima che un thread sospeso possa essere schedulato dal dispatcher del kernel deve essere ripristinato dallo stato sospeso con l'uso della funzione `ResumeThread()`. Entrambe le funzioni impostano un contatore in modo che se un thread viene sospeso per due volte, deve essere ripristinato per due volte prima che possa essere eseguito.

B.7.3.4 Sincronizzazione dei thread

Per sincronizzare l'accesso concorrente agli oggetti condivisi dai thread il kernel fornisce oggetti di sincronizzazione come semafori e mutex, che sono oggetti dispatcher, come descritto nel Paragrafo B.3.2.2. I thread possono anche essere sincronizzati con servizi del kernel che operano su oggetti kernel, come thread, processi e file, perché anche questi sono oggetti dispatcher. La sincronizzazione con oggetti dispatcher del kernel può essere ottenuta mediante l'uso delle funzioni `WaitForSingleObject()` e `WaitForMultipleObjects()`, che aspettano la segnalazione di uno o più oggetti dispatcher.

Un altro metodo di sincronizzazione è disponibile per i thread all'interno dello stesso processo che vogliono eseguire codice esclusivo. L'oggetto sezione critica Win32 è un oggetto mutex in modalità utente che può spesso essere acquisito e rilasciato senza entrare nel kernel. In un sistema multiprocessore, una sezione critica Win32 tenterà di ciclare continuamente in attesa del rilascio di una sezione critica detenuta da un altro thread. Se l'attesa dura troppo, il thread che vuole effettuare l'acquisizione allocherà un mutex kernel e cederà la sua cpu. Le sezioni critiche sono particolarmente efficienti perché il mutex kernel viene allocato solo quando c'è contesa e quindi utilizzato solo dopo aver tentato il continuo ciclare. La maggior parte dei mutex nei programmi non è mai effettivamente contesa: il risparmio è quindi notevole.

Prima di utilizzare una sezione critica, alcuni thread nel processo devono chiamare `InitializeCriticalSection()`. Ogni thread che voglia acquisire il mutex chiama `EnterCriticalSection()` e poi chiama `LeaveCriticalSection()` per rilasciare il mutex. Esiste anche una funzione `TryEnterCriticalSection()` che tenta di acquisire il mutex senza che il thread si blocchi.

Per i programmi che utilizzano lock di lettura-scrittura in modalità utente piuttosto che i mutex, Win32 supporta i lock swr (*slim reader-writer*). I lock swr hanno api simili a quelle per le sezioni critiche, come `InitializeSRWLock`, `AcquireSRWLockXXX` e `ReleaseSRWLockXXX`, dove XXX ha valore `Exclusive` o `Shared`, a seconda se il thread desidera l'accesso in scrittura o se è sufficiente l'accesso in lettura all'oggetto protetto da lock. L'api Win32 supporta anche le variabili condizionali, che possono essere utilizzate sia con le sezioni critiche sia con i lock swr.

B.7.3.5 Pool di thread

La frequente creazione e cancellazione di thread può imporre un alto costo alle applicazioni e ai servizi che usano ciascuno di loro per svolgere piccole quantità di lavoro. I pool di thread di Win32 offrono tre servizi ai programmi in modalità utente: una coda a cui si possono delegare le richieste di lavoro (tramite la funzione `SubmitThreadpoolWork()`), una api (`RegisterWaitForSingleObject()`) utilizzabile per legare le chiamate di ritorno agli handle in attesa e alcune api per lavorare con i timer (`CreateThreadpoolTimer()` e `WaitForThreadpoolTimerCallbacks()`) e per creare un legame con le chiamate di ritorno alle code di completamento di i/o (`BindIoCompletionCallback()`).

L'obiettivo dei pool di thread è migliorare le prestazioni e ridurre l'impatto della memoria: i thread sono relativamente costosi e un processore non può che eseguire un solo thread per volta, indipendentemente dal numero di thread disponibili. I pool di thread

tentano di ridurre il numero di thread eseguibili con un lieve differimento delle richieste di lavoro (riutilizzando ciascun thread per più richieste), fornendo al contempo thread a sufficienza per sfruttare efficacemente le cpu della macchina. Le api per la chiamata di ritorno associata ad attese o timer permettono al pool di thread di ridurre il numero di thread in un processo, usandone molto meno di quanti ne sarebbero necessari se un processo dovesse dedicare thread separati per servire l'attesa di un handle, di un timer o di una porta di completamento.

B.7.3.6 Fibre

Una **fibra** (*fiber*) è codice utente sottoposto a un regime di scheduling definito dall'utente. Le fibre lavorano interamente in modalità utente e il kernel non è a conoscenza della loro esistenza. Il meccanismo della fibra utilizza i thread di Windows come se fossero delle cpu per l'esecuzione delle fibre. Le fibre sono pianificate in maniera cooperativa, nel senso che non subiscono mai prelazione, ma devono cedere esplicitamente il thread su cui sono in esecuzione. Quando una fibra cede un thread, un'altra fibra può essere pianificata per l'esecuzione su quel thread dal sistema run-time (il codice run-time del linguaggio di programmazione).

Il sistema crea una fibra richiamando una tra le due funzioni `ConvertThreadToFiber()` o `CreateFiber()`; la differenza principale fra queste due funzioni è che la seconda non avvia l'esecuzione della fibra appena creata: per cominciare l'esecuzione l'applicazione deve richiamare la funzione `SwitchToFiber()`; per terminare l'esecuzione di una fibra l'applicazione deve richiamare la funzione `DeleteFiber()`.

Le fibre non sono raccomandate per i thread che utilizzano le api Win32 invece delle funzioni della libreria C standard a causa di potenziali incompatibilità. I thread Win32 in modalità utente possiedono un teb (*thread-environment block*) che contiene numerosi campi specifici del thread utilizzati dalle api Win32. Le fibre devono condividere il teb del thread su cui sono in esecuzione. Ciò può portare alcuni problemi quando un'interfaccia Win32 mette le informazioni di stato nel teb per una data fibra e in seguito le informazioni vengono sovrascritte da una fibra differente. Le fibre sono incluse nell'api Win32 per facilitare il porting di applicazioni legacy unix che sono state scritte per un modello di thread in modalità utente come Pthreads.

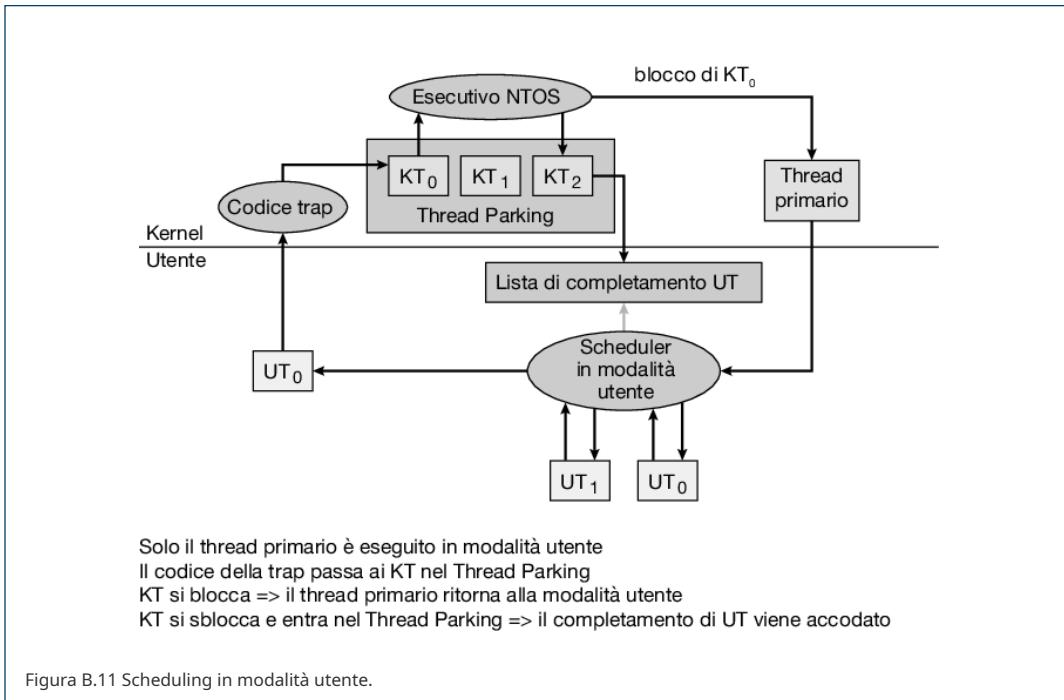
B.7.3.7 UMS e ConcRT

Un nuovo meccanismo di Windows 7, lo **scheduling in modalità utente** (*User-Mode Scheduling*, ums), risolve diversi limiti delle fibre. Si ricordi, prima di tutto, che le fibre sono inaffidabili per l'esecuzione di api Win32 perché non hanno propri teb. Quando un thread che esegue una fibra si blocca nel kernel, lo scheduler utente perde per un certo tempo il controllo della cpu, mentre il dispatcher del kernel si fa carico dello scheduling. I problemi possono verificarsi quando le fibre cambiano lo stato nel kernel di un thread, come i token di priorità o di personalizzazione, o quando iniziano un i/o asincrono.

ums fornisce un modello alternativo riconoscendo che ogni thread di Windows è in realtà formato da due thread: un thread del kernel (kt) e un thread utente (ut). Ogni tipo di thread ha un proprio stack e un proprio insieme di registri salvati. kt e ut appaiono come un singolo thread al programmatore perché i thread ut non si possono mai bloccare, ma devono sempre entrare nel kernel dove avviene un passaggio implicito al corrispondente kt. ums utilizza i teb di ogni ut per identificarli univocamente. Quando un ut entra nel kernel viene effettuato un passaggio esplicito al kt corrispondente all'ut identificato dal teb corrente. La ragione per cui il kernel non sa quale ut è in esecuzione è che i thread ut possono invocare uno scheduler in modalità utente, come fanno le fibre. In ums, però, lo scheduler scambia i thread ut, oltre a scambiare i teb.

Quando un ut entra nel kernel, il suo kt può essere bloccato. Quando ciò accade, il kernel passa a un thread di scheduling, che ums chiama thread primario, e usa questo thread per rientrare nello scheduler in modalità utente in modo da poter scegliere un altro ut da eseguire. Alla fine, un kt bloccato completerà le sue operazioni e sarà pronto per tornare alla modalità utente. Poiché ums ha già riattivato lo scheduler in modalità utente per eseguire un ut diverso, accoda l'ut corrispondente al kt completato a una lista di completamento in modalità utente. Quando lo scheduler in modalità utente deve scegliere un nuovo ut a cui passare, può esaminare la lista di completamento e trattare qualsiasi ut sulla lista come candidato per lo scheduling.

A differenza delle fibre, ums non è destinato all'utilizzo diretto da parte del programmatore. I dettagli della scrittura di uno scheduler in modalità utente possono essere molto impegnativi e ums non include un tale scheduler. Gli scheduler provengono invece dalle librerie di un linguaggio di programmazione costruite su ums. Microsoft Visual Studio 2010 dispone di Concurrency Runtime (Concrt), un ambiente di programmazione concorrente per C++. Concrt fornisce uno scheduler in modalità utente insieme a funzionalità per la scomposizione dei programmi in task che possono essere poi pianificati sulle cpu disponibili. Concrt fornisce il supporto per gli stili `par_for` dei costrutti, oltre a una gestione rudimentale delle risorse e a primitive di sincronizzazione dei task. Le caratteristiche principali di ums sono descritte nella Figura B.11.



B.7.3.8 Winsock

Winsock è l'api di Windows per i socket. Winsock è un'interfaccia a livello di sessione in gran parte compatibile con i socket unix, ma dotata di alcune estensioni aggiuntive di Windows. Winsock fornisce un'interfaccia standardizzata a molti protocolli di trasporto che possono avere diversi schemi di indirizzamento, in modo che qualsiasi applicazione Winsock possa funzionare su qualsiasi pila di protocolli compatibile con Winsock. Winsock ha subito un importante aggiornamento in Windows Vista con l'aggiunta del tracing, del supporto IPv6, della impersonazione, di nuove api di sicurezza e di molte altre caratteristiche.

Winsock segue il modello Windows Open System Architecture (wosa), che fornisce un'interfaccia del provider di servizi (spi) standard tra le applicazioni e i protocolli di rete. Le applicazioni possono caricare e scaricare i protocolli stratificati che costruiscono funzionalità addizionali, come una sicurezza aggiuntiva, sopra agli strati dei protocolli di trasporto. Winsock supporta operazioni asincrone e notifiche, multicasting affidabile, socket sicuri e socket in modalità kernel. C'è anche il supporto per modelli di utilizzo semplificati, come la funzione `wsaConnectByName()` che accetta la destinazione come una stringa che specifica il nome o l'indirizzo IP del server e il servizio o il numero di porta di destinazione.

B.7.4 Comunicazione fra processi mediante messaggi di Windows

Le applicazioni Win32 gestiscono la comunicazione tra processi in molti modi. Uno di essi è basato sulla condivisione degli oggetti del kernel; un altro metodo, particolarmente diffuso per le applicazioni con interfaccia utente grafica Win32, è basato sullo scambio di messaggi di Windows. Un thread può spedire un messaggio a un altro thread o a una finestra richiamando una fra le funzioni `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()` e `SendMessageCallback()`. La differenza fra le funzioni *Post* e quelle *Send* è che le prime sono asincrone: restituiscono immediatamente il controllo, quindi il thread chiamante non sa esattamente quando il messaggio giungerà a destinazione; le funzioni *Send*, invece, sono sincrone e bloccano quindi il chiamante finché il messaggio non sia stato recapitato.

Un thread può trasmettere dati insieme con un messaggio; in questo caso i dati devono essere copiati, perché processi diversi hanno spazi d'indirizzi distinti. Il sistema copia i dati richiamando `SendMessage()` per trasmettere un messaggio di tipo `WM_COPYDATA` con una struttura dati `COPYDATASTRUCT` contenente la lunghezza e l'indirizzo dei dati da trasferire; quando si trasmette il messaggio, il sistema operativo copia i dati in una nuova regione della memoria, e ne fornisce l'indirizzo virtuale al processo ricevente.

Ogni thread Win32 ha la propria coda d'ingresso dalla quale riceve messaggi. Si tratta di un'architettura più affidabile rispetto alla condivisione della coda d'ingresso dell'ambiente Windows a 16 bit, perché con le code distinte un'applicazione bloccata non impedisce la ricezione dei dati per le altre applicazioni. Se un'applicazione Win32 non impiega la funzione `GetMessage()` per trattare gli eventi nella sua coda d'ingresso, la coda si riempirà, e dopo circa 5 secondi il sistema contrasseggerà l'applicazione come "Non rispondente".

B.7.5 Gestione della memoria

L'api Win32 mette a disposizione delle applicazioni molti modi per utilizzare la memoria: la memoria virtuale, i file associati allo spazio d'indirizzi di memoria, gli *heap* e la memoria locale dei thread.

B.7.5.1 Memoria virtuale

Per prenotare o eseguire l'assegnazione di una regione della memoria virtuale, un'applicazione invoca la funzione `VirtualAlloc()`, e per eseguire le operazioni inverse invoca `VirtualFree()`. Queste funzioni permettono all'applicazione di specificare l'indirizzo virtuale a partire dal quale si deve assegnare la memoria; esse operano con multipli della dimensione della pagina di memoria. Alcuni esempi di queste funzioni sono riportati dalla Figura B.12.

```
// riserva 16 MB all'estremo superiore
// del nostro spazio di indirizzi
void *buf = VirtualAlloc(0,0x1000000, MEM_RESERVE |
    PAGE_READWRITE);
// effettua l'assegnazione degli 8 MB superiori
// dello spazio riservato
VirtualAlloc(buf + 0x800000,0x800000, MEM_COMMIT, PAGE_READWRITE);
// fa qualcosa
// annulla l'assegnazione
VirtualFree(buf + 0x800000,0x800000, MEM_DECOMMIT);
// rilascia tutto lo spazio assegnato
VirtualFree(buf,0, MEM_RELEASE);
```

Figura B.12 Frammenti di codice per l'assegnazione della memoria virtuale.

Un processo può fissare nella memoria fisica alcune sue pagine richiamando la funzione `VirtualLock()`; il massimo numero fissabile di pagine per processo è 30, sempre che il processo non richiami prima la funzione `SetProcessWorkingSetSize()` per aumentare la dimensione minima del working set.

B.7.5.2 Mappatura di file in memoria

Un'applicazione può anche far uso della memoria associando un file al proprio spazio d'indirizzi; si tratta anche di un metodo conveniente per la condivisione di memoria fra due processi: entrambi i processi associano lo stesso file alla loro memoria virtuale. La mappatura di un file è un procedimento a più fasi, come dimostra l'esempio della Figura B.13.

```
// apre il file o lo crea nel caso in cui non esista
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);
// crea uno spazio di 8 MB per la mappatura del file
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE, SEC_COMMIT,
    0, 0x800000, "SHM_1");
// ottiene una vista dello spazio mappato
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS, 0, 0, 0x800000);
// fa qualcosa
// annulla la mappatura del file
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);
```

Figura B.13 Frammenti di codice per la mappatura in memoria di un file.

Se un processo desidera mappare uno spazio d'indirizzi al solo scopo di condividere memoria con un altro processo, la presenza di un file è superflua: il processo può invocare `CreateFileMapping()` specificando come handle del file il valore `0xffffffff`, oltre a una certa dimensione; l'oggetto risultante si può condividere per ereditarietà, per nome o per duplicazione dell'handle.

B.7.5.3 Heap

Il terzo modo in cui un'applicazione può usare la memoria, analogo all'uso di `malloc()` e `free()` nel C standard, è rappresentato dallo heap. Uno *heap* nell'ambiente Win32 è semplicemente una regione riservata di uno spazio d'indirizzi. Un processo Win32 è dotato al momento della sua creazione di uno heap predefinito (*default heap*); poiché molte applicazioni Win32 sono multithread è necessario sincronizzare gli accessi allo heap per evitare che le sue strutture dati siano danneggiate da accessi concorrenti di thread diversi.

Win32 mette a disposizione le seguenti funzioni per la gestione e l'assegnazione di uno heap: `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()` e `HeapDestroy()`; l'api Win32 fornisce anche le funzioni `HeapLock()` e `HeapUnlock()` che garantiscono a un thread l'accesso esclusivo a uno heap; a differenza di `VirtualLock()`, queste funzioni eseguono solo la sincronizzazione, e non fissano pagine nella memoria fisica.

Lo heap originale Win32 era ottimizzato per un uso efficiente dello spazio. Ciò ha portato a notevoli problemi con la frammentazione dello spazio degli indirizzi nei programmi dei server di grandi dimensioni in esecuzione per lunghi periodi di tempo. Un nuovo progetto di heap a bassa frammentazione (lfh), introdotto in Windows xp, ha notevolmente ridotto il problema della frammentazione. Il gestore di heap di Windows 7 è in grado di attivare automaticamente lfh quando conviene.

B.7.5.4 Memoria locale dei thread

Un quarto modo in cui un'applicazione può usare la memoria è rappresentato dal meccanismo di memorizzazione locale dei thread (tls). Di solito le funzioni che fanno uso di dati statici o globali non operano correttamente in un ambiente multithread: la funzione run-time `strtok()` del linguaggio C, per esempio, impiega una variabile statica per tener traccia della posizione corrente durante la scansione di una sequenza di caratteri; affinché due thread concorrenti possano eseguire correttamente `strtok()` devono tenere distinte le variabili che mantengono la posizione corrente. tls fornisce un meccanismo per mantenere le istanze delle variabili globali per la funzione che viene eseguita, ma non condivise con alcun altro thread.

Il meccanismo tls può essere messo in atto sia tramite metodi dinamici sia tramite metodi statici. Il metodo dinamico è illustrato nella Figura B.14. Esso alloca spazio dello heap globale e lo collega al blocco di ambiente del thread che Windows assegna a ogni thread in modalità utente. Il teb è facilmente accessibile da ogni thread e viene utilizzato non solo per tls, ma per tutte le informazioni di stato in modalità utente specifiche dei thread.

```
// riserva spazio per una variabile
DWORD var_index = TlsAlloc();
// gli assegna il valore 10
TlsSetValue(var_index,10);
// lo rilegge
int var = TlsGetValue(var_index);
// rilascia l'indice
TlsFree(var_index);
```

Figura B.14 Codice per l'assegnazione dinamica di memoria locale a un thread.

Per usare in un thread una variabile statica locale, in modo da assicurare che ogni thread ne possieda una copia privata, un'applicazione dovrebbe dichiarare la variabile come segue:

```
-declspec(thread) DWORD cur_pos = 0;
```

B.8 Sommario

Il sistema operativo Windows è stato concepito da Microsoft come sistema operativo portabile ed estendibile, capace di trarre vantaggio dalle innovazioni nelle tecniche e nell'hardware; Windows gestisce l'elaborazione parallela simmetrica e supporta diversi ambienti operativi, tra cui processori sia a 32 sia a 64 bit e i calcolatori numa. L'uso degli oggetti del kernel per fornire i servizi fondamentali e la gestione del modello di elaborazione client-server permette a Windows di offrire un'ampia gamma di ambienti d'applicazione. Windows fornisce la gestione della memoria virtuale, servizi integrati di caching e scheduling con prelazione. Adotta un modello di sicurezza elaborato e comprende caratteristiche di internazionalizzazione. Windows è eseguibile su un'ampia varietà di calcolatori; in questo modo gli utenti possono scegliere e aggiornare i propri calcolatori, e i dispositivi che li compongono, in funzione delle loro disponibilità finanziarie e delle prestazioni richieste senza dover modificare le applicazioni eseguite.

Esercizi di ripasso

B.1 Che tipo di sistema operativo è Windows? Descrivete due delle sue caratteristiche principali.

B.2 Elencate gli obiettivi di progetto di Windows. Descrivetene due in dettaglio.

B.3 Descrivete il processo di avvio di un sistema Windows.

B.4 Descrivete i tre livelli principali dell'architettura del kernel di Windows.

B.5 Che lavoro svolge il gestore degli oggetti?

B.6 Che tipi di servizio offre il gestore dei processi?

B.7 Che cos'è una chiamata di procedura locale?

B.8 Quali sono le responsabilità del gestore dell'i/o?

B.9 Quali tipologie di networking sono supportate da Windows? Come sono implementati i protocolli di trasporto da Windows? Descrivete due protocolli di rete.

B.10 Descrivete l'organizzazione dello spazio dei nomi di ntfs.

B.11 Come tratta le strutture dati ntfs? Come viene ripristinato ntfs dopo un crash di sistema? Che cosa viene garantito dopo che è avvenuto un ripristino?

B.12 Come alloca la memoria utente Windows?

B.13 Descrivete alcuni dei modi in cui un'applicazione utilizza la memoria per mezzo della api Win32.

Esercizi

B.14 Quando e perché si dovrebbe usare la funzionalità della procedura di chiamata differita in Windows?

B.15 Che cos'è un handle, e come fa un processo a ottenerne uno?

B.16 Descrivete il funzionamento del gestore della memoria virtuale. In che modo il gestore della vm può migliorare le prestazioni?

B.17 Descrivete un'applicazione utile della funzionalità che vieta l'accesso alle pagine fornita da Windows.

B.18 Descrivete le tre tecniche previste per trasmettere i dati in una procedura di chiamata locale. Quali differenti contesti portano all'applicazione dell'una o dell'altra tecnica di scambio dei messaggi?

B.19 Da quale entità è gestita la cache di Windows? E con quali modalità?

B.20 Per quali aspetti la struttura della directory ntfs differisce da quella adottata dai sistemi operativi unix?

B.21 Che cos'è un processo, e in che modo il sistema Windows lo gestisce?

B.22 Che cosa sono le “fibre” disponibili in Windows? In che cosa differiscono dai thread?

B.23 In che cosa lo scheduling in modalità utente (ums) di Windows 7 differisce dalle fibre? Individuate alcuni compromessi tra fibre e ums.

B.24 ums ritiene che il thread sia composto da due parti, un ut e un kt. In che cosa potrebbe essere utile consentire agli ut di continuare l'esecuzione in parallelo con i loro kt?

B.25 Quali sono i vantaggi e gli svantaggi in termini di prestazioni nel permettere ai kt e agli ut di essere eseguiti su diversi processori?

B.26 Perché l'auto-mappa occupa grandi quantità di spazio degli indirizzi virtuali, ma non occupa memoria virtuale aggiuntiva?

B.27 In che modo l'auto-mappa facilita al gestore della vm lo spostamento delle pagine della tabella delle pagine da e verso il disco? Dove sono conservate le pagine della tabella delle pagine sul disco?

B.28 Quando un sistema Windows si sospende, il sistema viene spento. Supponiamo di sostituire la cpu o aumentare la quantità di ram su un sistema che è stato sospeso. Il sistema potrebbe funzionare? Motivate la risposta.

B.29 Fornite un esempio che mostra come l'uso di un conteggio di sospensione è utile nella sospensione e nella ripresa dei thread in Windows.