



# Sistemi operativi

Il volume si pone l'obiettivo di illustrare in modo chiaro e allo stesso tempo semplice i concetti e le tecniche fondamentali dei sistemi operativi, iniziando il lettore ai principi base e, successivamente, avanzati dei sistemi operativi e non alla sola conoscenza dei sistemi Linux, Windows, Vista e Solaris. Le interazioni del sistema operativo con l'elaboratore da una parte e gli utenti dall'altra sono delineate attraverso un crescendo di dettagli pratici, di illustrazioni, di esempi e di esercizi svolti.

I numerosi casi di studio, inoltre, presentano agli studenti l'analisi di alcune esperienze operative che potranno incontrare nel corso della loro carriera.

Ogni capitolo è completato da un ricco apparato di esercizi, problemi avanzati, esercizi di laboratorio e domande che permettono allo studente di consolidare e verificare il proprio livello di apprendimento.

Per stabilire un forte legame con le esigenze dei corsi di Informatica e di Ingegneria informatica degli atenei italiani, nell'edizione italiana sono stati approfonditi alcuni concetti e tecniche.

All'indirizzo web [www.ateneonline.it/dhamhare](http://www.ateneonline.it/dhamhare) sono disponibili materiali di supporto: per i docenti i lucidi in formato PowerPoint, per gli studenti le soluzioni di tutti i problemi del testo.

**collana di istruzione scientifica**  
serie di informatica

Dhananjay M. Dhamdhere

## **Sistemi operativi**

Edizione italiana a cura di  
Alfredo Petrosino

Traduzione a cura di  
Alfredo Petrosino e Graziano Pravadelli

**McGraw-Hill**

---

Milano • New York • San Francisco • Washington D.C. • Auckland  
Bogotà • Lisboa • London • Madrid • Mexico City • Montreal  
New Delhi • San Juan • Singapore • Sydney • Tokyo • Toronto

Copyright © 2014, McGraw-Hill Education (Italy) S.r.l. Via Ripamonti, 89 - 20141 Milano.

Titolo originale:

*OPERATING SYSTEM: A CONCEPT-BASED APPROACH*

Copyright edizione a  
stampa © 2009

The McGraw-Hill Companies, Inc. All right reserved

Copyright edizione a  
stampa © 2010

The McGraw-Hill Companies, S.r.l.  
Publishing Group Italia  
Via Ripamonti, 89 - 20139 Milano

**McGraw-Hill**

*A Division of The McGraw-Hill Companies*



I diritti di traduzione, di riproduzione, di memorizzazione elettronica e di adattamento totale e parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

La stampa dell'opera è consentita esclusivamente per uso personale.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

Editor: Paolo Roncoroni

Traduzione: Alfredo Petrosino (Capitoli 1-15), Graziano Pravadelli (Capitoli 16-20)

Produzione: Donatella Giuliani

Grafica di copertina: Editta Gelsomini

Realizzazione ePUB: codeMantra

ISBN 978-883-8691-76-8

---

# Indice breve

---

**Parte 1 - Panoramica**

- 1 Introduzione
- 2 Il SO, il computer e i programmi utente
- 3 Panoramica dei sistemi operativi
- 4 Struttura dei sistemi operativi

**Parte 2 - Gestione dei processi**

- 5 Processi e thread
- 6 Sincronizzazione dei processi: memoria condivisa
- 7 Scheduling
- 8 Deadlock
- 9 Sincronizzazione dei Processi: message passing
- 10 Sincronizzazione e scheduling nei sistemi operativi multiprocessore

**Parte 3 - Gestione della memoria**

- 11 Gestione della memoria
- 12 Memoria virtuale

**Parte 4 - File system e gestione dell'I/O**

- 13 File system
- 14 Implementazione delle operazioni su file
- 15 Sicurezza e protezione

**Parte 5 - Sistemi operativi distribuiti**

- 16 Sistemi operativi distribuiti
- 17 Problematiche teoriche di un sistema distribuito
- 18 Algoritmi di controllo distribuiti
- 19 File system distribuiti
- 20 Sicurezza nei sistemi distribuiti

---

# Indice generale

---

## Prefazione

## Ringraziamenti dell'Editore

## Guida alla lettura

## Parte 1 - Panoramica

### 1 Introduzione

- 1.1 Viste di un sistema operativo
- 1.2 Obiettivi di un SO
  - 1.2.1 Uso efficiente
  - 1.2.2 Convenienza per l'utente
  - 1.2.3 Assenza di interferenze
- 1.3 Funzionamento di un SO
  - 1.3.1 Gestione dei programmi
  - 1.3.2 Gestione delle risorse
  - 1.3.3 Sicurezza e protezione
- 1.4 Introduzione del libro
  - 1.4.1 Introduzione ai sistemi operativi
  - 1.4.2 Gestire le elaborazioni degli utenti
  - 1.4.3 Gestione della memoria
  - 1.4.4 Gestione dei file e dei dispositivi di I/O
  - 1.4.5 Sistemi operativi distribuiti

Riepilogo

Domande

Problemi

Note bibliografiche

### 2 Il SO, il computer e i programmi utente

- 2.1 Principi fondamentali del funzionamento di un SO
- 2.2 Il computer
  - 2.2.1 La CPU
  - 2.2.2 Memory Management Unit (MMU)
  - 2.2.3 Gerarchia della memoria
  - 2.2.4 Input/Output
  - 2.2.5 Interrupt
- 2.3 Interazione del SO con il computer e i programmi utente
  - 2.3.1 Controllare l'esecuzione dei programmi
  - 2.3.2 Servire gli interrupt
  - 2.3.3 System call

Riepilogo

Domande

Problemi

Note bibliografiche

### 3 Panoramica dei sistemi operativi

- 3.1 Ambienti di elaborazione e natura delle elaborazioni
- 3.2 Classi di sistemi operativi
- 3.3 Efficienza, prestazioni del sistema e servizio per l'utente
- 3.4 Sistemi di elaborazione batch

- 3.5 Sistemi multiprogrammati
    - 3.5.1 Priorità dei programmi
  - 3.6 Sistemi time-sharing
    - 3.6.1 Swapping dei programmi
  - 3.7 Sistemi operativi real-time
    - 3.7.1 Sistemi hard e soft real-time
    - 3.7.2 Caratteristiche di un sistema operativo real-time
  - 3.8 Sistemi operativi distribuiti
    - 3.8.1 Tecniche speciali di sistemi operativi distribuiti
  - 3.9 Moderni sistemi operativi
- Riepilogo  
Domande  
Problemi  
Note bibliografiche

#### **4 Struttura dei sistemi operativi**

- 4.1 Funzionamento di un SO
  - 4.2 Struttura di un sistema operativo
    - 4.2.1 Politiche e meccanismi
    - 4.2.2 Portabilità ed espandibilità dei sistemi operativi
  - 4.3 Sistemi operativi con struttura monolitica
  - 4.4 Sistemi operativi strutturati a livelli
  - 4.5 Macchina virtuale e sistemi operativi
  - 4.6 Sistemi operativi basati su kernel
    - 4.6.1 Evoluzione della struttura basata su kernel
  - 4.7 Sistemi operativi basati su microkernel
  - 4.8 Casi di studio
    - 4.8.1 Architettura di Unix
    - 4.8.2 Il Kernel di Linux
    - 4.8.3 Il Kernel di Solaris
    - 4.8.4 Architettura di Windows
- Riepilogo  
Domande  
Problemi  
Note bibliografiche

### **Parte 2 - Gestione dei processi**

#### **5 Processi e thread**

- 5.1 Processi e programmi
  - 5.1.1 Cosa è un processo?
  - 5.1.2 Relazioni tra processi e programmi
  - 5.1.3 Processi figli
  - 5.1.4 Concorrenza e parallelismo
- 5.2 Implementazione dei processi
  - 5.2.1 Stati del processo e transizioni di stato
  - 5.2.2 Il contesto del processo e il process control block
  - 5.2.3 Salvataggio del contesto, scheduling e dispatching
  - 5.2.4 Gestione degli eventi
  - 5.2.5 Condivisione, comunicazione e sincronizzazione tra processi
  - 5.2.6 Segnali
- 5.3 Thread
  - 5.3.1 Thread POSIX
  - 5.3.2 Thread di livello kernel, di livello utente e ibridi
- 5.4 Casi di studio: processi e thread
  - 5.4.1 Processi in Unix
  - 5.4.2 Processi e thread in Linux
  - 5.4.3 Thread in Solaris
  - 5.4.4 Processi e thread in Windows

Riepilogo  
Domande  
Problemi  
Laboratorio: implementare una shell  
Note bibliografiche

## 6 Sincronizzazione dei processi: memoria condivisa

6.1 Che cos'è la sincronizzazione dei processi?  
6.2 Race condition  
6.3 Sezioni critiche  
    6.3.1 Proprietà dell'implementazione di una sezione critica  
6.4 Controllo della sincronizzazione e operazioni indivisibili  
6.5 Approcci alla sincronizzazione  
    6.5.1 Ciclare o bloccare  
    6.5.2 Supporto hardware per la sincronizzazione dei processi  
    6.5.3 Approcci algoritmici, primitive di sincronizzazione e costrutti di programmazione concorrente  
6.6 Struttura dei sistemi concorrenti  
6.7 Problemi classici di sincronizzazione dei processi  
    6.7.1 Produttori-consumatori con buffer limitati  
    6.7.2 Lettori e scrittori  
    6.7.3 I filosofi a cena  
6.8 Approccio algoritmico per implementare le sezioni critiche  
    6.8.1 Algoritmi per due processi  
    6.8.2 Algoritmi per  $n$  processi  
6.9 Semafori  
    6.9.1 Uso dei semafori nei sistemi concorrenti  
    6.9.2 Produttori-consumatori usando i semafori  
    6.9.3 Lettori-scrittori utilizzando i semafori  
    6.9.4 Implementazione dei semafori  
6.10 Un esempio di problema di sincronizzazione  
    6.10.1 Soluzione  
    6.10.2 Il codice C  
6.11 Monitor  
    6.11.1 Monitor in Java  
6.12 Casi di studio relativi alla sincronizzazione dei processi  
    6.12.1 Sincronizzazione dei thread POSIX  
    6.12.2 Sincronizzazione dei processi in Unix  
    6.12.3 Sincronizzazione dei processi in Linux  
    6.12.4 Sincronizzazione dei processi in Solaris  
    6.12.5 Sincronizzazione dei processi in Windows

Riepilogo  
Domande  
Problemi  
Laboratorio 1: sincronizzazione  
Laboratorio 2: comunicazione tra processi  
Laboratorio 3: scheduler del disco  
Note bibliografiche

## 7 Scheduling

7.1 Scheduling: terminologia e concetti  
    7.1.1 Tecniche fondamentali di scheduling  
    7.1.2 Il ruolo della priorità  
7.2 Politiche di scheduling nonpreemptive  
    7.2.1 Scheduling FCFS  
    7.2.2 Scheduling Shortest Job First (SJF)  
    7.2.3 Scheduling Highest Response Ratio Next (HRN)  
7.3 Politiche di scheduling preemptive

- 7.3.1 Scheduling Round-Robin con Time-Slicing (RR)
  - 7.3.2 Scheduling Least Completed Next (LCN)
  - 7.3.3 Scheduling Shortest Time to Go (STG)
  - 7.4 Scheduling in pratica
    - 7.4.1 Scheduler a lungo, medio e breve termine
    - 7.4.2 Strutture dati e meccanismi di scheduling
    - 7.4.3 Scheduling basato su priorità
    - 7.4.4 Scheduling Round-Robin con Time-Slicing
    - 7.4.5 Scheduling multilivello
    - 7.4.6 Scheduling fair share
    - 7.4.7 Prelazionabilità del kernel
    - 7.4.8 Scheduling: euristiche
    - 7.4.9 Gestione del consumo di energia
  - 7.5 Scheduling real-time
    - 7.5.1 Precedenze e schedulabilità di un processo
    - 7.5.2 Scheduling della deadline
    - 7.5.3 Scheduling rate monotonic
  - 7.6 Casi di studio
    - 7.6.1 Scheduling in Unix
    - 7.6.2 Scheduling in Solaris
    - 7.6.3 Scheduling in Linux
    - 7.6.4 Scheduling in Windows
  - 7.7 Analisi delle prestazioni delle politiche di scheduling
    - 7.7.1 Analisi della prestazione attraverso implementazione
    - 7.7.2 Simulazione
    - 7.7.3 Modellazione matematica
- Riepilogo  
 Domande  
 Problemi  
 Problemi avanzati  
 Note bibliografiche

## **8 Deadlock**

- 8.1 Definizione di deadlock
- 8.2 I deadlock nell'allocazione delle risorse
  - 8.2.1 Condizioni per un deadlock di risorsa
  - 8.2.2 Modelli dello stato di allocazione delle risorse
- 8.3 Gestione dei deadlock
- 8.4 Individuazione e risoluzione dei deadlock
  - 8.4.1 Un algoritmo di individuazione dei deadlock
  - 8.4.2 Risoluzione dei deadlock
- 8.5 Prevenzione dei deadlock
  - 8.5.1 Allocazione globale
  - 8.5.2 Ranking delle risorse
- 8.6 Evitare i deadlock
- 8.7 Algoritmo del banchiere con allocazioni variabili
  - 8.7.1 Soluzione
- 8.8 Caratterizzazione dei deadlock delle risorse tramite i modelli a grafi
  - 8.8.1 Sistemi Singola-Istanza, Singola-Richiesta (SISR)
  - 8.8.2 Sistemi Istanza-Multipla, Singola-Richiesta (MISR)
  - 8.8.3 Sistemi Singola-Istanza, Richiesta-Multipla (SIMR)
  - 8.8.4 Sistemi Istanza-Multipla, Richiesta-Multipla (MIMR)
  - 8.8.5 Processi in deadlock
- 8.9 Casi di studio
  - 8.9.1 Gestione dei deadlock in Unix
  - 8.9.2 Gestione dei deadlock in Windows

- Riepilogo  
 Domande

Problemi  
Problemi avanzati  
Note bibliografiche

## **9 Sincronizzazione dei processi: message passing**

9.1 Panoramica sul message passing  
    9.1.1 Denominazione diretta e indiretta  
    9.1.2 Send bloccanti e non-bloccanti  
    9.1.3 Eccezioni nel message passing  
9.2 Implementazione del message passing  
    9.2.1 Buffering dei messaggi tra processi  
    9.2.2 Consegnare dei messaggi tra processi  
9.3 Mailbox  
9.4 Protocolli di alto livello che utilizzano il message passing  
    9.4.1 Il Simple Mail Transfer Protocol (SMTP)  
    9.4.2 Remote Procedure Call (RPC)  
    9.4.3 Standard del message passing per la programmazione parallela  
9.5 Casi di studio  
    9.5.1 Message passing in Unix  
    9.5.2 Message passing in Windows  
Riepilogo  
Domande  
Problemi  
Note bibliografiche

## **10 Sincronizzazione e scheduling nei sistemi operativi multiprocessore**

10.1 Architettura dei sistemi multiprocessore  
    10.1.1 Architettura SMP  
    10.1.2 Architettura NUMA  
10.2 Problematiche nei sistemi operativi multiprocessore  
10.3 Struttura del kernel  
10.4 Sincronizzazione dei processi  
    10.4.1 Hardware speciale per la sincronizzazione dei processi  
    10.4.2 Uno schema di software scalabile per la sincronizzazione dei processi  
10.5 Scheduling dei processi  
10.6 Casi di studio  
    10.6.1 Mach  
    10.6.2 Linux  
    10.6.3 Supporto SMP in Windows  
Riepilogo  
Domande  
Problemi  
Note bibliografiche

## **Parte 3 - Gestione della memoria**

### **11 Gestione della memoria**

11.1 Gestione della gerarchia di memoria  
11.2 Allocazione statica e dinamica della memoria  
11.3 Esecuzione dei programmi  
    11.3.1 Rilocazione  
    11.3.2 Linking  
    11.3.3 Tipologie di programmi utilizzate nei sistemi operativi  
11.4 Allocazione della memoria a un processo  
    11.4.1 Stack e heap  
    11.4.2 Il Modello di allocazione della memoria  
    11.4.3 Protezione della memoria  
11.5 Gestione dell'heap  
    11.5.1 Riuso della memoria

- 11.5.2 Buddy system e allocatori potenza del 2
  - 11.5.3 Confronto tra allocatori di memoria
  - 11.5.4 Gestione dell'heap in Windows
  - 11.6 Allocazione contigua della memoria
  - 11.7 Allocazione non contigua della memoria
    - 11.7.1 Indirizzo logico, indirizzo fisico e traduzione di indirizzo
    - 11.7.2 Approcci all'allocazione non contigua della memoria
    - 11.7.3 Protezione della memoria
  - 11.8 Paginazione
  - 11.9 Segmentazione
  - 11.10 Segmentazione con paginazione
  - 11.11 Allocazione della memoria del kernel
  - 11.12 Utilizzo efficiente della RAM idle
- Riepilogo  
 Domande  
 Problemi  
 Problemi avanzati  
 Note bibliografiche

## 12 Memoria virtuale

- 12.1 Principi di base della memoria virtuale
  - 12.2 Paginazione su richiesta
    - 12.2.1 Paginazione su richiesta: concetti preliminari
    - 12.2.2 Supporto hardware alla paginazione
    - 12.2.3 Organizzazione pratica delle tabelle delle pagine
    - 12.2.4 Operazioni di I/O in ambiente paginato
  - 12.3 Il gestore della memoria virtuale
    - 12.3.1 Panoramica del funzionamento del gestore della memoria virtuale
  - 12.4 Politiche di sostituzione delle pagine
    - 12.4.1 Esempi realistici di politiche di sostituzione delle pagine
  - 12.5 Controllo dell'allocazione di memoria per un processo
  - 12.6 Pagine condivise
    - 12.6.1 Copy-on-write
  - 12.7 File mappati in memoria
  - 12.8 Casi di studio
    - 12.8.1 Memoria virtuale in Unix
    - 12.8.2 Memoria virtuale in Linux
    - 12.8.3 Memoria virtuale in Solaris
    - 12.8.4 Memoria virtuale in Windows
  - 12.9 Memoria virtuale con uso di segmentazione
    - 12.9.1 Gestione della memoria
    - 12.9.2 Condivisione e protezione
    - 12.9.3 Segmentazione con paginazione
- Riepilogo  
 Domande  
 Problemi  
 Laboratorio: simulazione di un gestore della memoria virtuale  
 Problemi avanzati  
 Note bibliografiche

## Parte 4 - File system e gestione dell'I/O

### 13 File system

- 13.1 Elaborazione dei file: una panoramica
  - 13.1.1 File system e IOCS
  - 13.1.2 Elaborazione dei file in un programma
- 13.2 File e operazioni sui file
- 13.3 Fondamenti dell'organizzazione dei file e metodi di accesso
  - 13.3.1 Organizzazione sequenziale dei file

- 13.3.2 Organizzazione diretta dei file
  - 13.3.3 Organizzazione dei file indicizzata
  - 13.3.4 Metodi di accesso
  - 13.4 Directory
    - 13.4.1 Alberi di directory
    - 13.4.2 Grafi di directory
    - 13.4.3 Operazioni sulle directory
    - 13.4.4 Organizzazione delle directory
  - 13.5 Montaggio dei file system
  - 13.6 Protezione dei file
  - 13.7 Allocazione dello spazio sul disco
    - 13.7.1 Allocazione concatenata
    - 13.7.2 Allocazione indicizzata
    - 13.7.3 Il problema delle prestazioni
  - 13.8 Interfaccia tra file system e IOCS
  - 13.9 Elaborazione dei file
    - 13.9.1 Azioni del file system per la `open`
    - 13.9.2 Azioni del file system durante un'operazione su file
    - 13.9.3 Azioni del file system per la `close`
  - 13.10 Semantiche della condivisione dei file
  - 13.11 Affidabilità del file system
    - 13.11.1 Perdita di consistenza del file system
    - 13.11.2 Approcci all'affidabilità del file system
  - 13.12 Journaling file system
  - 13.13 File system virtuale
  - 13.14 Casi di studio
    - 13.14.1 File system Unix
    - 13.14.2 File system di Linux
    - 13.14.3 File system di Solaris
    - 13.14.4 File system di Windows
  - 13.15 Prestazioni dei file system
    - 13.15.1 File system di tipo log-structured
- Riepilogo  
 Domande  
 Problemi  
 Problemi avanzati  
 Note bibliografiche

## **14 Implementazione delle operazioni su file**

- 14.1 Livelli dell'input-output control system
- 14.2 Panoramica dell'organizzazione dell'I/O
- 14.3 Dispositivi di I/O
  - 14.3.1 Nastri magnetici
  - 14.3.2 Dischi magnetici
  - 14.3.3 Tecniche di organizzazione dei dati
  - 14.3.4 Tecnologie di collegamento dei dischi
  - 14.3.5 RAID
  - 14.3.6 Dischi ottici
- 14.4 I/O a livello di dispositivo
  - 14.4.1 Programmazione dell'I/O
- 14.5 IOCS fisico
  - 14.5.1 Dispositivi logici
  - 14.5.2 Strutture dati del IOCS fisico
  - 14.5.3 Organizzazione del IOCS fisico
  - 14.5.4 Implementazione del IOCS fisico
- 14.6 Driver di dispositivo
- 14.7 Scheduling del disco
- 14.8 Tempo di trasferimento nello scheduling del disco

- 14.8.1 Soluzione
- 14.9 Buffering dei record
- 14.10 Blocking dei record
- 14.11 Metodi di accesso
- 14.12 Cache del disco e dei file
- 14.13 Cache del disco unificata
- 14.14 Casi di studio
  - 14.14.1 Unix
  - 14.14.2 Linux
  - 14.14.3 Windows

Riepilogo  
Domande  
Problemi  
Problemi avanzati  
Note bibliografiche

## 15 Sicurezza e protezione

- 15.1 Sicurezza e protezione: introduzione
  - 15.1.1 Obiettivi di sicurezza e protezione
  - 15.1.2 Minacce alla sicurezza e alla protezione
- 15.2 Attacchi alla sicurezza
  - 15.2.1 Cavalli di Troia, virus e worm
  - 15.2.2 Tecnica del buffer overflow
- 15.3 Aspetti formali di sicurezza
- 15.4 Cifratura
  - 15.4.1 Attacchi ai sistemi di crittografia
  - 15.4.2 Tecniche di cifratura
- 15.5 Autenticazione e sicurezza delle password
- 15.6 Strutture di protezione
  - 15.6.1 Granularità della protezione
  - 15.6.2 Matrice di controllo degli accessi
  - 15.6.3 Liste di controllo degli accessi (ACL)
  - 15.6.4 Capability lists (C-list)
  - 15.6.5 Dominio di protezione
- 15.7 Capabilities
  - 15.7.1 Sistemi basati su capability
  - 15.7.2 Capability software
  - 15.7.3 Aree critiche nell'uso delle capability
- 15.8 Classificazione della sicurezza informatica
- 15.9 Casi di studio
  - 15.9.1 MULTICS
  - 15.9.2 Unix
  - 15.9.3 Linux
  - 15.9.4 Windows

Riepilogo  
Domande  
Problemi  
Note bibliografiche

## Parte 5 - Sistemi operativi distribuiti

### 16 Sistemi operativi distribuiti

- 16.1 Caratteristiche dei sistemi distribuiti
- 16.2 Nodi di un sistema distribuito
- 16.3 Integrazione dell'attività dei nodi di un sistema distribuito
- 16.4 Comunicazione affidabile tra processi
  - 16.4.1 Naming dei processi
  - 16.4.2 Semantica dei protocolli di comunicazione
  - 16.4.3 Protocolli IPC

- 16.5 Paradigmi di calcolo distribuito
  - 16.5.1 Elaborazione client-server
  - 16.5.2 Chiamate a procedura remota
  - 16.5.3 Remote evaluation
  - 16.5.4 Casi di studio
- 16.6 Networking
  - 16.6.1 Tipi di rete
  - 16.6.2 Topologia di rete
  - 16.6.3 Tecnologie di rete
  - 16.6.4 Strategie di connessione
  - 16.6.5 Instradamento
  - 16.6.6 Protocolli di rete
    - 16.6.7 Larghezza e latenza della banda di rete
- 16.7 Modello di un sistema distribuito
- 16.8 Problematiche progettuali di un sistema operativo distribuito
- Riepilogo
- Domande di riepilogo
- Problemi
- Note bibliografiche

## **17 Problematiche teoriche di un sistema distribuito**

- 17.1 Le nozioni di tempo e stato
- 17.2 Stati ed eventi in un sistema distribuito
  - 17.2.1 Stato locale e stato globale
  - 17.2.2 Gli eventi
- 17.3 Tempo, clock e precedenze tra eventi
  - 17.3.1 Precedenza tra eventi
  - 17.3.2 Clock logici
  - 17.3.3 Vettori di clock
- 17.4 Registrazione dello stato di un sistema distribuito
  - 17.4.1 Proprietà di uno state recording consistente
  - 17.4.2 Un algoritmo per la registrazione di state recording consistenti
- Riepilogo
- Domande
- Problemi
- Note bibliografiche

## **18 Algoritmi di controllo distribuiti**

- 18.1 Funzioni degli algoritmi di controllo distribuiti
- 18.2 Correttezza degli algoritmi di controllo distribuiti
- 18.3 Mutua esclusione distribuita
  - 18.3.1 L'algoritmo di Ricart e Agrawala
  - 18.3.2 Algoritmi di mutua esclusione che utilizzano un token
- 18.4 Gestione dei deadlock distribuiti
  - 18.4.1 Problemi delle tecniche centralizzate per la rilevazione dei deadlock
  - 18.4.2 Rilevazione dei deadlock distribuiti
  - 18.4.3 Prevenzione dei deadlock distribuiti
- 18.5 Algoritmi di schedulazione distribuiti
- 18.6 Rilevazione distribuita della conclusione
- 18.7 Algoritmi di elezione
- 18.8 Problemi pratici nell'uso degli algoritmi di controllo distribuiti
  - 18.8.1 Gestione delle risorse
  - 18.8.2 Migrazione dei processi
- Riepilogo
- Domande
- Problemi
- Note bibliografiche

## **19 File system distribuiti**

- 19.1 Problematiche di progetto in un file system distribuito
    - 19.1.1 Funzionalità di un file system distribuito
  - 19.2 Trasparenza
  - 19.3 Semantica della condivisione dei file
  - 19.4 Tolleranza ai guasti
    - 19.4.1 Disponibilità
    - 19.4.2 Guasti su client e server
    - 19.4.3 File server stateless
  - 19.5 Prestazioni di un FSD
    - 19.5.1 Accesso a file efficiente
    - 19.5.2 File caching
    - 19.5.3 Scalabilità
  - 19.6 Casi di studio
    - 19.6.1 Il network file system di Sun
    - 19.6.2 I file system Andrew e Coda
    - 19.6.3 GPFS
    - 19.6.4 Windows
- Riepilogo
- Domande
- Problemi
- Note bibliografiche

## **20 Sicurezza nei sistemi distribuiti**

- 20.1 Problematiche di sicurezza in un sistema operativo distribuito
    - 20.1.1 Politiche e meccanismi per garantire la sicurezza
    - 20.1.2 Attacchi alla sicurezza di un sistema distribuito
  - 20.2 Sicurezza dei messaggi
    - 20.2.1 Distribuzione delle chiavi
    - 20.2.2 Prevenzione degli attacchi basati su replicazione dei messaggi
    - 20.2.3 Autenticazione reciproca
  - 20.3 Autenticazione di dati e messaggi
    - 20.3.1 Autorità di certificazione e certificati digitali
    - 20.3.2 Codici per l'autenticazione dei messaggi e firme digitali
  - 20.4 Autenticazione basata su entità esterne
    - 20.4.1 Kerberos
    - 20.4.2 Secure sockets layer (SSL)
- Riepilogo
- Domande
- Problemi
- Note bibliografiche

## **Indice analitico**

---

# Prefazione

---

## Obiettivo

L'obiettivo principale di un corso introduttivo sui sistemi operativi è spiegarne i concetti e le tecniche fondamentali. Il raggiungimento di questo obiettivo è complicato dallo scenario attuale, in cui gli studenti acquisiscono numerose, e talvolta non corrette, informazioni da Internet, senza essere in grado di organizzarle in modo strutturato. Risulta pertanto necessario fornire loro i concetti basati sulle informazioni che hanno già acquisito senza rendere la presentazione troppo accademica e, contestualmente, presentare gli stessi in maniera chiara a coloro che si avvicinano per la prima volta allo studio dei sistemi operativi. L'intento di questo libro è quello di soddisfare entrambe queste esigenze.

L'approccio scelto è quello di spiegare in modo rigoroso i concetti alla base di un sistema operativo, presentato come un intermediario tra elaboratore e utenti in grado di offrire un buon servizio a questi ultimi utilizzando in modo efficiente le risorse del primo. Le interazioni SO-elaboratore da un lato e SO-utenti dall'altro sono descritte con un crescendo di dettagli pratici, mettendo in risalto quelle caratteristiche chiave dell'architettura degli elaboratori essenziali per lo studio dei sistemi operativi.

In questa chiave il testo si caratterizza per un'impostazione didattica formale ma al contempo ricca di esempi ed esercizi svolti, con richiami e suggerimenti orientati all'efficienza e alla professionalità, perseguitando in tal modo l'obiettivo dichiarato di iniziare il lettore ai principi base e avanzati dei sistemi operativi e non alla sola conoscenza dei sistemi Linux, Windows, Vista e Solaris.

Il testo analizza in modo comparativo le organizzazioni, la funzionalità, i meccanismi, le politiche e la gestione dei sistemi operativi per le varie architetture dei sistemi di elaborazione, sia monoprocessoresso sia multiprocessoresso, e solo successivamente quella orientata ai sistemi distribuiti, approfondendo sempre gli aspetti di progettazione, di configurazione, di ottimizzazione e di manutenzione dei sistemi operativi moderni.

Gli aspetti caratteristici di questo approccio sono:

- i concetti fondamentali sono presentati in termini semplici;
- vengono immediatamente stabilite le relazioni tra concetti e tecniche;
- numerosi esempi illustrano concetti e tecniche;
- i dettagli implementativi e casi di studio sono distribuiti in tutto il testo.

L'edizione italiana è stata arricchita con ulteriori esempi e problemi con l'obiettivo di illustrare aspetti più approfonditi dei concetti e delle tecniche presentate, con un forte legame con le esigenze dei corsi istituzionali di Informatica e di Ingegneria informatica degli atenei italiani.

## Organizzazione del libro

Il testo è composto di cinque parti, ciascuna delle quali è caratterizzata dai seguenti obiettivi.

**Parte 1 - Panoramica** Questa parte è composta da quattro capitoli. Il [Capitolo 1](#) spiega come la convenienza per l'utente, l'uso efficiente delle risorse, la sicurezza e la protezione siano gli elementi fondamentali di un sistema operativo e descrive i passi necessari per implementarli. Include anche una sintesi degli argomenti trattati nel libro. Il [Capitolo 2](#) spiega come un sistema operativo usa le funzionalità dell'hardware di un

elaboratore per organizzare l'esecuzione dei programmi degli utenti e gestire le loro richieste. Il [Capitolo 3](#) descrive le varie classi di sistemi operativi, illustra i concetti e le tecniche fondamentali di ciascuna classe ed elenca le tecniche che sono adottate in quelli moderni. Il [Capitolo 4](#) descrive le metodologie di progettazione che consentono a un sistema operativo di adattarsi alle differenti architetture dei calcolatori e ai vari ambienti di elaborazione in cui viene adottato.

**Parte 2 - Gestione dei processi** La [Parte 2](#) discute di come possano convivere più processi e thread in un calcolatore, fornendo dettagli su come il sistema operativo esegue i processi in modo da fornire un buon servizio agli utenti, sfruttando con efficienza le risorse disponibili. Il [Capitolo 5](#) descrive come vengono creati i processi e i thread, le loro interazioni per raggiungere un particolare obiettivo e come sono controllati dal sistema operativo. I restanti cinque capitoli trattano argomenti specifici della gestione dei processi, quali la sincronizzazione e lo scheduling dei processi, il deadlock, la comunicazione tra processi, la sincronizzazione e lo scheduling nei sistemi operativi multiprocessore.

**Parte 3 - Gestione della memoria** I due capitoli di questa parte sono dedicati all'allocazione e alla condivisione della memoria tra i processi. Il [Capitolo 11](#) tratta i concetti fondamentali della gestione della memoria, dando enfasi al problema della *frammentazione della memoria*, che si verifica quando un'area della memoria è inutilizzabile perché è troppo piccola, e le tecniche per evitarla. Il [Capitolo 12](#) descrive l'implementazione della *memoria virtuale*, che risolve il problema della frammentazione della memoria e supporta anche l'esecuzione di programmi complessi.

**Parte 4 - File system e gestione dell'I/O** Questa parte è composta da tre capitoli. Il [Capitolo 13](#) illustra le tecniche per creare, accedere, condividere e memorizzare in maniera affidabile i file. Il [Capitolo 14](#) descrive i dispositivi di I/O e spiega come vengono implementate in maniera efficiente le operazioni sui file. Il [Capitolo 15](#) tratta le tecniche di sicurezza e protezione che permettono di impedire accessi non autorizzati ai file.

**Parte 5 - Sistemi operativi distribuiti** Un sistema operativo distribuito differisce da uno convenzionale poiché le risorse, i processi e le operazioni di controllo del sistema operativo sono presenti nei vari elaboratori che formano un sistema distribuito. Questa differenza genera una serie di problemi che riguardano le prestazioni, l'affidabilità e la sicurezza delle elaborazioni e dello stesso sistema operativo. Questi argomenti sono trattati nei cinque capitoli che compongono quest'ultima parte del libro.

A conclusione della prefazione un ringraziamento particolare va attribuito a Graziano Pravadelli che ha curato la traduzione della quinta parte del volume. Sono felice di poter inoltre ringraziare Alessio Ferone che con il suo estremo impegno ed entusiasmo ha contribuito notevolmente alla realizzazione della versione attuale del volume; a esso si aggiungono Alessia Albanese e Alessandro Perfetto per la loro attenta opera di revisione.

*Alfredo Petrosino*  
Università degli Studi di Napoli "Parthenope"  
Dipartimento di Scienze Applicate

---

## Ringraziamenti dell'Editore

---

L'Editore ringrazia i revisori che con le loro preziose indicazioni hanno contribuito alla realizzazione dell'edizione italiana di *Sistemi operativi*:

Giorgio Brajnik, *Università degli Studi di Udine*  
Marina Lenisa, *Università degli Studi di Udine*  
Giuseppe Lo Re, *Università degli Studi di Palermo*  
Giuseppe Pappalardo, *Università degli Studi di Catania*  
Vincenzo Piuri, *Università degli Studi di Milano*  
Davide Sangiorgi, *Università degli Studi di Bologna*  
Ivan Scagnetto, *Università degli Studi di Udine*

# Guida alla lettura

**CAPITOLO 5**  
**Processi e thread**

**Obiettivi di apprendimento**

- Definizione di processo
- Processi e programmi
- Implementazione dei processi: stati, contesto, eventi
- Condivisione, comunicazione e sincronizzazione tra processi: i segnali
- Thread
- Processi e thread nei sistemi operativi: Unix, Linux, Solaris, Windows

Il concetto di *processo* ci aiuta a capire come i programmi vengono eseguiti in un sistema operativo. Un processo è un programma in esecuzione che utilizza un insieme di risorse. Abbiamo specificato "un" perché nel sistema operativo possono essere presenti diverse istanze dello stesso programma allo stesso tempo; queste esecuzioni costituiscono differenti processi. Questa situazione si verifica quando gli utenti avviano indipendentemente le esecuzioni dei programmi, ognuno con i propri dati, pure quando viene eseguito un programma codificato con tecniche di programmazione concorrente. Il kernel alloca le risorse ai processi e li schedula per l'utilizzo della CPU. In questo modo, si realizza l'esecuzione uniforme di programmi sequenziali e concorrenti.

Anche un *thread* è un programma in esecuzione ma viene eseguito nell'ambito di un processo, ovvero utilizza il codice, i dati e le risorse di un processo. È possibile che molti thread vengano eseguiti nell'ambito dello stesso processo; in questo caso condividono il codice, i dati e le risorse del processo. Il context switch dei thread genera meno overhead rispetto al context switch dei processi.

In questo capitolo, spiegheremo come il kernel controlla processi e thread, come tiene traccia dei loro *stati* e come utilizza le informazioni di stato per organizzare la loro esecuzione. Discuteremo inoltre di come un programma può creare processi o thread concorrenti e di come questi possono interagire per raggiungere un obiettivo comune.

**5.1 Processi e programmi**

Un programma è un'entità passiva che non effettua nessuna operazione da solo; per effettuare le azioni per cui è stato creato, deve essere eseguito. Un *processo* è un programma in esecuzione. Di fatto esegue le azioni specificate all'interno del programma. Un sistema operativo rende possibile l'esecuzione dei programmi degli utenti schedulando i processi per l'utilizzo della CPU.

Gli **Obiettivi di apprendimento** aiutano lo studente a focalizzare l'attenzione sui concetti fondamentali del capitolo.

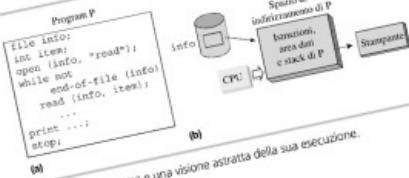


Figura 5.1 Un programma e una visione astratta della sua esecuzione.

### 5.1.1 Cosa è un processo?

Per comprendere cosa è un processo, bisogna prima capire come il SO esegue i programmi. Il programma P mostrato in Figura 5.1(a) contiene le dichiarazioni di un file *info* e di un variabile *item* e le istruzioni per leggere dei valori da *info*, usati per effettuare alcune elaborazioni e stampare un risultato prima di terminare l'esecuzione. Durante l'esecuzione, le istruzioni di questo programma usano i valori contenuti nelle aree dati e nello stack per eseguire le elaborazioni. La Figura 5.2(b) mostra una visione astratta dell'esecuzione del programma P. Le istruzioni, i dati e lo stack del programma costituiscono il suo spazio di indirizzamento. Per realizzare l'esecuzione del programma P, il SO alloca la memoria per lo spazio di indirizzamento di P, alloca una stampante per stampare i risultati, imposta il meccanismo attraverso cui P accede al file *info* e schedula P per l'esecuzione. La CPU è mostrata come un rettangolo leggermente ombreggiato poiché non esegue sempre le istruzioni di P, il SO condivide la CPU tra l'esecuzione di P e l'esecuzione di altri programmi.

In base a quanto detto, è possibile definire un processo come segue:

**Definizione 5.1 Processo** Un programma in esecuzione che utilizza le risorse a esso allocate.

Quando un utente avvia l'esecuzione di un programma, il SO crea un nuovo processo e gli assegna un id univoco. Successivamente alloca alcune risorse al processo, una quantità di memoria sufficiente per contenere lo spazio di indirizzamento del programma e alcuni dispositivi come la tastiera e il monitor, modo da consentire l'interazione con l'utente. Il processo può effettuare delle chiamate di sistema durante l'esecuzione per richiedere ulteriori risorse come per esempio un file. Nel seguito ci riferiamo allo spazio di indirizzamento del programma e allo spazio di indirizzamento a esso allocato, rispettivamente, come spazio di indirizzamento e risorse del processo.

Di ogni concetto viene fornita una **Definizione** compiuta e univoca, che diventa un punto di riferimento durante lo studio del capitolo.



Figura 5.4 Transizioni di stato fondamentali per un processo.

Quando un processo  $P_i$  nello stato *running* effettua una richiesta per un'operazione di I/O, il suo stato deve essere impostato a *blocked* finché l'operazione non viene completata. Al termine dell'operazione di I/O, lo stato di  $P_i$  viene portato da *blocked* a *ready* poiché ora richiede l'uso della CPU. Simili modifiche allo stato si verificano quando un processo effettua qualche richiesta che non può essere soddisfatta immediatamente dal SO. Lo stato del processo viene impostato a *blocked* al momento della richiesta, ovvero quando si verifica l'evento della richiesta ed è portato a *ready* quando la richiesta è stata soddisfatta. Lo stato di un processo *ready* viene modificato a *running* quando si procede al *disabling*. Lo stato di un processo *running* è portato a *ready* quando il processo è *prelazzionato* o se un processo a più alta priorità passa nello stato *ready* o se il processo ha utilizzato tutto il tempo di CPU messogli a disposizione dal SO (vedi Sezioni 3.5.1 e 3.6). La Tabella 5.4 riassume le cause delle transizioni di stato.

La Figura 5.4 mostra le fondamentali transizioni di stato per un processo. Un nuovo processo è impostato nello stato *ready* dopo che le risorse richieste sono state allocata. Poi entra negli stati *running*, *blocked* e *ready* un certo numero di volte come risultato degli eventi descritti nella Tabella 5.4. A un certo punto entra nello stato terminato.

#### Esempio 5.2 Transizioni di stato del processo

Si consideri il sistema time-sharing dell'Esempio 3.2, che utilizza una time slice di 10 ms. Nei sistemi sono attivi due processi  $P_1$  e  $P_2$ .  $P_1$  ha un CPU burst di 15 ms seguito da un'operazione di I/O che dura 100 ms, mentre  $P_2$  ha un CPU burst di 30 ms seguito da un'operazione di I/O che dura 60 ms. L'esecuzione di  $P_1$  e  $P_2$  è stata descritta nella Figura 3.7. La Tabella 5.5 illustra le transizioni di stato durante il funzionamento del sistema. L'esecuzione del programma procede come segue: al tempo 0 entrambi i processi sono nello stato *ready*. Lo scheduler seleziona il processo  $P_1$  per l'esecuzione e imposta il suo stato a *running*. A 10 ms,  $P_1$  viene *prelazzionato* e  $P_2$  è sospeso a *disabling*. Dunque lo stato di  $P_1$  è impostato a *ready* e lo stato di  $P_2$  è impostato a *running*. A 20 ms,  $P_1$  viene *prelazzionato* e  $P_2$  è mandato in esecuzione. A 25 ms,  $P_1$  entra nello stato *blocked* a causa dell'operazione di I/O. Viene effettuato il *dispatch* di  $P_2$  poiché si trova nello stato *ready*. A 35 ms,  $P_1$  viene *prelazzionato* poiché

Per ciascun concetto importante esposto viene proposto un **Esempio** che ne presenta i risvolti più concreti e applicativi.

**Algoritmo 8.2 Algoritmo del banchiere**

**Input**

- $n$  : Numero di processi;
- $r$  : Numero di classi di risorse;
- $Blocked$  : insieme di processi;
- $Running$  : insieme di processi;
- $P_{processo\_richiedente}$  : Processo che effettua la nuova richiesta di risorsa;
- $Risorse\_massime$  : array [1..n, 1..r] di integer;
- $Risorse\_allocate$  : array [1..n, 1..r] di integer;
- $Risorse\_richieste$  : array [1..n, 1..r] di integer;
- $Risorse\_totali\_allocate$  : array [1..r] di integer;
- $Risorse\_totali$  : array [1..r] di integer;

**Strutture dati**

- $Active$  : insieme di processi;
- $fattibile$  : boolean;
- $Nuova\_richiesta$  : array [1..r] di integer;
- $Allocazione\_simulata$  : array [1..n, 1..r] di integer;
- $Risorse\_totali\_allocate\_simulata$  : array [1..r] di integer;

**1.**  $Active = Running \cup Blocked$ ;

**2.**  $\text{for } k = 1..r$  /\* Nuova.richiesta[k] = Risorse.richieste[processo.richiedente,k];  
Allocazione\_simulata[k] = Risorse.allocate[k];  
Allocazione\_simulata[processo.richiedente,k] := Allocazione\_simulata[processo.richiedente,k] + Nuova.richiesta[k];  
Risorse.totali\_allocate\_simulata[k] := Risorse.totali\_allocate[k] + Nuova.richiesta[k];\*/

**3.**  $fattibile = true$ ;  
**for**  $k = 1..r$  /\* Controllare se lo stato di allocazione proiettato è fattibile \*/  
**if**  $Risorse.totali[k] < Risorse.totali_allocate_simulata[k]$  **then** fattibile = false;

**4.** **if** fattibile = true  
**then** /\* Controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro \*/  
**while** l'insieme Active contiene un processo  $P_i$  tale che  
Per ogni  $k$ ,  $Risorse.totali[k] - Risorse.totali\_allocate\_simulata[k] \geq Risorse.massime[i,k] - Allocazione\_simulata[i,k]$   
Rimuovi  $P_i$  da Active;  
**for**  $k = 1..r$   
 $Risorse.totali\_allocate\_simulata[k] := Risorse.totali\_allocate\_simulata[k] - Allocazione\_simulata[i,k];$

**5.** **if** l'insieme Active è vuoto  
**then** /\* Lo stato di allocazione proiettato è uno stato di allocazione sicuro \*/  
**for**  $k = 1..r$  /\* Cancellare la richiesta dalle richieste in attesa \*/  
 $Risorse.richieste[processo.richiedente,k] := 0;$   
**for**  $k = 1..r$  /\* Accettare la richiesta \*/  
 $Risorse.allocate[processo.richiedente,k] := Risorse.allocate[processo.richiedente,k] + Nuova.richiesta[k];$   
 $Risorse.totali_allocate[k] := Risorse.totali_allocate[k] + Nuova.richiesta[k];$

Gli **Algoritmi** più importanti sono messi in evidenza per una facile ed efficace consultazione.

**Pianificazione della capacità**  
L'analisi delle prestazioni può essere utilizzata per la pianificazione della capacità. Per esempio, le formule mostrate in Tabella 7.7 possono essere usate per determinare i valori di importanti parametri come la misura dell'ultimo dei processi ready usato dal kernel.

Come esempio, consideriamo un SO in cui il tasso medio di arrivo di richieste è 5 richieste per secondo, e il tempo medio di risposta per richieste è 3 secondi. La lunghezza media della coda è calcolata dalla formula di Little (Equazione (7.6)), come  $5 \times 3 = 15$ . Notare che le code supereranno questa lunghezza di volta in volta. L'esempio seguente fornisce un criterio di decisione riguardo alla capacità della coda dei processi ready.

**Esempio 7.15 – Pianificazione della capacità con l'uso dell'analisi delle code**  
Un kernel consente fino a  $n$  ingressi nella coda delle richieste ready. Se la coda è piena, quando una nuova richiesta arriva, la richiesta viene rifiutata e lascia il SO. Sia  $p_i$  la probabilità che la coda delle ready contenga i processi in ogni istante, si può mostrare che risulta:

$$p_i = \frac{p^i \times (1 - p)}{1 - p^{i+1}} \quad (7.7)$$

Per  $p = 0.5$  e  $n = 3$ ,  $p_0 = \frac{8}{15}$ ,  $p_1 = \frac{4}{15}$ ,  $p_2 = \frac{2}{15}$ , e  $p_3 = \frac{1}{15}$ . Quindi il 6.7 per cento di richieste viene perso. Si dovrebbe usare un valore maggiore di  $n$  per ridurre il numero di richieste perse.

## Sommario

Lo scheduler di un SO decide quale deve essere il prossimo processo a essere elaborato dalla CPU e per quanto tempo deve essere elaborato. Le sue decisioni influenzano sia il servizio utente che le prestazioni del sistema. In questo capitolo, abbiamo discusso tre tecniche degli scheduler di processo: scheduling basato su priorità, nordino delle richieste e variazione del time slice; abbiamo studiato come gli scheduler le usano per fornire un'auspicata combinazione tra servizio utente e prestazioni di sistema. Inoltre, abbiamo studiato lo scheduling real-time.

Una politica di scheduling nonpreemptive esegue lo scheduling solo quando il processo che la CPU sta elaborando termine; la politica focalizza solamente sul nordino delle richieste per migliorare il tempo di completamento medio dei processi. La politica shortest job first (SJF) ha il problema della starvation, poiché alcuni processi possono essere mandati indefinitamente. La politica highest response ratio next (HRRN) non ha questo problema perché il rapporto di risposta di un processo tende ad aumentare all'aumentare dell'attesa della CPU.

Le politiche di scheduling preemptive prelazionano un processo quando è considerato opportuno effettuare una nuova decisione di scheduling. La politica round-robin (RR) elabora tutti i processi a turno, limitando la quantità del tempo di CPU usato da ciascun processo al valore del time slice. La politica least completed next (LCN) seleziona il processo che ha ricevuto la minima quantità di servizio, mentre la politica shortest time to go (STG) seleziona il processo che è più vicino al completamento.

Al termine di ogni capitolo un **Sommario** riassume i principali argomenti trattati.

**Domande**

- Classificare ognuna delle seguenti affermazioni come vera o falsa:
  - Un ciclo nel grafo di richiesta e allocazione delle risorse (RRAG) è una condizione necessaria e sufficiente per un deadlock se ogni classe di risorsa contiene solo un'unità di risorsa.
  - La risoluzione di deadlock garantisce che i deadlock non si verificheranno successivamente.
  - La politica "tutte le risorse insieme" di prevenzione dei deadlock assicura che la condizione di attesa circolare non si verificherà mai nel sistema.
  - La politica ranking delle risorse di prevenzione dei deadlock assicura che la condizione di hold-and-wait non si verificherà mai nel sistema.
  - Se un insieme di processi  $D$  è in deadlock, l'insieme  $Blocked$  dell'Algoritmo 8.4.1 contenrà alcuni di questi processi quando termina l'esecuzione dell'algoritmo; comunque,  $Blocked$  può non contenere tutti.
  - Se un processo  $P_i$  richiede  $r$  unità di una classe di risorsa  $R_i$  e  $r$  unità di  $R_j$  sono libere, allora l'algoritmo del banchiere allocherà definitivamente  $r$  unità per  $P_i$ .
  - L'algoritmo del banchiere non garantisce che i deadlock non si verificheranno successivamente.
  - Un SO ha una sola classe di risorse controllata dall'algoritmo del banchiere. 12 unità della risorsa sono state attualmente allocate ai processi, di tali risorse 2 sono state allocate per il processo  $P_1$ . Se  $P_1$  ha come massimo di risorse necessarie 5, il sistema è sicuro.
  - Un SO che usa l'algoritmo del banchiere per più risorse è stato in funzione per qualche tempo con quattro processi. Un nuovo processo arriva nel sistema. Inizialmente non ha risorse allocate. Il nuovo stato di allocazione del sistema è sicuro?
  - Se ogni classe di risorsa in un sistema ha una sola unità di risorsa, ogni ciclo nel RRAG del sistema è anche un taglio di risorse.
  - Un SO ha  $n$  unità di risorsa di una classe di risorse. Tre processi usano questa classe di risorse e ognuno di essi ha come massimo di risorse necessarie 3 unità di risorsa. Il modo e l'ordine in cui i processi richiedono le unità della classe di risorse non è noto. Qual è il minimo valore di  $n$  affinché il funzionamento del sistema sia privo di deadlock?
  - a. 3,    b. 7,    c. 6,    d. 9

Soluzioni dei problemi sul sito [www.ateneonline.it/dhamdhere](http://www.ateneonline.it/dhamdhere)**Problemi**

- Giustificare in maniera chiara perché i deadlock possono verificarsi in un sistema produttore-consumatore a buffer limitato (bounded buffer).
- Quando viene utilizzato il ranking delle ri-
- sorse come politica di prevenzione dei deadlock, un processo può richiedere un'unità della classe di risorsa  $R_i$  solo se  $rank_i > rank_j$  per ogni classe di risorsa  $R_j$  le cui risorse sono allocate a esso. Spiega-

Lo studente può verificare la propria preparazione, sia teorica sia applicativa, tramite le **Domande** e i **Problemi** di fine capitolo. Le soluzioni sono disponibili sul sito dedicato al testo [www.ateneonline.it/dhamdhere](http://www.ateneonline.it/dhamdhere).

### Laboratorio 1: comunicazione tra processi

Un sistema di comunicazione tra processi mediante messaggi utilizza la convenzione asymmetric naming che verrà descritta nella Sezione 9.1.1, che fa uso delle seguenti regole per inviare un messaggio, un mittente fornisce l'id del processo destinatario al quale deve essere consegnato. Per ricevere un messaggio, un processo fornisce semplicemente il nome di una variabile in cui il messaggio dovrebbe essere depositato; il sistema restituisce un messaggio inviato al processo richiedente da qualche altro processo.

Il sistema si compone di un monitor chiamato Communication Manager e di quattro processi. Il monitor fornisce le operazioni send e receive, che implementano lo scambio dei messaggi utilizzando un pool globale di 20 buffer di messaggi. Il funzionamento del sistema è il seguente:

1. Ogni processo ha un comportamento ciclico. Il suo funzionamento è governato da comandi di contenuti in un file di comandi utilizzato esclusivamente dal processo. In ogni iterazione, il processo legge un comando dal file e richiama un'appropriata funzione del monitor. Tre comandi sono supportati:
  - a. `send <process_id> <message_text>`: il processo dovrebbe inviare un messaggio.
  - b. `receive <variable_name>`: il processo dovrebbe ricevere un messaggio.
2. Quando un processo richiama l'operazione send, il monitor copia il testo del messaggio in un elemento vuoto del buffer. Se, al momento, il processo destinatario del messaggio è bloccato su un'operazione receive, il messaggio gli viene consegnato come descritto al punto 3 ed il processo viene rinnativato. In ogni caso, il controllo viene restituito al processo che sta eseguendo l'operazione send. Se il buffer è pieno, il processo che sta eseguendo l'operazione send viene bloccato finché non si libera una posizione del buffer.
3. Quando un processo invoca un'operazione receive, i messaggi a esso indirizzati gli vengono recapitati secondo l'ordine FIFO. Il monitor cerca nel buffer il primo messaggio non consegnato al processo, copia il testo del messaggio nella variabile passata dal processo e libera l'elemento del buffer. Se un processo che sta eseguendo l'operazione send risulta bloccato come descritto al punto 2, non viene attivato. Il processo che sta eseguendo l'operazione receive viene bloccato se non ci sono messaggi da consegnargli. Viene rinnativato quando un messaggio gli viene inviato.
4. Dopo aver eseguito un'operazione send o receive, il monitor scrive i dettagli dell'operazione eseguita in file di log.
5. Il monitor rileva una situazione di deadlock, in cui alcuni processi sono bloccati indefinitamente. Scrive i dettagli della situazione di deadlock nel file di log e termina la propria esecuzione.
6. Il sistema di comunicazione tra processi tramite messaggi termina la sua esecuzione quando tutti i processi hanno completato la propria esecuzione.

Implementare il monitor CommunicationManager e testarne il funzionamento con diversi insiem di comandi che creino situazioni interessanti, incluse alcune situazioni di deadlock.

### Laboratorio 2: scheduler del disco

Uno scheduler del disco è quella parte del SO che decide l'ordine in cui le operazioni di I/O devono essere eseguite su un disco per ottenere un alto throughput del disco (vedi Sezione 14.7). I processi che vogliono eseguire un'operazione sul disco utilizzano un monitor chiamato DiskScheduler ed il seguente pseudocodice:

In alcuni capitoli la sezione **Laboratorio** propone allo studente esercitazioni da effettuare al calcolatore.

```
Var Scheduler-disco : Disco_Mon_Type;
Parbegin
begin { Processa utente P_i }
var indirizzo_blocchetto_disco : integer;
repeat
{leggi un comando dal file F_i }
{P_i, Operazione_IO,
Scheduler-disco.Richiesta_IO
indirizzo_blocchetto_disco};
{ Effettua l'Operazione_IO }
Scheduler-disco.IO_completato (P_i);
{ Resto del ciclo }
forever
end;
... { altri processi utente }
Parend;
```

Ogni processo ha un comportamento ciclico, il suo funzionamento è governato dai comandi contenuti in un file dei comandi utilizzato esclusivamente dal processo.

Ogni comando serve per eseguire un'operazione di read o write su un blocco del disco. In ogni iterazione, un processo legge un comando dal suo file dei comandi ed invoca l'operazione del monitor IO-request per passare i dettagli dell'operazione di IO. IO-request blocca il processo finché l'operazione di IO che ha richiesto non viene schedulata. Quando il processo viene attivato, ritorna da IO-request ed esegue la sua operazione di IO. Dopo aver completato l'operazione di IO, invoca l'operazione del monitor IO-complete così che il monitor può schedulare la prossima operazione di IO. Il monitor scrive i dettagli delle sue azioni in un file di log.

Implementare il tipo monitor Disk\_Mon\_Type. Per semplicità, si assuma che le operazioni di IO vengono schedulate in ordine FIFO e che il numero di processi non ecceda 10. (Suggerimento: ammettere l'id di un processo insieme con i dettagli della sua operazione di IO in una lista interna al monitor. Decidere quante variabili di condizione sono necessarie per bloccare ed attivare i processi.)

Modificare Disk\_Mon\_Type in modo tale che le operazioni di IO siano eseguite dal monitor stessa piuttosto che dai processi utente. (Suggerimento: l'operazione IO-complete non sarà più necessaria.)

### Note bibliografiche

Dijkstra (1965) discute il problema della mutua esclusione, descrive l'algoritmo di Dekker e presenta un'algoritmo di mutua esclusione per  $n$  processi. Lamport (1974, 1979) descrive e dimostra l'algoritmo del peniteniero. Ben Ari (1982) descrive l'evoluzione degli algoritmi di mutua esclusione e fornisce una dimostrazione dell'algoritmo di Dekker. Ben Ari (2006) discute la programmazione concorrente e distribuita. Peterson (1981), Lamport (1986, 1991) e Raynal (1986) rappresentano altri fondi sugli algoritmi di mutua esclusione. Dekker (1965) propone altri fondi sugli algoritmi di mutua esclusione. Hoare (1972) e Brinch Hansen (1972) discutono le regioni critiche e le regioni condizionali critiche, che sono costrutti di sincronizzazione che precedono i monitor. Brinch Hansen (1973) e Hoare (1974) descrivono il concetto di monitor. Buhr et al. (1995) descrivono diverse implementazioni di monitor. Richter (1999) de-

Soluzioni dei problemi sul sito [www.olinenonline.it/oliformare](http://olinenonline.it/oliformare)

Le **Note bibliografiche**, che chiudono ogni capitolo, forniscono una bibliografia ragionata allo studente che voglia approfondire ulteriormente gli argomenti trattati.

---

# PARTE 1

## Panoramica

---

Un sistema operativo controlla l'uso delle risorse di sistema di un computer come CPU, memoria, e dispositivi di I/O per soddisfare le richieste di elaborazione dei suoi utenti. Gli utenti si aspettano convenienza, qualità del servizio, e la garanzia che altri utenti non siano in grado di interferire con le proprie attività; gli amministratori di sistema invece si aspettano un utilizzo efficiente delle risorse del computer e buone prestazioni nell'esecuzione dei programmi utente. Queste diverse aspettative possono essere caratterizzate come *convenienza per l'utente, uso efficiente, sicurezza e protezione*; costituiscono l'obiettivo principale di un sistema operativo. Il punto fino al quale un sistema operativo fornisce convenienza per l'utente oppure un utilizzo efficiente dipende dall'*ambiente di elaborazione*, vale a dire dall'hardware di sistema del computer, dalle interfacce con altri computer, e dal tipo di elaborazioni richieste dai suoi utenti.

Per differenti ambienti di elaborazione sono state sviluppate diverse classi di sistemi operativi. Analizzeremo i concetti fondamentali e le tecniche utilizzate in ogni classe di sistemi operativi, nonché la convenienza per l'utente e l'uso efficiente che essi forniscono. Un moderno sistema operativo ha elementi appartenenti a diverse classi di sistemi operativi, dunque molti di questi concetti e di queste tecniche si possono ritrovare allo stesso modo nei moderni sistemi operativi.

Un moderno sistema operativo deve poter essere utilizzato su computer con differenti architetture; inoltre, deve restare al passo con l'evoluzione degli ambienti di elaborazione. Affronteremo le metodologie di progettazione di un sistema operativo in modo che possa essere implementato su differenti architetture, ed evolvere con il suo ambiente di elaborazione.

---

### Linee guida per la Parte 1



Diagramma schematico dell'ordine con cui i capitoli di questa parte dovranno essere affrontati in un corso.

### Capitolo 1 - Introduzione

In questo capitolo si parla di come gli utenti percepiscono la *convenienza per l'utente*, di come il sistema operativo realizza l'*uso efficiente* delle risorse, e di come assicura la *sicurezza* e la *protezione*. Inoltre, il capitolo introduce la nozione di *utilizzo efficace* di un computer come combinazione della convenienza per l'utente e

dell'uso efficiente che meglio si addice a uno specifico ambiente di elaborazione. In questo capitolo si descrivono anche le azioni necessarie alla gestione dei programmi e delle risorse, e la realizzazione degli obiettivi di sicurezza e protezione. L'ultimo paragrafo del capitolo è una panoramica dell'intero libro che descrive i concetti e le tecniche discusse in ogni capitolo e la loro importanza nell'ambito dei sistemi operativi.

## **Capitolo 2 - Il SO, il computer e i programmi utente**

Questo capitolo affronta le caratteristiche hardware di un computer rilevanti per le operazioni e le prestazioni di un sistema operativo (SO). Inoltre, il capitolo descrive come un SO utilizza alcune caratteristiche hardware per controllare l'esecuzione dei programmi utente ed effettuare le operazioni di I/O, e come i programmi utente usano le caratteristiche dell'hardware per interagire con il sistema operativo e ottenere i servizi di cui necessitano.

## **Capitolo 3 - Panoramica dei sistemi operativi**

Questo capitolo si occupa dei principi fondamentali di un sistema operativo; è un capitolo chiave del libro. Affronta la natura dell'elaborazione in diversi ambienti di elaborazione e le caratteristiche dei sistemi operativi utilizzate in questi ambienti, e continua analizzando le nozioni di efficienza, di prestazioni del sistema e del servizio utente. I paragrafi successivi riguardano le cinque classi di sistemi operativi - *sistemi batch, multiprogrammazione, timesharing, real-time* e *sistemi operativi distribuiti* - e descrive i concetti principali e le tecniche che vengono utilizzate per raggiungere i rispettivi obiettivi. Nell'ultimo paragrafo si discute di come un moderno SO applica i concetti e le tecniche utilizzate in questi sistemi operativi.

## **Capitolo 4 - Struttura dei sistemi operativi**

La struttura di un sistema operativo ha due tipi di caratteristiche: quelle che contribuiscono alla semplicità di codifica e alla efficienza dell'esecuzione e quelle che contribuiscono alla facilità con cui un SO può essere implementato su diversi sistemi di elaborazione, o può essere arricchito per incorporare nuove funzionalità. Questo capitolo affronta tre metodi per strutturare un sistema operativo. La *struttura a livelli* di un sistema operativo semplifica la codifica, la *struttura basata su kernel* facilita l'implementazione su diversi sistemi di elaborazione, e la *struttura basata su microkernel* consente la modifica delle caratteristiche di un sistema operativo per adattarlo ai cambiamenti dell'ambiente di elaborazione e, inoltre, facilita l'implementazione su differenti sistemi di elaborazione.

---

# CAPITOLO 1

## Introduzione

---

### Obiettivi di apprendimento

- Principi di un sistema operativo
- Obiettivi di un sistema operativo
- Funzionamento di un sistema operativo
- Panoramica delle funzioni di un sistema operativo

Il modo in cui si definisce un sistema operativo dipende da cosa ci si aspetta dal sistema di elaborazione. Ogni utente pensa in modo differente all'utilizzo del computer. In linguaggio tecnico, diremmo che un utente ha una *visione astratta* di un sistema di elaborazione, una visione che tiene conto solo di quelle caratteristiche che l'utente ritiene importanti.

Il sistema operativo, o SO, come spesso viene chiamato, è l'intermediario *tra* gli utenti e il sistema. Esso fornisce i servizi e le caratteristiche presenti nelle visioni *astratte* di *tutti* i suoi utenti. Esso inoltre consente ai servizi e alle caratteristiche di aggiornarsi nel tempo per adeguare i cambiamenti alle esigenze degli utenti.

Tipicamente, i progettisti di sistemi operativi devono gestire tre problemi: l'uso efficiente delle risorse del computer, la convenienza per gli utenti e la prevenzione delle interreferenze con le attività degli utenti. L'uso efficiente è più importante quando il sistema di elaborazione è dedicato ad applicazioni specifiche, la convenienza per l'utente è più importante nei personal computer, mentre entrambe sono egualmente importanti quando un sistema è condiviso da molti utenti. Dunque il progettista punta a una giusta combinazione, nell'ambiente del sistema operativo, di uso efficiente e convenienza per l'utente. La prevenzione dalle interferenze è obbligatoria in tutti gli ambiti di utilizzo.

Diamo ora uno sguardo d'insieme a cosa fa funzionare un sistema operativo. Vedremo come le sue funzioni di *gestione dei programmi* e *gestione delle risorse* aiutano ad assicurare un uso efficiente delle risorse e la convenienza per l'utente, e come le funzioni di *sicurezza* e *protezione* prevengono l'interferenza con programmi e risorse.

### 1.1 Viste di un sistema operativo

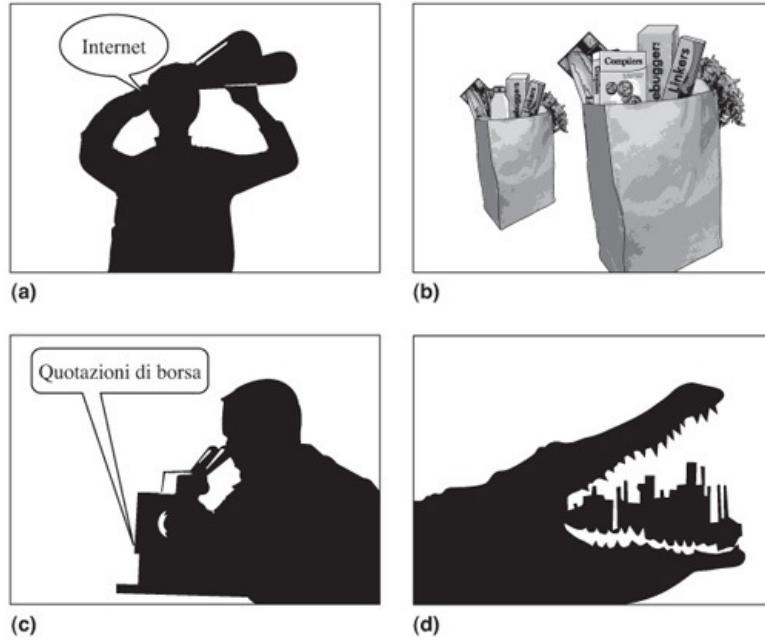
La domanda "Cosa è un SO?" verosimilmente ammette diverse risposte, in base agli interessi dell'utente. Per esempio:

- per uno studente, il SO è il software che permette l'accesso a Internet;
- per un programmatore, il SO è il software che consente di sviluppare programmi;
- per l'utente di un pacchetto di applicazione, il SO è semplicemente il software che permette l'uso di quel pacchetto;
- per un tecnico, per esempio di un impianto chimico computerizzato, il SO è quella componente invisibile di un sistema di elaborazione che controlla l'impianto.

Un utente percepisce un SO semplicemente come un mezzo per ottenere un utilizzo specifico di un sistema di elaborazione. Per uno studente, il solo scopo del computer è di avere accesso a Internet; il SO aiuta a raggiungere questo scopo. Dunque lo studente individua il sistema operativo come un mezzo per navigare in Internet. Il programmatore, l'utente di un pacchetto e il tecnico, in maniera analoga, identificano il SO in base allo scopo per cui utilizzano il computer. Poiché i loro obiettivi sono differenti, anche la loro percezione del SO è differente.

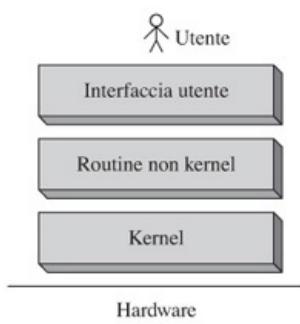
La [Figura 1.1](#) mostra i quattro punti di vista di un SO che abbiamo appena considerato. Essi sono *idee astratte*, perché ognuna si focalizza sulle caratteristiche essenziali dalla prospettiva dell'utente; essa include alcuni elementi reali ma ne ignora altri. Lo studente, l'utente di una applicazione e il tecnico sono utenti finali del SO; le

loro visioni non raggruppano tutte le caratteristiche di un SO. Il punto di vista di un programmatore è quello di uno sviluppatore di software. Esso include le caratteristiche del SO per lo sviluppo del software.



**Figura 1.1** Punti di vista di un SO: dello studente, del programmatore, dell’utente di una applicazione e di un tecnico.

Un progettista di SO ha una sua visione astratta del SO, in termini di struttura e relazioni tra le sue parti costituenti. La [Figura 1.2](#) illustra questa visione.



**Figura 1.2** Visione astratta di un SO dal punto di vista del progettista.

Ogni parte consiste di un numero di routine. Le funzionalità tipiche di queste parti sono le seguenti.

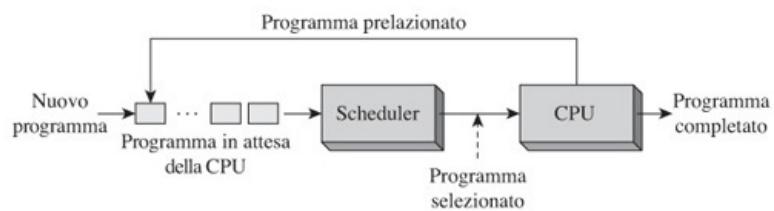
- *Interfaccia utente*: l’interfaccia utente accetta comandi per eseguire programmi e usa le risorse e i servizi forniti dal sistema operativo. Può essere un’interfaccia a linea di comando, come in Unix o Linux, che mostra all’utente un prompt dei comandi e accetta un comando utente, o può essere un’interfaccia grafica (GUI), come nel sistema operativo Windows, che interpreta i click del mouse come comandi utente.
- *Routine non di sistema*: queste routine implementano i comandi utente relativi all’esecuzione dei programmi e all’uso delle risorse del computer; sono richiamate dall’interfaccia utente.
- *Kernel*: il kernel è il cuore del SO. Esso controlla il computer e fornisce un insieme di funzioni e servizi per utilizzare la CPU, la memoria e le altre risorse del computer. Le

funzioni e i servizi del kernel sono richiamati dalle routine non-kernel e dai programmi utente.

Dal punto di vista del progettista di un SO emergono due caratteristiche, mostrate in [Figura 1.2](#). Il sistema operativo è una collezione di routine che facilitano l'esecuzione dei programmi utente e l'uso delle risorse del computer. Esso consiste di un'organizzazione gerarchica a livelli in cui ogni routine di livello superiore utilizza le operazioni fornite dalle routine di livello immediatamente inferiore. Infatti, ogni livello ha una visione astratta del livello sottostante, nel quale il livello successivo è una macchina in grado di interpretare alcuni comandi. Il fatto che un livello inferiore sia un insieme di routine piuttosto che un vero computer non fa alcuna differenza dal punto di vista del livello superiore. Ogni livello superiore agisce come una macchina più sofisticata rispetto al livello a esso inferiore. Dal punto di vista dell'utente, l'interfaccia appare come una macchina che interpreta i comandi nel linguaggio del SO.

Nel prosieguo del libro, verranno utilizzate le visioni astratte per presentare l'organizzazione delle componenti del sistema operativo. Questo si traduce in due benefici.

- *Gestire la complessità*: una visione astratta di un sistema contiene solo caratteristiche selezionate del sistema. Questa proprietà è utile per gestire la complessità durante la progettazione o lo studio di un sistema. Per esempio, una visione astratta di come un SO organizza l'esecuzione dei programmi utente (la [Figura 1.3](#) illustra tale idea successivamente in questo capitolo), si focalizza solo sulla gestione dei programmi; essa semplifica lo studio di un aspetto del SO non mostrando come il SO gestisce altre risorse come la memoria o i dispositivi di I/O.



**Figura 1.3** Uno schema dello scheduling.

- *Presentazione di uno schema generico*: una *astrazione* è utilizzata per presentare un generico schema che nella pratica ha molte varianti. Due esempi sono riscontrabili nella visione astratta del progettista di [Figura 1.2](#). L'interfaccia utente è una astrazione, mentre l'interfaccia a linea di comando e l'interfaccia grafica (GUI) sono due delle sue molte varianti. Il kernel tipicamente presenta un'astrazione del sistema alle routine non-kernel in modo che la diversità dell'hardware, per esempio differenti modelli di CPU e differenti modi di organizzare e accedere ai dati sul disco, risultino totalmente trasparenti alle routine non-kernel.

## 1.2 Obiettivi di un SO

Gli obiettivi fondamentali di un sistema operativo sono:

- *uso efficiente*: assicurare un uso efficiente delle risorse di un computer;
- *convenienza per l'utente*: fornire metodi convenienti per utilizzare il sistema;
- *assenza di interferenze*: prevenire interferenze nelle attività degli utenti.

Un SO deve assicurare un uso efficiente delle risorse e inoltre deve fornire metodi convenienti per utilizzarli. Queste considerazioni alcune volte risultano conflittuali. Per esempio, l'enfasi sui servizi veloci potrebbe significare che le risorse come la memoria devono rimanere allocate a un programma anche quando il programma non è in esecuzione; tuttavia, questo porterebbe a un uso inefficiente delle risorse. Quando si verificano questi conflitti, il progettista deve fare un compromesso per ottenere la combinazione di uso efficiente e convenienza per l'utente che meglio si adatta alla situazione. Questa è la nozione di *uso efficiente* del computer.

È possibile trovare un gran numero di sistemi operativi in uso perché ognuno fornisce

un differente aspetto dell'uso efficiente. Da una parte abbiamo i SO che forniscono servizi veloci richiesti da comandi e applicazioni di controllo, dall'altra abbiamo i SO che usano in modo efficiente le risorse per fornire elaborazione a basso costo, mentre nel mezzo ci sono i SO che forniscono differenti combinazioni dei due. L'interferenza con le attività degli utenti si possono presentare sotto forma di uso illegale o modifica dei programmi e dei dati degli utenti, o in forma di negazione di risorse e servizi per l'utente. Questa interferenza potrebbe essere causata o meno dagli utenti e ogni SO deve prevedere misure atte a prevenirla.

Nel seguito, si discuterà degli aspetti importanti di questi obiettivi fondamentali.

### 1.2.1 Uso efficiente

Un sistema operativo deve assicurare un uso efficiente delle fondamentali risorse di sistema della memoria, della CPU e delle periferiche di I/O come dischi e stampanti. Si può verificare una scarsa efficienza se un programma non utilizza le risorse assegnatigli, per esempio se la memoria o le periferiche di I/O allocate al programma restassero inutilizzate (idle). Una situazione del genere può avere un effetto a cascata: poiché la risorsa è allocata a un programma, essa non può essere assegnata ad altri programmi. Questi programmi non possono andare in esecuzione e dunque le risorse a essi allocate non vengono utilizzate. Inoltre, anche il sistema operativo utilizza risorse di CPU e memoria durante la sua esecuzione, e questo utilizzo di risorse costituisce l'*overhead* che contribuisce a ridurre le risorse disponibili per i programmi utente. Per ottenere buone prestazioni, il SO deve minimizzare lo spreco di risorse da parte dei programmi e il suo stesso overhead. L'uso efficiente delle risorse può essere ottenuto monitorandone l'utilizzo e attuando azioni correttive quando necessario. Comunque, monitorare l'uso delle risorse aumenta l'overhead e quindi abbassa l'efficienza. In pratica, i sistemi operativi che enfatizzano l'uso efficiente limitano l'overhead o restringendo il loro obiettivo all'efficienza di poche importanti risorse, come la CPU e la memoria, oppure non monitorando affatto l'uso delle risorse ma gestendo i programmi utente e le risorse in modo da garantire un'efficienza elevata.

### 1.2.2 Convenienza per l'utente

La convenienza per l'utente ha molti aspetti come mostrato nella [Tabella 1.1](#). All'inizio dei tempi dell'elaborazione con il calcolatore, la convenienza per l'utente era sinonimo di semplice necessità; la semplice possibilità di eseguire un programma scritto in un linguaggio di alto livello era considerato sufficiente. L'esperienza con i primi sistemi operativi ha portato alla richiesta di servizi migliori, che ai giorni nostri significa semplicemente maggiore velocità nel rispondere alle richieste dell'utente.

| Aspetto                          | Esempi  |
|----------------------------------|---|
| Soddisfacimento della necessità  | Capacità di eseguire programmi, utilizzo del file system  |
| Servizio buono                   | Velocità di risposta alle richieste di elaborazione       |
| Interfacce user friendly         | Comandi facili da usare, interfaccia utente grafica (GUI) |
| Nuovi modelli di programmazione  | Programmazione concorrente                                |
| Caratteristiche orientate al Web | Capacità di impostare server Web                          |
| Evoluzione                       | Aggiungere nuove caratteristiche, uso di nuove tecnologie |

**Tabella 1.1** Aspetti della convenienza per l'utente.

Altri aspetti della convenienza per l'utente si sono evoluti con l'utilizzo dei computer in nuovi campi. I primi sistemi operativi avevano *interfacce a linea di comando*, le quali richiedevano che l'utente digitasse un comando e specificasse i suoi parametri. Gli utenti avevano bisogno di un intenso addestramento per apprendere i comandi, il che era accettabile poiché molti utenti erano scienziati o professionisti del computer. Comunque, apparve subito necessario introdurre interfacce più intuitive per facilitare l'uso dei

computer da parte di nuove classi di utenti. In questo modo si svilupparono le *interfacce grafiche* (GUI). Queste interfacce utilizzavano *icone* sullo schermo per rappresentare programmi e file e interpretavano i click del mouse sulle icone e i menu associati come comandi a essi relativi. Sotto molti punti di vista, questo cambiamento può essere paragonato alla diffusione delle abilità di guida nella prima metà del ventesimo secolo. Col passare del tempo, guidare è diventato meno una specialità e più una capacità che poteva essere acquisita con poco addestramento ed esperienza.

Gli utenti affrontavano nuovi problemi con l'incremento della potenza computazionale. Nuovi modelli furono proposti per sviluppare soluzioni più efficienti per nuove classi di problemi. Alcuni di questi modelli poterono essere supportati dalla tecnologia dei compilatori e richiedevano poco supporto da parte del SO; due di questi modelli sono la programmazione modulare e quella orientata agli oggetti. Altri modelli come la programmazione concorrente richiedevano uno specifico supporto implementato nel sistema operativo.

L'avvento di Internet ha motivato l'impostazione di server Web che richiedono supporto di rete e la possibilità di scalare le prestazioni del server in risposta alla quantità di carico a esso diretto.

Gli utenti e le loro organizzazioni investono una considerevole quantità di tempo e sforzi per impostare le loro applicazioni attraverso il sistema operativo. Questo sforzo deve essere preservato quando vengono sviluppate nuove aree di applicazione e nuove tecnologie; dunque è necessario che i sistemi operativi evolvano per fornire nuove caratteristiche e supporto per le nuove aree di applicazione attraverso le nuove tecnologie.

### 1.2.3 Assenza di interferenze

L'utente di un sistema operativo può dover fronteggiare diversi tipi di interferenza nell'utilizzo del computer. L'esecuzione dei suoi programmi o la funzione dei servizi offerti dal sistema operativo possono essere disturbate dalle azioni di altri utenti. Il SO previene queste interferenze allocando risorse a uso esclusivo dei programmi utente e dei servizi del sistema stesso, e prevenendo l'accesso illegale alle risorse. Un'altra forma di interferenza riguarda i programmi e i dati memorizzati nei file degli utenti. Un utente può collaborare con altri utenti nello sviluppo o nell'utilizzo di una applicazione, e dunque può voler condividere alcuni suoi file. I tentativi di accesso a i suoi file eseguiti da ogni altro utente sono illegali e costituiscono un'interferenza alle sue attività. Per prevenire queste forme di interferenza, un SO deve conoscere quale file di un utente può essere utilizzato da altri utenti. Infatti, l'uso di un'autorizzazione consente a un utente di segnalare al sistema operativo i nomi dei suoi collaboratori e i nomi dei file cui possono avere accesso. Il SO utilizza questa informazione per prevenire accessi illegali ai file.

## 1.3 Funzionamento di un SO

Gli obiettivi principali di un SO durante il funzionamento sono l'esecuzione dei programmi, l'utilizzo delle risorse e la prevenzione delle interferenze tra programmi e tra risorse. Di conseguenza, le tre funzioni principali sono:

- *gestione dei programmi*: il sistema operativo inizializza i programmi, organizza l'utilizzo della CPU, e li termina quando hanno completato la loro esecuzione. Poiché molti programmi sono in esecuzione contemporaneamente nel sistema, il SO esegue una funzione chiamata *scheduling* per selezionare il programma da eseguire;
- *gestione delle risorse*: il SO alloca le risorse come la memoria e i dispositivi di I/O quando un programma li richiede. Al termine dell'esecuzione del programma, il sistema operativo dealloca queste risorse e le assegna ad altri programmi che le richiedono;
- *sicurezza e protezione*: il sistema operativo non dovrebbe permettere a nessun utente di utilizzare in modo illegale programmi o risorse nel sistema, o di interferire con il loro funzionamento. Questo obiettivo è perseguito congiuntamente dalle funzioni di sicurezza e protezione. Per esempio si consideri come il SO previene l'accesso illegale a un file. La funzione di *sicurezza* previene l'utilizzo non autorizzato dei servizi e delle risorse del computer. La funzione di *protezione* evita che gli utenti diversi dal proprietario del file o dagli utenti autorizzati da quest'ultimo, abbiano accesso al file.

La [Tabella 1.2](#) descrive le operazioni comunemente effettuate da un sistema operativo.

Quando un computer viene acceso, automaticamente carica un programma memorizzato in una parte riservata di una periferica di I/O, generalmente un disco, e avvia l'esecuzione del programma. Questo programma esegue una tecnica software conosciuta come *bootstrapping* per caricare in memoria il software necessario per la fase di boot, che chiameremo *procedura di boot* - il programma inizialmente caricato in memoria, carica altri programmi in memoria, che a loro volta carichano altri programmi, e così via finché non viene caricata la procedura di boot nella sua totalità. La procedura di boot crea una lista di tutte le risorse hardware nel sistema e passa il controllo del sistema al sistema operativo.

| Task   | Eseguito  |
|--|---|
| Costruire una lista di risorse   | Durante la fase di boot   |
| Memorizzare le informazioni per la sicurezza                           | Mentre vengono registrati nuovi utenti  |
| Verificare l'identità di un utente                                     | Al momento del login  |
| Inizializzare l'esecuzione dei programmi                               | Quando richiesto dall'utente  |
| Memorizzare le informazioni per l'autorizzazione                       | Quando un utente specifica che i suoi collaboratori possono avere accesso a i suoi programmi o ai suoi dati |
| Eseguire l'allocazione delle risorse                                   | Quando richiesto dagli utenti o dai programmi   |
| Preservare lo stato corrente delle risorse                             | Durante l'allocazione e la deallocazione delle risorse  |
| Preservare lo stato corrente dei programmi ed effettuare lo scheduling | In maniera continuativa durante l'esecuzione del SO   |

**Tabella 1.2** Funzioni comuni eseguite dai sistemi operativi.

Un amministratore di sistema specifica quali persone sono registrate come utenti del sistema. Il SO consente solo a queste persone di autenticarsi per utilizzare le sue risorse e i suoi servizi. Un utente autorizza i suoi collaboratori ad accedere ai programmi e ai dati segnalandoli al SO. Il SO annota questa informazione e la usa per implementare la protezione. Il SO esegue anche un insieme di funzioni per implementare la propria nozione di utilizzo effettivo. Queste funzioni includono lo scheduling dei programmi e il tener traccia dell'informazione relativa allo stato e all'uso delle risorse.

I paragrafi successivi presentano una breve panoramica sulle responsabilità del SO nella gestione dei programmi e delle risorse e nell'implementazione della sicurezza e della protezione.

### 1.3.1 Gestione dei programmi

Le moderne CPU hanno la capacità di eseguire le istruzioni di un programma a un tasso molto elevato, e dunque è possibile per un SO alternare l'esecuzione di diversi programmi e continuare a fornire un buon servizio all'utente. La funzione chiave per ottenere l'esecuzione alternata dei programmi è lo *scheduling*, che decide a quale programma deve essere concesso l'utilizzo della CPU in ogni istante. La [Figura 1.3](#) mostra una visione astratta dello scheduling. Lo *scheduler*, implementato come routine del SO, mantiene una lista dei programmi in attesa di essere eseguiti dalla CPU, e ne seleziona uno per l'esecuzione. Nei sistemi operativi che forniscono un servizio equo a tutti i programmi, lo scheduler specifica anche per quanto tempo il programma può utilizzare la CPU. Il SO sottrae la CPU al programma al termine del periodo di tempo specificato, e la concede a un altro programma. Questa azione è chiamata *prelazione* (preemption). Un programma che perde la CPU a causa della prelazione è inserito nuovamente nella lista dei programmi in attesa di essere eseguiti dalla CPU (programmi pronti per l'esecuzione).

La politica di scheduling adottata da un SO può influenzare sia l'uso efficiente della CPU sia i servizi per l'utente. Se un programma viene prelazionato dopo essere stato in esecuzione per un breve periodo di tempo, l'overhead dello scheduling sarebbe elevato a

causa delle frequenti prelazioni. Tuttavia, ogni programma dovrebbe subire solo un piccolo ritardo prima di avere l'opportunità di utilizzare la CPU, il che risulterebbe in un buon servizio per l'utente. Se un programma venisse prelazionato dopo un periodo di tempo più lungo, l'overhead dovuto allo scheduling sarebbe minore, ma i programmi dovrebbero subire ritardi più lunghi, e dunque il servizio per l'utente risulterebbe peggiore.

### 1.3.2 Gestione delle risorse

L'allocazione e la deallocazione delle risorse possono essere effettuate usando una tabella delle risorse. Ogni elemento della tabella contiene il nome e l'indirizzo di una risorsa e il suo stato attuale, che indica se è disponibile o allocata a qualche programma. La [Tabella 1.3](#) rappresenta una tabella per la gestione dei dispositivi di I/O.

| Nome della risorsa | Classe    | Indirizzo | Stato dell'allocazione    |
|--------------------|-----------|-----------|---------------------------|
| stampante1         | Stampante | 101       | Allocata a P <sub>1</sub> |
| stampante2         | Stampante | 102       | Libera                    |
| stampante3         | Stampante | 103       | Libera                    |
| disco1             | Disco     | 201       | Allocata a P <sub>1</sub> |
| disco2             | Disco     | 202       | Allocata a P <sub>2</sub> |
| cdw1               | CD writer | 301       | Libera                    |

**Tabella 1.3** Tabella delle risorse per i dispositivi di I/O.

Essa è creata dalla procedura di boot rilevando la presenza di dispositivi di I/O nel sistema, e viene aggiornata dal sistema operativo in maniera consistente con le allocazioni e le deallocazioni eseguite. Poiché è possibile accedere direttamente a ogni parte di un disco, differenti parti del disco possono essere gestite come dispositivi differenti. In questo modo i dispositivi disco1 e disco2 nella [Tabella 1.3](#) potrebbero essere due parti di uno stesso disco.

Due sono le strategie di allocazione delle risorse più utilizzate. Nell'approccio basato sul *partizionamento delle risorse* il SO decide *a priori* quali risorse dovrebbero essere allocate a ogni programma utente; per esempio, può decidere che a un programma sia allocato un 1 MB di memoria, 1000 blocchi del disco e un monitor. Il SO divide le risorse del sistema in molte *partizioni di risorse*, o semplicemente *partizioni*; ogni partizione include 1 MB di memoria, 1000 blocchi di disco e un monitor. Una partizione è allocata a un programma utente nel momento in cui deve iniziare l'esecuzione. Per facilitare l'allocazione delle risorse, la tabella delle risorse contiene elementi relativi alle partizioni delle risorse piuttosto che alle singole risorse come nella [Tabella 1.3](#).

Il partizionamento delle risorse è semplice da implementare, dunque è soggetto a meno overhead; comunque, pecca in flessibilità. Le risorse sono sprecate se una partizione contiene più risorse di quanto un programma necessiti. Inoltre, il SO non può mandare in esecuzione un programma se le sue richieste eccedono le risorse disponibili nella partizione. Questo vale anche se vi sono risorse libere in un'altra partizione.

Nell'approccio per la gestione delle risorse denominato *pool-based*, il SO alloca le risorse prelevandole da un insieme unico di risorse. Quando un programma richiede una risorsa, il sistema operativo consulta la tabella delle risorse e, se disponibile, la alloca. Questo metodo è soggetto all'overhead dovuto alla allocazione e alla deallocazione su richiesta delle risorse. Comunque, evita entrambi i problemi del metodo del partizionamento delle risorse: una risorsa allocata non è sprecata, e la richiesta di una risorsa può essere soddisfatta se ne esiste una istanza disponibile.

#### Risorse virtuali

Una *risorsa virtuale* è una risorsa fittizia – è una illusione creata dal SO attraverso l'uso di risorse reali. Un SO può usare la stessa risorsa reale per supportare diverse risorse virtuali. In questo modo, può dare l'impressione di avere un numero di risorse maggiore di quelle effettivamente disponibili. Ogni utilizzo di una risorsa virtuale risulta nell'utilizzo di un'appropriata risorsa reale.

L'uso delle risorse virtuali è iniziato con l'utilizzo dei dispositivi virtuali. Per prevenire mutue interferenze tra programmi, sarebbe una buona norma allocare un dispositivo a uso esclusivo di un programma. Comunque, un computer non possiede molte risorse

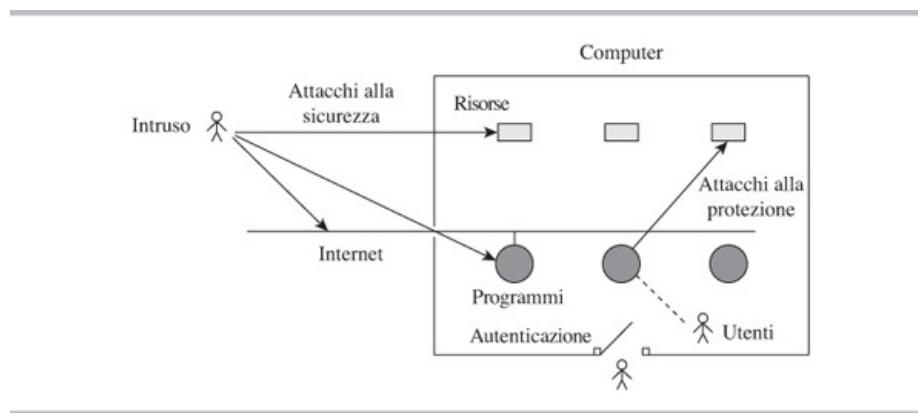
reali, per cui si utilizzano i dispositivi virtuali. Un SO crea una risorsa virtuale quando un utente ha bisogno di un dispositivo di I/O; per esempio, i dischi denominati disco1 e disco2 nella [Tabella 1.3](#) potrebbero essere due dischi virtuali, basati su un unico disco reale, che sono allocati, rispettivamente, ai programmi P<sub>1</sub> e P<sub>2</sub>.

Le risorse virtuali vengono utilizzate anche nei sistemi operativi attuali. Un server di stampa è un tipico esempio di risorsa virtuale. Quando un programma desidera stampare un file, il server di stampa semplicemente copia il file nella coda di stampa. Il programma che ha richiesto la stampa continua l'esecuzione come se la stampa fosse avvenuta. Il server di stampa esamina continuamente la coda di stampa e stampa i file che trova nella coda. Molti sistemi operativi forniscono una risorsa virtuale chiamata *memoria virtuale*, attraverso la quale si ha l'impressione di avere una memoria maggiore di quella fisicamente disponibile nel computer. Il suo utilizzo consente a un programmatore di eseguire un programma la cui dimensione può eccedere la dimensione delle memorie reali.

Alcuni sistemi operativi creano *macchine virtuali* (VM) in modo che ogni macchina possa essere allocata a un utente. Il vantaggio di questo approccio è duplice. L'allocazione di una macchina virtuale a ogni utente elimina le interferenze reciproche tra gli utenti. Inoltre, consente a ogni utente di selezionare un SO di sua scelta che sia in esecuzione sulla sua macchina virtuale. In effetti, questa organizzazione permette agli utenti di usare simultaneamente sistemi operativi differenti sullo stesso computer (Paragrafo 4.5).

### 1.3.3 Sicurezza e protezione

Come accennato nel Paragrafo 1.2.3, un SO deve garantire che nessun utente possa utilizzare in maniera illegale programmi e risorse del sistema, o interferire con il loro funzionamento in alcun modo. La funzione della *sicurezza* riguarda l'uso illegale o le interferenze operate da persone o programmi fuori dal controllo del sistema operativo, mentre la funzione di *protezione* riguarda le stesse situazioni ma eseguite dai suoi utenti. La [Figura 1.4](#) illustra le minacce alla sicurezza e alla protezione di un SO.



**Figura 1.4** Visione d'insieme delle minacce alla sicurezza e alla protezione.

In un classico ambiente stand-alone, un computer opera in completo isolamento. In un tale sistema, le minacce alla sicurezza e alla protezione possono essere gestite facilmente. Si ricordi che un SO conserva le informazioni necessarie per implementare le funzioni di sicurezza e protezione ([Tabella 1.2](#)). L'identità di un utente che vuole utilizzare il computer viene verificata attraverso una password al momento del login. Questa azione, chiamata *autenticazione*, assicura che nessuna persona, a eccezione degli utenti registrati, possa utilizzare il computer. Di conseguenza, non si verificano problemi di sicurezza nel sistema se la procedura di autenticazione è infallibile. In tale ambito, le forme di interferenza menzionate precedentemente nel Paragrafo 1.2.3 sono tutte relative a problematiche di protezione. Il SO protegge l'esecuzione dei programmi e dei servizi con l'aiuto di caratteristiche hardware come la *protezione della memoria*. Impedisce le interferenze consentendo a un utente di avere accesso a un file solo se è il proprietario oppure è stato autorizzato ad accedervi dal proprietario del file.

Quando un sistema è connesso a Internet, e un utente scarica un programma dalla rete, c'è il pericolo che il programma scaricato possa interferire con altri programmi o

risorse del sistema. Questa è una minaccia alla sicurezza poiché l'interferenza è causata da persone al di fuori del sistema, chiamate *intrusi*, che o hanno scritto il programma scaricato oppure lo hanno modificato, in modo che interferisse con altri programmi. Tali minacce alla sicurezza sono attuate attraverso *cavalli di troia*, ovvero programmi che hanno una funzione conosciuta legittima e una funzione dannosa nascosta, e *virus*, cioè una parte di codice con una funzione dannosa che si lega ad altri programmi nel sistema e si trasmette ad altri sistemi. Un'altra classe di minacce alla sicurezza è costituita da programmi chiamati *worm*, che si replicano grazie a difetti nelle impostazioni di sicurezza dei sistemi operativi. I worm possono replicarsi a tassi molto elevati e causare infezioni diffuse. Il worm Code Red del 2001 si è trasmesso a un quarto di milione di sistemi in 9 ore.

I sistemi operativi fronteggiano le minacce alla sicurezza attraverso una varietà di mezzi, utilizzando sofisticate tecniche di autenticazione, eliminando i problemi relativi alla sicurezza quando vengono scoperti, assicurando che i programmi non possano essere modificati, e utilizzando *firewall* per filtrare il traffico di rete non autorizzato. Ci si aspetta che gli utenti contribuiscano alla sicurezza utilizzando password difficili da indovinare ed essendo cauti nello scaricare programmi da Internet.

## 1.4 Introduzione del libro

Un computer, i servizi che fornisce agli utenti e i loro programmi, e le sue interfacce con altri sistemi costituiscono l'*ambiente di elaborazione*. I sistemi operativi sono progettati per offrire l'*utilizzo efficace* del computer e per garantire l'assenza di interferenze nelle attività dei suoi utenti. Le Parti I-IV di questo libro sono incentrate principalmente sui sistemi operativi per ambienti di elaborazione convenzionali caratterizzati dall'uso di un singolo sistema dotato di un'unica CPU; solo il [Capitolo 10](#) affronta i sistemi operativi per ambienti di elaborazione multiprocessore. I sistemi operativi distribuiti sono discussi nei capitoli della Parte V.

In tutto il libro, verranno usate le visioni astratte per presentare la progettazione e l'implementazione dei sistemi operativi perché, come detto nel Paragrafo 1.1, le visioni astratte aiutano a gestire la complessità e a presentare concetti e idee di applicabilità generale.

### 1.4.1 Introduzione ai sistemi operativi

La prima parte del libro consiste dei Capitoli 1-4, dei quali il capitolo corrente è il [Capitolo 1](#). L'inizio dello studio dei sistemi operativi comincerà nel [Capitolo 2](#) con la discussione di come un sistema operativo interagisce con il computer e con i programmi utente.

#### Eventi e interrupt

Un SO alterna l'esecuzione di diversi programmi utente da parte della CPU. Mentre un programma utente è in esecuzione, alcune condizioni riguardanti la sua attività o riguardanti le attività in altri programmi possono richiedere l'attenzione del SO. Richiedere l'attenzione del SO equivale a effettuare il passaggio del controllo della CPU al sistema operativo. Il sistema operativo utilizza la CPU per eseguire le istruzioni che analizzano l'evento ed effettuano le azioni appropriate. Nell'attesa di un evento, il SO schedula un programma utente da fare eseguire alla CPU. Per questo motivo l'esecuzione del SO è detta *guidata dagli eventi* (*event driven*). Per esempio, al termine di un'operazione di I/O, il SO informa il programma che aveva richiesto l'operazione di I/O e avvia un'altra operazione di I/O sul dispositivo; se un programma richiede una risorsa, il SO alloca la risorsa se disponibile. In ogni caso, esegue lo scheduling per selezionare il prossimo programma da eseguire.

La [Figura 1.5](#) è un'idea astratta, anche chiamata *logical view*, del funzionamento di un sistema operativo.



**Figura 1.5** Un sistema operativo nel suo ambiente di elaborazione.

La fine di un'operazione di I/O o la richiesta di una risorsa da parte di un programma causa la generazione di un *interrupt* nel sistema.

La CPU è progettata per rilevare un interrupt e commutare sé stessa per eseguire il codice del SO. La vista fisica, che è la base per lo studio dei sistemi operativi, è sviluppata nel [Capitolo 2](#).

### **Utilizzo efficace di un computer**

Gli ambienti di elaborazione si sono evoluti in risposta ai miglioramenti delle architetture dei computer e alle nuove richieste degli utenti. Ogni ambiente di elaborazione aveva una differente nozione di uso efficace, quindi il suo SO usava differenti tecniche per realizzarlo. Un moderno ambiente di elaborazione annovera caratteristiche di diversi ambienti di elaborazione, come quelli non interattivi, time-sharing, e ambienti di elaborazioni distribuiti, dunque le tecniche adottate in questi ambienti sono usate allo stesso modo nei moderni SO. Il [Capitolo 3](#) affronta queste tecniche per costruire il substrato necessario allo studio dei sistemi operativi.

### **Portabilità ed estendibilità dei sistemi operativi**

I primi sistemi operativi erano sviluppati per specifici computer, e dunque strettamente integrati con le architetture dei sistemi. I moderni sistemi operativi come Unix e Windows propongono due nuovi requisiti: il sistema operativo deve essere *portabile*, cioè dovrebbe essere possibile installarlo su diverse architetture, ed *estendibile* così che possa adattarsi ai nuovi requisiti dettati dai cambiamenti dell'ambiente di elaborazione. Il [Capitolo 4](#) mostra le tecniche di progettazione dei sistemi operativi per la portabilità e l'estendibilità.

## **1.4.2 Gestire le elaborazioni degli utenti**

I Capitoli 5-10, che costituiscono la Parte II del libro, affrontano i vari aspetti della gestione dei programmi. Il [Capitolo 5](#) getta le basi di questo studio discutendo come il sistema operativo gestisce l'esecuzione dei programmi.

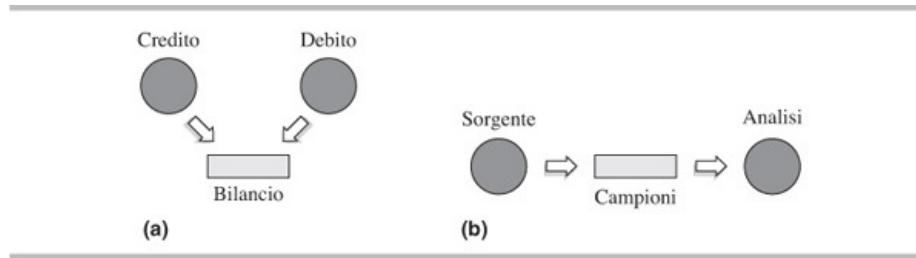
### **Processi e thread**

Un *processo* è un programma in esecuzione. Un SO utilizza un processo come unità di misura del lavoro di elaborazione, cioè gli alloca le risorse e lo schedula per l'utilizzo della CPU. Inoltre, esegue la *commutazione di processo* nel momento in cui decide di prelazionare un processo e schedularne un altro per l'utilizzo della CPU ([Figura 1.3](#)). La commutazione del processo consiste nel salvataggio delle informazioni relative al processo prelazionato e l'inserimento delle informazioni relative al nuovo processo schedulato, il che comporta l'utilizzo della CPU e costituisce l'*over-head* del sistema operativo. La nozione di *thread* viene introdotta per ridurre l'*over-head* di un SO. La commutazione dei thread richiede una minore quantità di informazioni da salvare e a cui bisogna accedere se confrontata con la commutazione dei processi. Comunque, processi e thread sono simili per altri aspetti, e dunque si userà il termine *processo* come termine generico per entrambi, eccetto quando si entra nei dettagli della spiegazione dei thread.

### **Sincronizzazione di processi**

I processi che hanno un obiettivo comune devono coordinare le loro attività in modo tale

da poter effettuare le rispettive azioni nell'ordine desiderato. Questo requisito è chiamato *sincronizzazione di processi*. La [Figura 1.6](#) illustra due tipi di sincronizzazione di processi. La [Figura 1.6\(a\)](#) mostra i processi chiamati *credito* e *debito* che hanno accesso al bilancio di un conto bancario. I loro risultati possono essere non corretti se entrambi i processi aggiornano il bilancio allo stesso tempo, dunque devono effettuare gli aggiornamenti strettamente uno di seguito all'altro. La [Figura 1.6\(b\)](#) mostra un processo chiamato *produttore* che produce alcuni dati e li mette in una variabile chiamata *campioni*, e il processo chiamato *consumatore* che effettua l'analisi sui dati contenuti nella variabile *campioni*. In questo caso il processo *consumatore* non dovrebbe effettuare l'analisi finché il processo *produttore* non ha caricato i nuovi dati in *campioni*, e il processo *produttore* non dovrebbe produrre il prossimo carico di dati finché il processo *consumatore* non ha analizzato i dati precedenti. I linguaggi di programmazione e i sistemi operativi forniscono diverse funzioni che i processi possono utilizzare per implementare la sincronizzazione, mentre le architetture hardware forniscono alcune istruzioni speciali per supportare l'implementazione di queste funzioni. Tutte queste caratteristiche e queste tecniche di sincronizzazione dei processi sono descritte nel [Capitolo 6](#).



**Figura 1.6** Due tipi di sincronizzazione dei processi.

### **Scambio di messaggi**

I processi possono anche interagire attraverso lo *scambio di messaggi*. Quando un processo invia alcune informazioni in un messaggio a un altro processo, il sistema operativo memorizza il messaggio in una struttura interna finché il processo destinatario effettua la richiesta di ricevere un messaggio. Diversamente dalla situazione in [Figura 1.6\(b\)](#), la sincronizzazione tra il processo che invia e quello che riceve il messaggio è effettuata dal sistema operativo, questo fa attendere il processo destinatario se non gli è stato inviato nessun messaggio nel momento in cui effettua la richiesta di ricevere un messaggio. I dettagli relativi allo scambio di messaggi sono descritti nel [Capitolo 9](#).

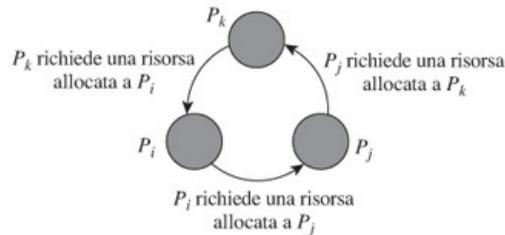
### **Scheduling**

La natura dell'ambiente di elaborazione determina se l'utilizzo efficace di un computer implica l'uso efficiente delle sue risorse, l'elevata convenienza per l'utente, o una opportuna combinazione dei due. Un SO realizza l'utilizzo efficace attraverso un politica di scheduling che rende possibile la condivisione della CPU tra più processi. In questo modo, molti processi procedono allo stesso tempo, il che contribuisce a un servizio veloce per tutti gli utenti, e dunque a un'elevata convenienza per l'utente. Il modo in cui la CPU è condivisa tra i processi determina l'uso delle risorse allocate ai processi, e dunque l'uso efficiente del sistema. Nel [Capitolo 7](#), verranno illustrate le classiche politiche di scheduling, rivolte all'uso efficiente del computer, o all'elevata convenienza per l'utente, e le politiche di scheduling usate nei moderni sistemi operativi, il cui obiettivo è una combinazione ottimale di uso efficiente e convenienza per l'utente.

### **Deadlock**

Nei sistemi operativi i processi utente condividono le risorse. Se una risorsa richiesta da qualche processo  $P_i$  è attualmente allocata al processo  $P_j$ ,  $P_i$  deve attendere finché  $P_j$  non rilascia la risorsa. Queste attese a volte causano uno stallo o *deadlock*, ovvero una situazione in cui i processi attendono le azioni di altri processi per un tempo infinito. La [Figura 1.7](#) mostra questa situazione. La freccia tracciata dal processo  $P_i$  al processo  $P_j$  indica che il processo  $P_i$  è in attesa poiché ha richiesto una risorsa attualmente allocata al processo  $P_j$ . I processi  $P_j$  e  $P_k$  allo stesso modo sono in attesa delle risorse allocate, rispettivamente, ai processi  $P_k$  e  $P_i$ . Dunque i tre processi si trovano in una situazione di

deadlock. Un deadlock compromette le prestazioni di un sistema poiché i processi coinvolti nel deadlock non possono fare nessun progresso e le risorse allocate non sono utilizzate. Le tecniche di gestione dei deadlock usate nei sistemi operativi verranno affrontate nel [Capitolo 8](#).



**Figura 1.7** Un deadlock che coinvolge tre processi.

### Sistemi operativi multiprocessore

Un sistema di elaborazione multiprocessore può fornire elevate prestazioni poiché le sue CPU possono essere utilizzate da più processi simultaneamente. Per realizzare questi vantaggi, il sistema operativo deve utilizzare speciali tecniche di scheduling e sincronizzazione per garantire che i processi possano essere eseguiti dalle CPU in modo efficiente e armonico. Queste tecniche vengono esaminate nel [Capitolo 10](#).

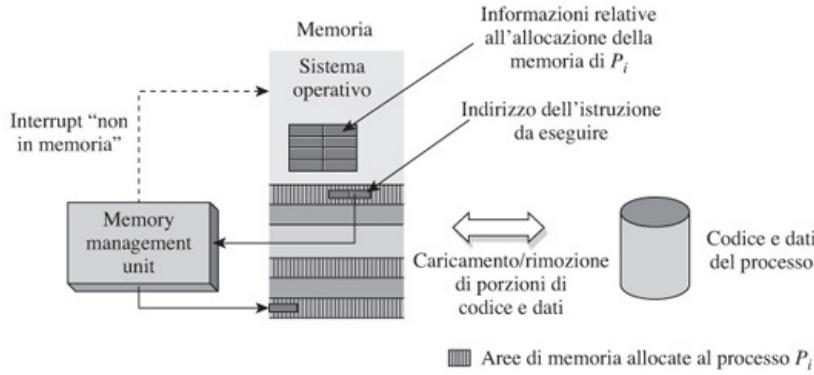
### 1.4.3 Gestione della memoria

La gestione della memoria coinvolge l'allocazione efficiente, il rilascio e il riuso della memoria per assecondare le richieste dei processi. Nel classico modello di allocazione della memoria, viene allocata al processo una singola area di memoria contigua. Questo modello non supporta il riuso di un'area di memoria che non sia abbastanza grande da contenere un nuovo processo, pertanto il kernel deve usare tecniche di *compattamento* per combinare diverse aree libere di memoria in un'unica grande area; questo causa un sostanziale overhead. Il *modello di allocazione di memoria non contiguo* permette di allocare a un processo molte aree di memoria separate, che consentono l'immediato riutilizzo di molte piccole aree. Le tecniche di riutilizzo della memoria e il modello di allocazione della memoria non contiguo verranno discussi nel [Capitolo 11](#). Il kernel usa speciali tecniche per garantire le proprie richieste di memoria in maniera efficiente. Anche queste tecniche sono discusse nel [Capitolo 11](#).

#### Memoria virtuale

I moderni sistemi operativi implementano la *memoria virtuale*, che consiste nella possibilità di memorizzare una quantità di dati maggiore della memoria fisica presente nel sistema. Si ottiene questo risultato memorizzando sul disco il codice e i dati di un processo, e caricandone in memoria solo alcune parti. In questo modo, un processo può essere eseguito anche se la sua dimensione eccede la dimensione della memoria.

Il sistema operativo utilizza il modello di allocazione della memoria non contiguo per implementare la memoria virtuale. A tal fine, mantiene una tabella contenente le informazioni relative alla allocazione della memoria per segnalare quali parti del codice e dei dati di un processo sono presenti in memoria, e a quali indirizzi di memoria. Durante l'esecuzione del processo, la CPU passa l'indirizzo di ogni istruzione o dato utilizzato a una speciale unità hardware chiamata *memory management unit* (MMU). La MMU consulta le informazioni relative all'allocazione della memoria del processo e determina gli indirizzi in memoria dove sono memorizzati il codice e i dati. Se l'istruzione o i dati richiesti non si trovano in memoria, la MMU genera un interrupt "non in memoria". Il sistema operativo carica in memoria la porzione di codice o dati che contiene le istruzioni o i dati richiesti, per il quale potrebbe dover rimuovere altre parti dalla memoria, e ripristina l'esecuzione del processo. La [Figura 1.8](#) illustra tale schema di funzionamento quando un processo  $P_i$  è in esecuzione.

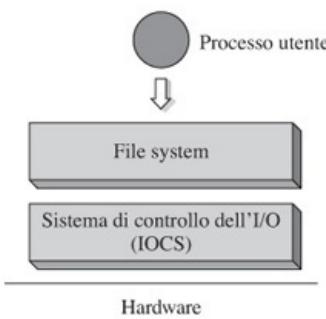


**Figura 1.8** Uno schema del funzionamento della memoria virtuale.

Un interrupt “non in memoria” rallenta il progresso di un processo, pertanto il sistema operativo deve effettuare due decisioni chiave per generare un tasso ridotto di tali interrupt: *quante* e *quali* porzioni di codice e dati di un processo tenere in memoria. Le tecniche usate per prendere questa decisione sono descritte nel [Capitolo 12](#).

#### 1.4.4 Gestione dei file e dei dispositivi di I/O

Un file system deve soddisfare diverse aspettative degli utenti: fornire accesso veloce a un file, proteggere il file dall’accesso di persone non autorizzate, e fornire operazioni affidabili in caso di malfunzionamenti come un I/O difettoso o la mancanza di corrente e, inoltre, assicurare un uso efficiente dei dispositivi di I/O. Un file system utilizza un’organizzazione a livelli per separare le varie fasi coinvolte e soddisfare queste aspettative; la [Figura 1.9](#) mostra una vista astratta. Il livello superiore, che rappresenta lo stesso file system, permette all’utente di condividere i suoi file con altri utenti, implementa la protezione dei file e fornisce affidabilità. Per implementare un’operazione su file, il file system invoca il livello sottostante, che contiene l’*input output control system* (IOCS). Questo livello assicura a un processo un accesso veloce ai file e un uso efficiente dei dispositivi di I/O.



**Figura 1.9** Una panoramica del file system e dell’input output control system (IOCS).

##### File system

Il file system fornisce a ogni utente una vista logica nella quale l’utente ha una *directory home* in un luogo appropriato della gerarchia del file system. L’utente può creare directory, o cartelle, come vengono chiamate nel sistema operativo Windows, nella sua home directory, e altre directory o cartelle in queste directory, e così via. Un utente può autorizzare alcuni collaboratori ad avere accesso a un file comunicando al sistema operativo i nomi dei collaboratori e il file cui possono avere accesso. Il file system utilizza queste informazioni per implementare la protezione. Per realizzare l’affidabilità, il file system previene possibili danni ai dati contenuti in un file, e ai suoi stessi dati, come per esempio le directory, chiamati *metadati*. Questi danni possono essere causati da malfunzionamenti come, per esempio, dispositivi di I/O difettosi o assenza di corrente.

Tutte queste caratteristiche del file system sono discusse nel [Capitolo 13](#).

### **Input Output Control System (IOCS)**

L'IOCS implementa le operazioni sui file trasferendo i dati tra un processo e un file memorizzato su un dispositivo di I/O. Inoltre garantisce un'implementazione efficiente delle operazioni sui file attraverso tre aspetti: riducendo il tempo richiesto per implementare il trasferimento dei dati tra un processo e un dispositivo di I/O, riducendo il numero di volte che i dati devono essere trasferiti tra un processo e un dispositivo di I/O, e massimizzando il numero di operazioni di I/O che un dispositivo può completare per unità di tempo. Le tecniche utilizzate per ottenere questi risultati sono mostrate nel [Capitolo 14](#).

### **Sicurezza e protezione**

Le minacce alla sicurezza e alla protezione, e l'organizzazione utilizzata per implementare la sicurezza e la protezione, sono state affrontate precedentemente nel Paragrafo 1.3.3. Tipicamente si adottano password cifrate mediante una funzione di crittografia conosciuta solo al suo interno. La crittografia rafforza l'organizzazione della sicurezza poiché un intruso non può ottenere le password degli utenti se non attraverso una ricerca esaustiva, che comporterebbe la generazione di ogni possibile stringa come password. Le varie minacce alla sicurezza e alla protezione, le tecniche di crittografia e i vari metodi utilizzati per implementare la protezione sono descritti nel [Capitolo 15](#).

## **1.4.5 Sistemi operativi distribuiti**

Un sistema distribuito è composto da diversi computer, ognuno con la sua memoria, connessi attraverso hardware di rete e software dedicato. Ogni computer è chiamato *nodo*. L'uso dei sistemi distribuiti consente di ottenere tre vantaggi fondamentali: aumento della velocità di esecuzione di una applicazione schedulando simultaneamente i suoi processi su differenti nodi del sistema, elevata affidabilità attraverso la ridondanza dei sistemi e delle risorse, e condivisione delle risorse tra i nodi. Per realizzare questi vantaggi, un SO distribuito deve tenere in conto le seguenti problematiche.

- L'utilizzo della rete causa ritardi nel trasferimento dei dati tra i nodi di un sistema distribuito. Questi ritardi possono condurre a una visione incosistente dei dati posizionati su nodi differenti, e rendere difficile la conoscenza dell'ordine temporale con cui gli eventi si sono susseguiti nel sistema.
- Le funzioni di controllo come lo scheduling, l'allocazione delle risorse, il rilevamento dei deadlock devono essere eseguiti su nodi diversi per ottenere un incremento della velocità e devono fornire un'esecuzione affidabile anche quando risultano difettosi i computer o le componenti di rete.
- Lo scambio di messaggi tra processi in esecuzione su nodi differenti può verificarsi attraverso reti pubbliche e sistemi non controllati dal sistema operativo distribuito. Un intruso può sfruttare questa caratteristica per manomettere i messaggi o creare falsi messaggi per ingannare la procedura di autenticazione e mascherarsi da utente accreditato nel sistema ([Figura 1.4](#)).

I capitoli della Parte V presentano vari aspetti del sistema operativo distribuito. Il [Capitolo 16](#) mostra il modello di un sistema distribuito in termini di hardware e software di rete, e i paradigmi dell'elaborazione distribuita, che consentono di distribuire l'elaborazione sui diversi nodi. Il [Capitolo 17](#) affronta le problematiche teoriche che nascono dai ritardi di rete e i metodi per evitarli. Il [Capitolo 18](#) discute di come il sistema operativo effettua le sue operazioni di controllo in maniera distribuita. Il [Capitolo 19](#) descrive le tecniche utilizzate per migliorare l'affidabilità e le prestazioni dei sistemi operativi distribuiti, mentre il [Capitolo 20](#) affronta le problematiche di sicurezza nei sistemi distribuiti e le tecniche adottate per risolverli.

## **Riepilogo**

Le richieste di un utente sono determinate dal ruolo svolto dal computer nel supportare le sue necessità. Per alcuni utenti, l'utilizzo di un computer consiste semplicemente nella necessità di navigare in Internet o mandare e-mail, mentre per

alcuni altri significa eseguire programmi per elaborare dati o eseguire elaborazioni scientifiche. Un sistema operativo deve soddisfare i bisogni di *tutti* i suoi utenti, pertanto deve fornire diverse funzionalità.

Un computer moderno è dotato di molte risorse come memoria e spazio su disco, e inoltre possiede una CPU potente. Per assicurare che gli utenti del computer traggano beneficio da queste risorse, i sistemi operativi gestiscono molti programmi simultaneamente allocando alcune risorse a ogni programma e alternando la loro esecuzione da parte della CPU. Un SO deve soddisfare tre requisiti per garantire efficacia dell'elaborazione:

- uso *efficiente*: assicurare l'uso efficiente delle risorse del computer;
- *convenienza per l'utente*: fornire metodi convenienti per utilizzare il computer;
- *assenza di interferenze*: prevenire interferenze nelle attività dei suoi utenti.

Un sistema operativo soddisfa questi requisiti eseguendo tre funzioni primarie durante la sua esecuzione: gestione dei programmi, gestione delle risorse, e sicurezza e protezione. Un SO è un software complesso che può contenere milioni di righe di codice, dunque si usa l'astrazione per dominare la complessità nello studio della sua progettazione.

L'astrazione aiuta a focalizzare l'attenzione su un aspetto specifico del sistema, sia esso un sistema hardware come un computer, un sistema software come un sistema operativo, o un sistema della vita reale come una rete di trasporto urbano, e ignorando i dettagli che non sono rilevanti per questo aspetto. L'astrazione verrà utilizzata in tutto il libro per studiare i differenti aspetti della progettazione e del funzionamento dei sistemi operativi.

Il libro è organizzato come segue: si comincia discutendo di come un sistema operativo interagisce con un computer per controllarne il funzionamento. Successivamente, si passa alla studio di come il sistema operativo gestisce l'esecuzione dei programmi, l'allocazione della memoria e l'utilizzo dei file da parte dei programmi e di come garantisce la sicurezza e la protezione. Questo discorso è seguito dallo studio dei sistemi operativi distribuiti, che controllano l'esecuzione di svariati computer collegati tra di loro.

## Domande

- 1.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. La procedura di boot è utilizzata per inizializzare un programma utente.
  - b. La tecnica della prelazione è utilizzata per far condividere la CPU tra i programmi utente.
  - c. Le risorse possono essere sprecate se un SO utilizza l'allocazione delle risorse pool-based.
  - d. L'assegnazione di risorse virtuali a un processo previene le interferenze reciproche tra processi.
  - e. Le minacce messe in atto da un utente autenticato sono minacce alla sicurezza.
- 1.2. Indicare quale delle seguenti tecniche/organizzazioni forniscono (i) convenienza per l'utente e (ii) uso efficiente del sistema.
  - a. Memoria virtuale.
  - b. Protezione dei file.
  - c. Allocazione non contigua della memoria.
- 1.3. Classificare le seguenti affermazioni come attacchi alla sicurezza o protezione.
  - a. Scrivere la propria password su di un pezzo di carta.
  - b. Autorizzare tutti gli utenti ad accedere in lettura e scrittura a un proprio file.
  - c. Lasciare il monitor incustodito durante una sessione.
  - d. Scaricare un programma che si sa contenere un virus.

## Problemi

- 1.1. Un computer consente l'esecuzione di due sistemi operativi, SO<sub>1</sub> e SO<sub>2</sub>. Un

programma  $P$  viene eseguito sempre con successo nel sistema operativo  $SO_1$ .

Quando viene eseguito nel  $SO_2$ , a volte termina l'esecuzione con l'errore "risorse insufficienti per proseguire l'esecuzione", ma altre volte viene eseguito con successo. Qual è la ragione del comportamento del programma  $P$ ? Può essere risolto? In caso affermativo, spiegare come e descriverne le conseguenze. (Suggerimento: pensare alle politiche di gestione delle risorse.)

- 1.2. Un sistema operativo time-sharing usa la seguente politica di scheduling: ad un programma è concessa una limitata quantità di tempo di CPU, detta *time slice*, ogni volta che è selezionato per l'esecuzione. Il programma è prelazionato alla scadenza della time slice, ed è preso in considerazione per andare in esecuzione solo dopo che tutti gli altri programmi in attesa di utilizzare la CPU hanno avuto l'opportunità di utilizzare la CPU. Indicare (a) il servizio per l'utente e (b) l'efficienza nell'utilizzo in un sistema time-sharing.
- 1.3. Se un computer ha una CPU molto veloce ma poca memoria, solo pochi programmi possono risiedere in memoria e di conseguenza la CPU è spesso inutilizzata poiché manca lavoro. Lo *swap* è una tecnica per rimuovere dalla memoria un programma inattivo e caricare al suo posto un programma che richiede l'uso della CPU in modo tale che la CPU lo possa eseguire. Lo swapping migliora (a) il servizio per l'utente e (b) l'efficienza nell'utilizzo? Qual è il suo effetto sull'overhead del SO?
- 1.4. Commentare la validità della seguente affermazione: "L'allocazione partizionata delle risorse consente di ottenere maggiore convenienza per l'utente ma può causare scarsa efficienza".
- 1.5. Si dice che un programma è nello stato inattivo se non è coinvolto in alcuna attività (può essere in attesa di una azione da parte di un utente). Quali risorse consuma un programma inattivo? Come può essere ridotto il consumo di questa risorsa?
- 1.6. Un SO crea delle periferiche virtuali quando ci sono poche periferiche reali. La creazione di periferiche virtuali migliora (a) il servizio per l'utente, (b) l'efficienza nell'uso?
- 1.7. I deadlock si possono verificare nelle seguenti situazioni?
  - a. Un sistema utilizza l'allocazione partizionata delle risorse per un programma.
  - b. Un insieme di programmi comunicano attraverso lo scambio di messaggi durante la loro esecuzione.
- 1.8. Un utente desidera garantire l'accesso ad alcuni suoi file, ma si aspetta che il SO non consenta ai suoi collaboratori di accedere agli altri suoi file, e inoltre non consenta ai non collaboratori di avere accesso ai suoi file. Spiegare come questo obiettivo può essere ottenuto congiuntamente dall'utente e dal SO.

## Note bibliografiche

L'idea di un SO come il software che gestisce un computer è proposta solitamente in molti testi di sistemi operativi. Tanenbaum (2001), Nutt (2004), Silberschatz et al. (2005) e Stallings (2005) sono alcuni dei recenti testi di sistemi operativi.

Berzins et al. (1986) affronta il problema di ridurre la complessità della progettazione di un software di sistema costruendo un insieme di astrazioni che nascondono il funzionamento interno di un sottosistema. Molti libri di ingegneria del software discutono del ruolo dell'astrazione nella progettazione del software. L'articolo di Parnas e Siewiorek (1975) sul concetto di trasparenza nella progettazione del software è considerato un classico dell'ingegneria del software. Il libro di Booch (1994) affronta le astrazioni nello sviluppo del software object oriented.

Il concetto di periferiche virtuali è stato introdotto per la prima volta nel sistema di spooling del computer Atlas sviluppato all'università di Manchester. Esso è descritto in Kilburn et al (1961).

Ludwig (1998) e Ludwig (2002) descrivono diversi tipi di virus, mentre Berghel (2001) descrive il worm Code Red che ha causato molti danni nel 2001. Pfleeger e Pfleeger (2003) è un testo di sicurezza informatica. Garfinkel et al. (2003) affronta la sicurezza in Solaris, Mac OS, Linux e FreeBSD. Russinovich e Solomon (2005) si occupano delle caratteristiche della sicurezza Windows.

1. Berghel, H. (2001): "The Code Red worm," *Communications of the ACM*, **44** (12), 15-19.
2. Berzins, V., M. Gray, and D. Naumann (1986): "Abstraction-based software

- development," *Communications of the ACM*, **29** (5), 403-415.
- 3. Booch, G. (1994): *Object-Oriented Analysis and Design*, Benjamin-Cummings, Santa Clara.
  - 4. Garfinkel, S., G. Spafford, and A. Schwartz (2003): *Practical UNIX and Internet Security*, 3rd ed., O'Reilly, Sebastopol.
  - 5. Kilburn, T., D.J. Howarth, R.B. Payne, and F.H. Sumner (1961): "The Manchester University Atlas Operating System, Part I: Internal Organization," *Computer Journal*, **4** (3), 222-225.
  - 6. Ludwig, M.A. (1998): *The Giant Black Book of Computer Viruses*, 2nd ed., American Eagle, Show Low.
  - 7. Ludwig, M.A. (2002): *The Little Black Book of Email Viruses*, American Eagle, Show Low.
  - 8. Nutt, G. (2004): *Operating Systems-A Modern Perspective*, 3rd ed., Addison-Wesley, Reading, Mass.
  - 9. Parnas, D.L., and D.P. Siewiorek (1975): "Use of the concept of transparency in the design of hierarchically structured systems," *Communications of the ACM*, **18** (7), 401-408.
  - 10. Pfleeger, C.P., and S. Pfleeger (2003): *Security in Computing*, Prentice Hall, Englewood Cliffs, N.J.
  - 11. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
  - 12. Silberschatz, A., P.B. Galvin, and G. Gagne (2005): *Operating System Principles*, 7th ed., John Wiley, New York.
  - 13. Stallings, W. (2005): *Operating Systems-Internals and Design Principles*, 5th ed., Pearson Education, New York.
  - 14. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 2

## Il SO, il computer e i programmi utente

---

### Obiettivi di apprendimento

- Funzionamento di un sistema operativo
- Componenti di un computer: CPU, MMU, gerarchia di memoria, I/O, interrupt
- Esecuzione dei programmi
- Gestione degli interrupt
- Le chiamate di sistema

Come visto nel [Capitolo 1](#), il sistema operativo esegue molti compiti in modo ripetitivo come, per esempio, l'inizializzazione dei programmi e l'allocazione delle risorse. Ognuno di questi compiti viene chiamato *funzione di controllo*. Poiché il sistema operativo è un insieme di routine, e non una unità hardware, esegue le funzioni di controllo utilizzando le istruzioni della CPU. In questo modo, la CPU esegue sia i programmi utente che il sistema operativo. Un aspetto chiave per comprendere come funziona un sistema operativo è capire come interagisce con il sistema di calcolo e i programmi utente, cioè qual è il procedimento mediante il quale prende il controllo della CPU per eseguire una funzione di controllo, e come passa il controllo a un programma utente.

Si utilizza il termine *context switch* per identificare una azione che forza la CPU a sospendere l'esecuzione di un programma e a iniziare l'esecuzione di un altro programma. Quando il kernel ha la necessità di eseguire una funzione di controllo, la CPU passa all'esecuzione del kernel. Dopo aver completato la funzione di controllo, la CPU passa nuovamente all'esecuzione dei programmi utente.

Il presente capitolo inizia con una panoramica delle caratteristiche rilevanti di un computer, e in particolare di come un *interrupt* consente l'esecuzione del kernel da parte della CPU quando ha bisogno di eseguire una funzione di controllo. In un paragrafo successivo si discuterà di come la *gestione degli interrupt* e il concetto di chiamata di sistema (*system call*) possono facilitare l'interazione tra il sistema operativo e i programmi utente.

### 2.1 Principi fondamentali del funzionamento di un SO

Prima di affrontare le caratteristiche dei sistemi operativi nel [Capitolo 3](#), e la loro progettazione nei capitoli successivi, è importante comprendere il funzionamento di un SO, in termini di quali caratteristiche di un computer moderno sono importanti dal punto di vista del SO, come il SO sfrutta queste caratteristiche durante il suo funzionamento per controllare i programmi utente e le risorse, e per implementare sicurezza e protezione, e come i programmi utente ottengono i servizi dal SO.

Come discusso nel Paragrafo 1.1, il *kernel* è l'insieme di routine che costituiscono il cuore del sistema operativo. Esso controlla il funzionamento del computer implementando le operazioni discusse nel Paragrafo 1.3, pertanto ognuna di queste operazioni viene chiamata *funzione di controllo*. Inoltre fornisce servizi all'utente. Il kernel risiede in memoria durante il funzionamento del SO ed esegue le istruzioni utilizzando la CPU per implementare le sue funzioni di controllo e i servizi. In questo modo, la CPU è utilizzata sia dai programmi utente sia dal kernel.

Per un utilizzo efficiente del computer, la CPU dovrebbe eseguire i programmi utente per la maggior parte del tempo. Tuttavia, dovrebbe essere utilizzata per l'esecuzione del codice del kernel qualora nel sistema si verifichi una situazione che richieda l'attenzione del kernel (per esempio quando termina un'operazione di I/O oppure quando viene generato un interrupt, o quando un programma richiede qualche servizio al kernel). Nel Paragrafo 1.4, è stato utilizzato il termine *evento* per tale situazione. Di conseguenza, è necessario comprendere i dettagli del funzionamento del SO in termini di:

- come il kernel controlla il funzionamento del computer;
- come la CPU passa a eseguire il codice del kernel quando si verifica un evento;
- come un programma utente utilizza i servizi messi a disposizione dal kernel;
- come il kernel assicura la mancanza di reciproche interferenze tra programmi utente e tra un programma utente e il sistema operativo.

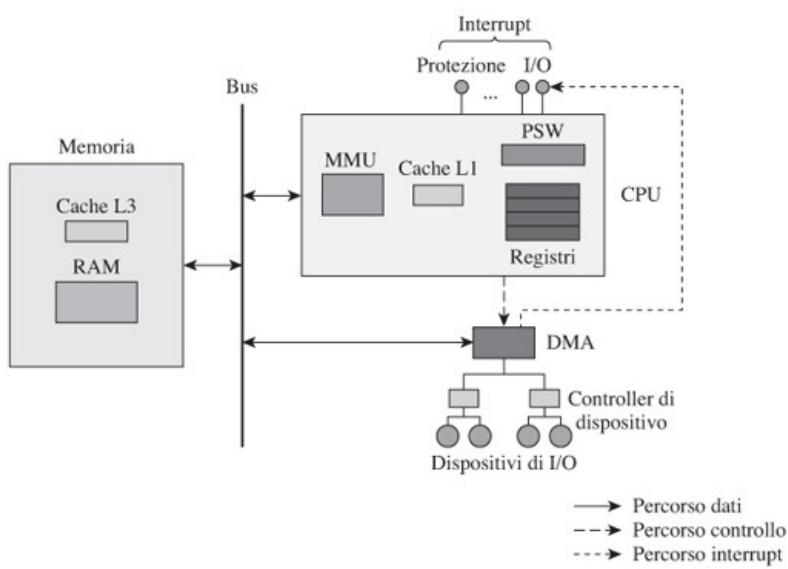
In questo capitolo verranno discussi gli elementi dell'architettura di un computer e verrà descritto come il kernel utilizza le caratteristiche dell'architettura del computer per controllarne il funzionamento. Successivamente verrà discusso come la nozione di *interrupt* sia utilizzata per fare eseguire alla CPU il codice kernel e inoltre verrà descritto come uno speciale tipo di interrupt, chiamato *interrupt software*, viene utilizzato dai programmi per comunicare le loro richieste al kernel.

L'assenza di reciproca interferenza tra i programmi utente e tra questi ultimi e il SO è assicurata dall'uso di due modalità di funzionamento della CPU. Quando la CPU è in *modalità kernel* può eseguire tutte le istruzioni del computer. Il kernel opera con la CPU in questa modalità in modo da controllare le operazioni del computer.

Quando la CPU si trova in *modalità utente* non può eseguire quelle istruzioni che, se usate in modo improprio, possono potenzialmente interferire con altri programmi o con il SO. La CPU viene impostata in questa modalità per eseguire i programmi utente. Un aspetto fondamentale per comprendere il funzionamento di un SO consiste nel conoscere come la CPU viene impostata in modalità kernel per eseguire il codice kernel e come viene impostata in modalità utente per eseguire i programmi utente.

## 2.2 Il computer

La [Figura 2.1](#) mostra lo schema di un computer in cui sono illustrate le unità funzionali rilevanti dal punto di vista di un sistema operativo. La CPU e la memoria sono direttamente connesse al bus, mentre le periferiche di I/O sono connesse al bus attraverso un controller e il DMA. Se la CPU e le periferiche di I/O provano ad accedere alla memoria allo stesso tempo, il bus consente solo a una di loro di procedere. Gli altri accessi sono ritardati finché l'operazione in corso non viene completata. Nei prossimi paragrafi verranno descritti dettagli importanti relativi alle unità funzionali. In un paragrafo successivo, verrà discusso come il SO utilizza le caratteristiche di un computer per controllare il funzionamento del computer e l'esecuzione dei programmi utente. Gli aspetti affrontati in questo capitolo si limitano ai computer con una singola CPU; le caratteristiche dei computer multiprocessore e dei sistemi distribuiti sono descritti nei capitoli successivi.



**Figura 2.1** Schema di un computer.

## 2.2.1 La CPU

### General-Purpose Registers (GPR) e Program Status Word (PSW)

Due caratteristiche della CPU sono visibili ai programmi utente o al sistema operativo. La prima è costituita da quei registri che sono utilizzati per memorizzare i dati, gli indirizzi, gli indici o lo stack pointer durante l'esecuzione dei programmi. Questi registri sono chiamati in modo generico *general-purpose registers* (GPR) o *registri visibili dagli utenti*, tuttavia si preferisce chiamarli GPR. La seconda caratteristica è rappresentata da un insieme di *registri di controllo*, che contengono l'informazione necessaria a controllare il funzionamento della CPU. Per semplicità l'insieme dei registri di controllo verrà chiamato *program status word* (PSW) e i singoli registri di controllo verranno individuati come *campi* della PSW.

La [Figura 2.2](#) descrive i campi della PSW due dei quali sono comunemente conosciuti dai programmati. Il *program counter* (PC) contiene l'indirizzo della prossima istruzione che deve essere eseguita dalla CPU. Il *codice di condizione* (CC) contiene un codice che descrive alcune caratteristiche relative al risultato dell'ultima operazione aritmetica o logica eseguita dalla CPU (per esempio se il risultato di una operazione aritmetica è 0 o il risultato di un confronto è "diverso"). Sebbene queste caratteristiche siano spesso memorizzate in un insieme di *flag*, nel seguito verranno trattate come il campo *codice di condizione* o come un campo chiamato *flag*. Il contenuto e l'uso degli altri registri di controllo verranno descritti di seguito in questo paragrafo.

| Program counter (PC) | Condition code (CC) | Modalità (M) | Informazioni di protezione della memoria (MPI) | Maschera degli interrupt (IM) | Codice interrupt (IC) |
|----------------------|---------------------|--------------|--|-------------------------------|-----------------------|
|----------------------|---------------------|--------------|--|-------------------------------|-----------------------|

| Campo                                    | Descrizione  |
|--|--|
| Program counter                          | Contiene l'indirizzo della prossima istruzione da eseguire.  |
| Condition code (flag)                    | Indica alcune caratteristiche del risultato di un'istruzione aritmetica ( $< 0$ , $= 0$ , $> 0$ ). Questo codice viene utilizzato nell'istruzione di salto condizionato.   |
| Modalità                                 | Indica se la CPU è in esecuzione in modalità kernel o in modalità utente. Si assume un campo di un solo bit con valore 0 per indicare che la CPU è in modalità kernel e 1 per indicare che è in modalità utente. |
| Informazioni di protezione della memoria | Informazioni relative alla protezione della memoria del processo in esecuzione. Questo campo è composto di sottocampi che contengono il registro base e il registro limite.                                      |
| Maschera degli interrupt                 | Indica quali interrupt sono abilitati e quali sono "mascherati".   |
| Codice interrupt                         | Describe la condizione o l'evento che ha causato l'ultimo interrupt. Questo codice è utilizzato da una routine di servizio dell'interrupt.   |

**Figura 2.2** Campi importanti della program status word (PSW).

### Modalità di funzionamento kernel e utente della CPU

La CPU può operare in due modalità chiamate *modalità utente* e *modalità kernel*. La CPU può eseguire determinate istruzioni solo quando è in modalità kernel. Queste istruzioni, chiamate *istruzioni privilegiate*, implementano operazioni cruciali la cui esecuzione da parte dei programmi utente interferirebbe con il funzionamento del SO o con le attività di altri programmi utente. Per esempio, un'istruzione che modifica il contenuto del campo *informazioni di protezione delle memoria* (memory protection information - MPI) del PSW potrebbe essere utilizzata per compromettere la protezione del sistema (il [Paragrafo 2.2.3](#) contiene un esempio).

Un SO impone la CPU in modalità kernel quando esegue le istruzioni del kernel, in modo da poter eseguire le operazioni privilegiate e la impone in modalità utente quando un programma utente è in esecuzione, in modo che il programma non possa interferire con il SO o altri programmi utente. Si assume che il campo *modalità* (M) della PSW sia di un solo bit che contiene uno 0 quando la CPU è in modalità privilegiata e un 1 quando è in modalità utente.

## Stato della CPU

Sia i registri general-purpose sia la PSW contengono tutte le informazioni necessarie per conoscere cosa sta facendo la CPU; diremo che queste informazioni costituiscono lo *stato* della CPU. Come detto nel Paragrafo 1.3.1, il kernel può prelazionare il programma attualmente in esecuzione dalla CPU (Figura 1.3). Per assicurare che il programma possa riprendere la sua esecuzione correttamente, il kernel salva lo stato della CPU quando sottrae la CPU al programma e ricarica lo stato della CPU salvato nei GPR e nella PSW quando deve ripristinare l'esecuzione del programma. L'Esempio 2.1 mostra come il salvataggio e il ripristino dello stato della CPU sia sufficiente per riprendere correttamente l'esecuzione di un programma.

### Esempio 2.1 Stato della CPU

La Figura 2.3(a) mostra un programma in linguaggio assembly di un ipotetico computer la cui CPU è dotata di due registri dati A e B, un registro indice X e il registro per lo stack pointer SP. Ogni istruzione del linguaggio assembly in questo programma corrisponde o a una istruzione della CPU oppure a una direttiva all'assembler (ndt.: compilatore assembly); per esempio l'ultima istruzione dichiara ALPHA come locazione di memoria contenente il valore 1. La prima istruzione sposta il valore di ALPHA nel registro A. La seconda istruzione confronta il valore nel registro A con il valore 1; questo confronto imposta un valore appropriato nel campo *codice di condizione* (anche chiamato campo *flag*). La terza istruzione, il cui codice operazione è BEQ, è un'istruzione di salto condizionato che trasferisce il controllo all'istruzione con la label NEXT se il risultato del confronto è "uguale". Si assuma che il risultato dell'istruzione COMPARE sia "uguale" e che il codice di condizione 00 corrisponda a questo risultato.

| Indirizzo | Istruzione        | PSW | PC   | CCM  |
|-----------|-------------------|-----|------|------|
|           |                   |     | 0150 | 00 1 |
| 0142      | MOVE A, ALPHA     |     |      |      |
| 0146      | COMPARE A, 1      |     |      |      |
| 0150      | BEQ NEXT          |     |      |      |
|           | ...               |     |      |      |
| 0192      | NEXT              |     |      |      |
|           | ...               |     |      |      |
| 0210      | ALPHA DCL_CONST 1 |     |      |      |

(a) (b)

**Figura 2.3** (a) Un programma; (b) stato della CPU dopo l'esecuzione dell'istruzione COMPARE.

Se il kernel sottrae la CPU al programma dopo che questo ha eseguito l'istruzione COMPARE, salva lo stato della CPU come mostrato in Figura 2.3(b). Lo stato consiste del contenuto della PSW e dei registri A, B, X e SP. Il PC contiene 150, ovvero l'indirizzo della prossima istruzione da eseguire. Il campo codice di condizione contiene 00 per indicare che i valori confrontati erano uguali. Il campo MPI contiene le informazioni relative alla protezione della memoria per il programma, che sarà affrontata nel Paragrafo 2.2.3. Se questo stato della CPU viene caricato nuovamente nella CPU, il programma riprenderà la sua esecuzione dall'istruzione BEQ che si trova nella locazione di memoria con indirizzo 150. Poiché il campo codice di condizione contiene 00, corrispondente al valore "uguale", l'istruzione BEQ trasferirà il controllo all'istruzione con label NEXT. In questo modo il programma riprende correttamente l'esecuzione.

## 2.2.2 Memory Management Unit (MMU)

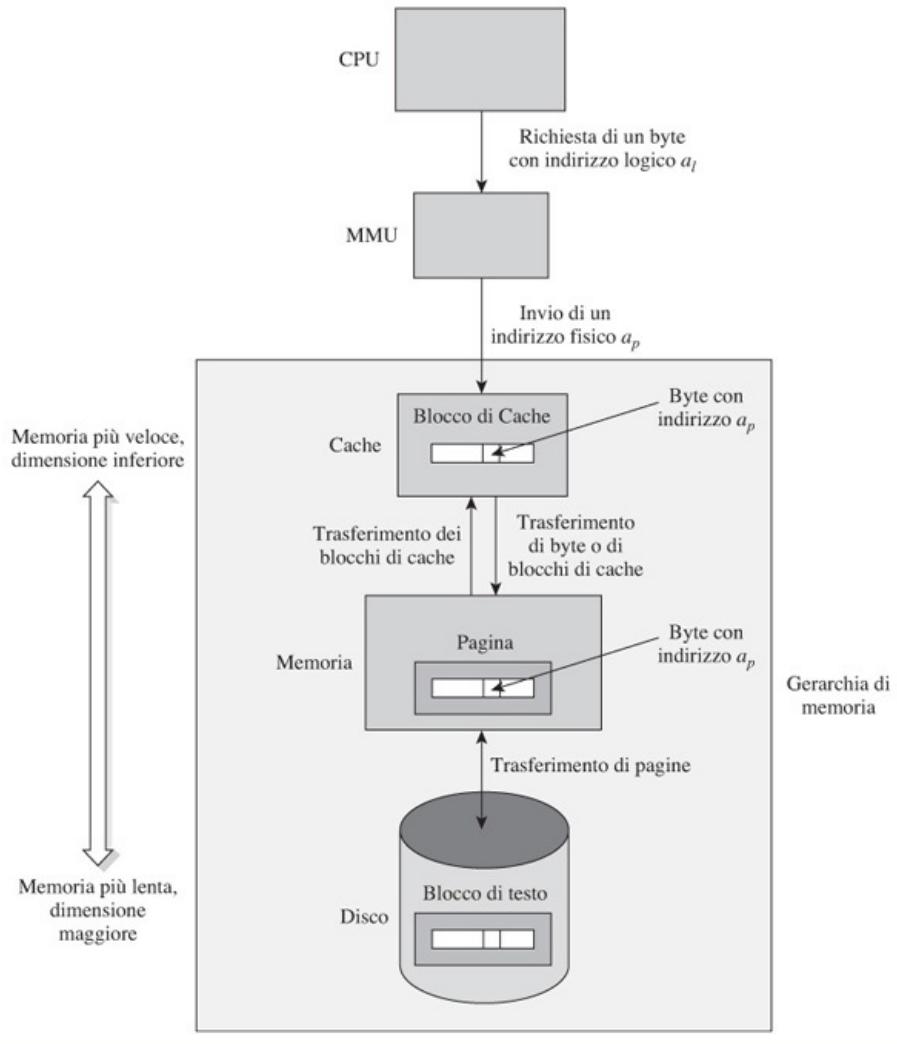
Come accennato nel Paragrafo 1.3.2, la tecnica della *memoria virtuale* consiste nel creare l'illusione di una memoria più grande della reale memoria installata. Come descritto nel Paragrafo 1.4.3, un SO implementa la memoria virtuale utilizzando l'allocazione della memoria non contigua e la MMU (Figura 1.8). Il SO alloca un insieme di aree di memoria a un programma e conserva le informazioni relative a queste aree in una tabella delle informazioni di allocazione della memoria. Durante l'esecuzione di un programma, la CPU passa alla MMU l'indirizzo di un dato o di una istruzione, usato nell'istruzione corrente. Questo indirizzo è chiamato *indirizzo logico*. La MMU utilizza le

informazioni relative all'allocazione della memoria per trovare l'indirizzo in memoria dove attualmente risiede il dato o l'istruzione richiesta. Questo indirizzo è chiamato *indirizzo fisico* e il processo per ottenerlo dall'indirizzo logico è chiamato *traduzione dell'indirizzo*. Per semplicità, non verranno qui descritti i dettagli relativi alla traduzione degli indirizzi, che saranno affrontati nel [Capitolo 12](#).

### 2.2.3 Gerarchia della memoria

Un computer dovrebbe idealmente contenere una memoria abbastanza capiente e abbastanza veloce, così che gli accessi alla memoria non rallentino la CPU. Tuttavia la memoria veloce è costosa e dunque è desiderabile qualcosa che possa fornire lo stesso servizio di una memoria capiente e veloce. La soluzione è una gerarchia della memoria che contenga un numero di unità di memoria con differenti velocità. La memoria più veloce - la più cota per una data quantità - è la più piccola nella gerarchia; le memorie più lente hanno dimensioni più grandi. La CPU accede solo alla memoria più veloce. Se il dato (o l'istruzione) di cui necessita è presente nella memoria più veloce, è usata direttamente; altrimenti il dato richiesto viene copiato dalla memoria più lenta nella memoria più veloce e successivamente utilizzato. Il dato rimane nella memoria più veloce finché non viene rimosso per fare spazio ad altri dati. Questa organizzazione aiuta a velocizzare gli accessi ai dati utilizzati più di frequente. Gli altri livelli nella gerarchia della memoria sono utilizzati in maniera analoga - se un dato non è presente in una memoria più veloce (miss), viene copiato da una memoria più lenta e così via. Il tempo effettivo di accesso alla memoria dipende da quanto frequentemente un miss si verifica in una memoria più veloce.

La [Figura 2.4](#) mostra lo schema di una semplice memoria gerarchica. La gerarchia contiene tre unità di memoria. La memoria cache è veloce e piccola. La memoria principale, anche chiamata *Random Access Memory* (RAM), è lenta e capiente; verrà semplicemente chiamata *memoria*. Il disco è l'unità più lenta e più capiente nella gerarchia. Affronteremo il funzionamento di questa gerarchia di memoria prima di parlare delle gerarchie di memoria dei moderni computer.



**Figura 2.4** Funzionamento di una memoria gerarchica.

### Memoria cache

La memoria cache contiene alcune istruzioni e valori di dati cui la CPU ha avuto accesso più di recente. Per aumentare le prestazioni della cache, l'hardware della memoria non trasferisce un singolo byte dalla memoria nella cache, ma carica sempre un blocco di memoria di dimensioni standard in un'area della cache chiamata *cache block* o *cache line* (linea di cache). In questo modo, l'accesso a un byte vicino a un byte referenziato di recente, può essere effettuato senza accedere alla memoria. Quando la CPU scrive un nuovo valore in un byte, il byte modificato viene scritto nella cache. Prima o dopo dovrà anche essere scritto in memoria. Diversi schemi sono stati utilizzati per scrivere un byte in memoria; un metodo semplice è quello di scrivere il byte nella cache e nella memoria allo stesso tempo. Questo procedimento è chiamato *write-through*.

Per ogni dato o istruzione richiesti durante l'esecuzione di un programma, la CPU effettua una ricerca nella cache confrontando gli indirizzi dei byte richiesti con gli indirizzi dei byte nei blocchi di memoria presenti in cache. Si verifica un *hit* se i byte richiesti sono presenti in memoria, nel qual caso si ha accesso direttamente ai byte; in caso contrario, si verifica un *miss* e i byte devono essere caricati in cache dalla memoria. L'*hit ratio* ( $h$ ) della cache è la frazione di byte cui la CPU ha accesso con un hit in cache. Elevati valori di hit ratio sono ottenuti in pratica come risultato di una legge empirica chiamata *località dei riferimenti* secondo la quale i programmi tendono a referenziare i byte in locazioni vicine ai byte referenziati di recente (*località spaziale*), e tendono a referenziare ciclicamente alcune istruzioni e dati (*località temporale*). Il tempo effettivo di accesso alla memoria (EAT) in una gerarchia di memoria, costituita da una cache e dalla memoria, è dato dalla seguente formula

$$\begin{aligned}
 t_{\text{EAT}} &= h \times t_{\text{cache}} + (1 - h) \times (t_{\text{tra}} + t_{\text{cache}}) \\
 &= t_{\text{cache}} + (1 - h) \times t_{\text{tra}}
 \end{aligned} \tag{2.1}$$

dove  $t_{\text{EAT}}$  = tempo di accesso effettivo alla memoria  
 $t_{\text{cache}}$  = tempo di accesso alla cache  
 $t_{\text{tra}}$  = tempo necessario per trasferire in cache un blocco dalla memoria.

Per garantire un hit ratio maggiore attraverso la località spaziale è necessario prevedere blocchi di cache più grandi. Tuttavia, un blocco di cache grande incrementerebbe il  $t_{\text{tra}}$ . Per questo motivo si utilizzano organizzazioni della memoria avanzate per ridurre il  $t_{\text{tra}}$  e dimensioni del blocco di cache tali da fornire la migliore combinazione di hit ratio e di  $t_{\text{tra}}$ . Il processore Intel Pentium utilizza un blocco di cache di 128 byte e un'organizzazione della memoria che consente di ottenere valori di  $t_{\text{tra}}$  pari a dieci volte il tempo di accesso alla memoria. Se consideriamo  $t_{\text{cache}} = 10$  ns e una memoria 10 volte più lenta della cache, abbiamo  $t_{\text{tra}} = 10 \times (10 \times 10)$  ns = 1000 ns. Con un hit ratio di 0.97, questa organizzazione produce un EAT pari a  $t_{\text{EAT}} = 40$  ns, che rappresenta il 40% del tempo di accesso alla memoria. È da notare che l'hit ratio in una cache è basso all'inizio dell'esecuzione di un programma perché poche istruzioni o dati sono stati trasferiti in cache. L'hit ratio è maggiore quando il programma rimane in esecuzione per lungo tempo.

Le gerarchie di memoria nei moderni computer differiscono da quella mostrata in [Figura 2.4](#) nel numero di memorie cache e nella disposizione della MMU. A causa della grande differenza di velocità tra memoria e cache, per ridurre il tempo di accesso effettivo alla memoria, si utilizza una gerarchia di memorie cache invece della singola cache mostrata in [Figura 2.4](#). Come mostrato in [Figura 2.1](#), una cache L1, ovvero una cache di primo livello, è montata sul chip della CPU. Sullo stesso chip può essere montata anche un'altra cache chiamata cache di livello 2 o L2, più lenta ma più capiente della L1. Una cache L3 ancora più capiente e lenta tipicamente è esterna alla CPU (in [Figura 2.1](#) è mostrata accanto alla memoria centrale). Tutti questi livelli di cache consentono di migliorare il tempo effettivo di accesso alla memoria. Per quantificare il miglioramento, è sufficiente sostituire nell'Equazione 2.1 il tempo di trasferimento di un blocco dal livello di cache più basso al posto di  $t_{\text{tra}}$  e utilizzare l'equazione in maniera analoga in modo da tener conto di un miss nella cache inferiore durante il trasferimento (Problema 2.9). Un'altra differenza è che la MMU viene sostituita da una configurazione parallela della MMU e della cache L1. In questo modo alla cache L1 viene inviato un indirizzo logico piuttosto che un indirizzo fisico. Questa configurazione elimina la necessità di traduzione degli indirizzi prima di effettuare la ricerca nella cache L1, velocizzando l'accesso ai dati nel caso in cui si verifichi un hit nella cache L1. Inoltre, consente di sovrapporre la traduzione dell'indirizzo effettuata dalla MMU con la ricerca nella cache L1, risparmiando tempo nel caso in cui si verifichi un miss nella cache L1.

### **La memoria**

In quanto parte della gerarchia della memoria, il funzionamento della memoria centrale è analogo al funzionamento della cache. Le similitudini riguardano il trasferimento di un blocco di byte, solitamente chiamato *pagina*, dal disco alla memoria quando un programma richiede un byte nel blocco, e il suo trasferimento dalla memoria al disco per fare spazio ad altri blocchi da portare in memoria. La differenza sta nel fatto che la gestione della memoria e il trasferimento dei blocchi tra memoria e disco sono effettuati dal software, diversamente dalla cache, dove sono effettuati dall'hardware. La gerarchia di memoria che comprende la memory management unit (MMU), la memoria e il disco è chiamata *memoria virtuale*. La memoria virtuale è discussa nel [Capitolo 12](#); in altre parti del libro, per semplicità, verrà ignorato il ruolo della MMU e dei dischi.

### **Protezione della memoria**

Molti programmi coesistono nella memoria del computer ed è dunque necessario prevenire che un programma legga o distrugga il contenuto della memoria utilizzata da un altro programma. Questo vincolo viene chiamato *protezione della memoria* ed è implementato controllando se un indirizzo di memoria utilizzato da un programma risiede fuori dall'area a esso allocata.

Per implementare la protezione della memoria vengono utilizzati due registri di

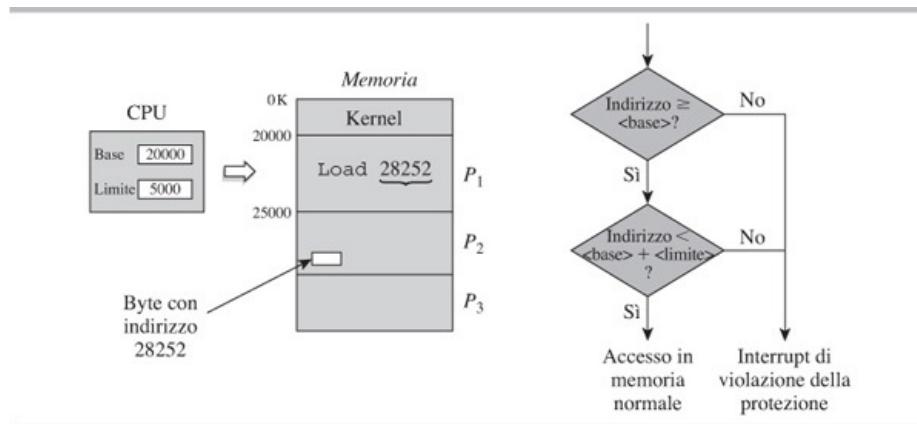
controllo. Il *registro base* contiene l'indirizzo di partenza della memoria allocata a un programma, e il *registro limite* contiene la dimensione della memoria allocata al programma. Di conseguenza, l'indirizzo dell'ultimo byte di memoria allocata a un programma è

$$\text{Indirizzo dell'ultimo byte} = \langle \text{base} \rangle + \langle \text{limite} \rangle - 1$$

dove  $\langle \text{base} \rangle$  e  $\langle \text{limite} \rangle$  indicano, rispettivamente, il contenuto del registro base e del registro limite. Prima di fare ogni riferimento in memoria, per esempio un riferimento alla locazione di memoria con indirizzo  $aaa$ , l'hardware di protezione della memoria verifica se  $aaa$  si trova al di fuori dell'intervallo degli indirizzi definiti dal contenuto dei registri base e limite. In questo caso l'hardware genera un interrupt per segnalare una violazione della protezione e interrompe l'accesso alla memoria. Come descritto in un paragrafo successivo, il kernel termina il programma in risposta all'interrupt. Il campo del PSW relativo all'*informazione di protezione della memoria* (MPI) (Figura 2.2) contiene i registri base e limite. In questo modo l'informazione di protezione della memoria diventa anche una parte dello stato della CPU e viene salvata e caricata quando il programma è prelazionato o riportato in esecuzione.

### Esempio 2.2 Basi della protezione della memoria

Il kernel alloca al programma  $P_1$  l'area di memoria di 5000 byte che va dalla locazione 20 000 alla 24 999. La Figura 2.5 mostra la protezione della memoria per questo programma utilizzando i registri base e limite. L'indirizzo base dell'area allocata (ovvero 20 000) è caricato nel registro base, mentre il numero 5000 viene caricato nel registro limite. Un interrupt di violazione di protezione della memoria verrebbe generato se l'istruzione eseguita dalla CPU utilizzasse un indirizzo al di fuori dell'intervallo 20 000-24 999, per esempio 28 252.



**Figura 2.5** Protezione della memoria utilizzando i registri *base* e *size*.

Un programma potrebbe compromettere lo schema di protezione della memoria caricando informazioni a sua scelta nel registro base e nel registro limite. Per esempio, il programma  $P_1$  potrebbe caricare l'indirizzo 0 nel registro base e la dimensione della memoria del computer nel registro limite e di conseguenza avrebbe la capacità di accedere o modificare qualunque parte della memoria, acquisendo la possibilità di interferire con il SO o altri programmi utente. Per evitare questo problema, le istruzioni per caricare i valori nei registri base e limite sono implementate come istruzioni privilegiate e pertanto quando la CPU si trova in modalità kernel. Poiché la CPU si trova in modalità utente durante l'esecuzione di un programma utente, questa organizzazione impedisce a un programma utente di compromettere lo schema di protezione della memoria.

La protezione della memoria nella cache è più complessa. Va ricordato, dalla discussione precedente, che alla cache L1 si accede utilizzando indirizzi logici. Un programma la cui dimensione è di  $n$  byte utilizza gli indirizzi logici da 0 a  $n - 1$ . In questo modo, molti programmi possono utilizzare gli stessi indirizzi logici e dunque un controllo basato sugli indirizzi logici non può essere utilizzato per decidere se un programma può avere accesso a un valore presente nella cache. Un semplice approccio alla protezione della memoria sarebbe lo svuotamento (*flush*) della cache, ovvero cancellare il contenuto

di tutta la cache nel momento in cui un programma inizia l'esecuzione o viene riportato in esecuzione. In questo modo, la cache non manterebbe il contenuto delle aree di memoria allocate ad altri programmi. Tuttavia ogni parte del programma caricata nella cache l'ultima volta che era in esecuzione, sarebbe ugualmente cancellata. In conseguenza di ciò, inizialmente le prestazioni del programma risulterebbero scarse a causa di un basso hit ratio. In uno schema alternativo, si memorizza l'identificativo del programma le cui istruzioni o dati sono caricati in un blocco della cache e solo a quel programma è consentito l'accesso al contenuto del blocco. Questo schema è implementato come segue: quando un programma genera un indirizzo logico contenuto in un blocco della cache, si verifica un hit solo se l'identificativo del programma coincide con l'identificativo del programma le cui istruzioni o dati sono caricati nel blocco della cache. Tale tecnica è preferita poiché non richiede di svuotare la cache e non influisce sulle prestazioni dei programmi.

## 2.2.4 Input/Output

Un'operazione di I/O richiede l'intervento della CPU, della memoria e di una periferica di I/O. Il modo in cui è implementato il trasferimento tra la memoria e la periferica di I/O determina il tasso di trasferimento e la misura in cui la CPU viene coinvolta nell'operazione di I/O. L'organizzazione dell'I/O utilizzata nei moderni computer si è evoluta attraverso una sequenza di passi mirati a ridurre il coinvolgimento della CPU. Oltre a consentire tassi di trasferimento più elevati, questa organizzazione libera la CPU in modo da consentirgli di effettuare altre attività mentre sono in esecuzione operazioni di I/O.

Assumiamo che gli operandi di una operazione di I/O rappresentino l'indirizzo di una periferica e i dettagli delle operazioni da effettuare. L'esecuzione di un'istruzione di I/O da parte della CPU inizializza l'operazione di I/O sulla periferica indicata. L'operazione di I/O viene eseguita in uno dei tre modi descritti nella [Tabella 2.1](#). Nella *modalità I/O programmato*, il trasferimento dei dati avviene attraverso la CPU. Dunque il trasferimento è lento e la CPU è interamente impegnata nella sua gestione.

| Modalità di I/O                           | Descrizione   |
|---|---|
| I/O programmato                           | Il trasferimento dati tra la periferica di I/O e la memoria avviene attraverso la CPU. La CPU non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O.   |
| Interrupt di I/O                          | La CPU è libera di eseguire altre istruzioni dopo aver eseguito l'istruzione di I/O. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte. Questa sequenza di operazioni viene ripetuta finché tutti i byte sono trasferiti. |
| I/O basato sul direct memory access (DMA) | Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus. La CPU non è coinvolta nel trasferimento dei dati. Il controller DMA genera un interrupt quando il trasferimento di tutti i byte è stato completato.   |

**Tabella 2.1** Modi per Effettuare le Operazioni di I/O.

Di conseguenza può essere effettuata solo un'operazione di I/O alla volta. La *modalità interrupt* è ugualmente lenta poiché effettua il trasferimento dati un byte alla volta con l'aiuto della CPU. Tuttavia la CPU è libera tra i diversi trasferimenti. La *modalità direct memory access (DMA)* può trasferire un blocco di dati tra la memoria e una periferica di I/O senza coinvolgere la CPU, ottenendo tassi di trasferimento elevati e supportando operazioni concorrenti di CPU e dispositivi di I/O. Le modalità interrupt e DMA consentono di eseguire operazioni di I/O simultaneamente su diversi dispositivi.

Le operazioni del DMA sono effettuate dal *controller DMA*, ovvero un processore dedicato all'esecuzione delle operazioni di I/O; tuttavia per semplicità questa distinzione non verrà mantenuta in questo capitolo e ci si riferirà a entrambi semplicemente come DMA. Nella [Figura 2.1](#), l'organizzazione dell'I/O utilizza un DMA. Diversi dispositivi di I/O della stessa classe sono collegati a un controller di periferica e alcuni controller di dispositivo sono collegati al DMA. Quando viene eseguita un'istruzione di I/O, per esempio un'istruzione di *read* sulla periferica *d*, la CPU trasferisce i dettagli

dell'operazione di I/O al DMA. A questo punto la CPU non è coinvolta nell'operazione di I/O ed è libera di eseguire altre istruzioni durante l'esecuzione dell'operazione di I/O. Il DMA passa i dettagli dell'operazione di I/O alla periferica di controllo che avvia l'operazione di *read* sulla periferica *d*. La periferica trasferisce i dati al controller, mentre il trasferimento dei dati tra il controller e la memoria è gestito dal DMA. In questo modo la CPU e il sottosistema di I/O possono operare in maniera concorrente. Al termine del trasferimento dei dati, il DMA genera un *interrupt di I/O*. Come descritto nel paragrafo successivo, la CPU passa all'esecuzione del kernel quando rileva un interrupt. Il kernel analizza la causa dell'interrupt e deduce che l'operazione di I/O è stata completata.

## 2.2.5 Interrupt

Un *evento* è ogni situazione che richiede l'attenzione del sistema operativo. Il progettista del sistema associa un *interrupt* a ogni evento, il cui unico scopo è di segnalare il verificarsi dell'evento al sistema operativo e consentirgli di effettuare le azioni appropriate per la gestione dell'evento. È implementato utilizzando la seguente organizzazione: nel ciclo di esecuzione di un'istruzione della CPU, è recuperata, decodificata ed eseguita l'istruzione il cui indirizzo è contenuto nel program counter (PC). Successivamente controlla se si è verificato un interrupt durante l'esecuzione dell'istruzione. In caso affermativo esegue un'*azione di interrupt* che salva lo stato della CPU, ovvero, il contenuto del PSW e i GPR, e carica i nuovi dati nel PSW e nei GPR in modo che la CPU cominci l'esecuzione, in modalità kernel, delle istruzioni di una *routine di servizio dell'interrupt*, (anche chiamata ISR). A un certo punto, il kernel può ripristinare l'esecuzione del programma interrotto semplicemente ricaricando lo stato salvato della CPU nel PSW e nei GPR (Esempio 2.1). Il progettista del sistema associa una priorità numerica a ogni interrupt. Se diversi interrupt si verificano allo stesso tempo, la CPU seleziona l'interrupt a più alta priorità per servirlo, mentre gli altri interrupt restano pendenti finché non verranno selezionati.

### Classi di interrupt

La [Tabella 2.2](#) descrive tre classi di interrupt che risultano molto importanti durante il normale funzionamento del SO. Un *interrupt di I/O* (o I/O interrupt) indica la fine di un'operazione di I/O o il verificarsi di una condizione di eccezione durante l'operazione di I/O. Un *timer interrupt* è utilizzato per implementare un'organizzazione scandita dal tempo. È utilizzata come segue: un *tick di clock* è definito come una specifica frazione di secondo. Un interrupt può essere generato sia periodicamente, ovvero dopo un predefinito numero di tick, o dopo un intervallo di tempo programmato, cioè dopo l'occorrenza di un numero di tick specificati in uno speciale *registro timer*, che può essere caricato mediante una istruzione privilegiata.

| Classe            | Descrizione   |
|-------------------|---|
| I/O interrupt     | Causato da condizioni come il completamento dell'I/O e il malfunzionamento delle periferiche di I/O.  |
| Timer interrupt   | Generato a intervalli di tempo o quando è trascorso uno specifico intervallo di tempo.  |
| Program interrupt | (1) Causato da condizioni di eccezioni che si verificano durante l'esecuzione di una istruzione, per esempio eccezioni aritmetiche come overflow, eccezioni di indirizzamento e violazioni di protezione della memoria.<br>(2) Causato dall'esecuzione di una speciale istruzione chiamata <i>istruzione di interrupt software</i> , il cui unico scopo è di generare un interrupt. |

**Tabella 2.2** Classi di interrupt.

Un *interrupt da programma* (o programma interrupt), anche chiamato *trap* o *eccezione*, è utilizzato per due scopi. L'hardware del computer utilizza l'interrupt da programma per indicare l'occorrenza di una condizione di eccezione durante l'esecuzione di un'istruzione, per esempio un overflow durante un'operazione aritmetica o una violazione di protezione della memoria (Paragrafo 2.2.3). I programmi utente utilizzano l'interrupt da programma per richiedere al kernel risorse o servizi alle quali non hanno la possibilità di accedere direttamente. I computer forniscono pertanto

un'istruzione speciale il cui unico scopo è di generare un interrupt da programma in modo che il controllo sia passato al kernel. L'operation code di questa istruzione è specifico per ogni macchina, per esempio è *int* nell'Intel Pentium, *trap* nel Motorola 68000 e *syscall* nel MIPS R3000. In generale, si assume che un computer fornisca un'istruzione chiamata *istruzione di interrupt software* con operation code SI e che l'interrupt da questa generata sia un *interrupt software*.

### **Codice interrupt**

Quando si verifica un interrupt di una certa classe, l'hardware imposta un *codice interrupt* nel campo IC del PSW per indicare quale specifico interrupt, all'interno di quella classe, si è verificato. Questa informazione è utile per sapere la causa. Per esempio, se si verifica un interrupt da programma, il codice interrupt aiuterebbe a decidere se è stato causato da una condizione di overflow durante un'operazione aritmetica o da una violazione di protezione della memoria.

I codici interrupt sono specifici per ogni architettura. Per un interrupt di I/O, il codice interrupt è tipicamente l'indirizzo della periferica di I/O che ha causato l'interrupt. Per un interrupt da programma, un computer assegna codici distinti per condizioni di eccezione come overflow e violazione di protezione della memoria, e riserva un insieme di codici per gli interrupt software. Generalmente, l'istruzione per generare un interrupt software (istruzione SI) prende un intero come operando, che viene utilizzato come codice interrupt quando si verifica l'interrupt. Se un computer non fornisce un operando nella funzione SI, un sistema operativo deve modificare la propria organizzazione; per esempio, può richiedere che un programma metta un codice di interrupt sullo stack prima che l'esecuzione dell'istruzione SI causi un interrupt software.

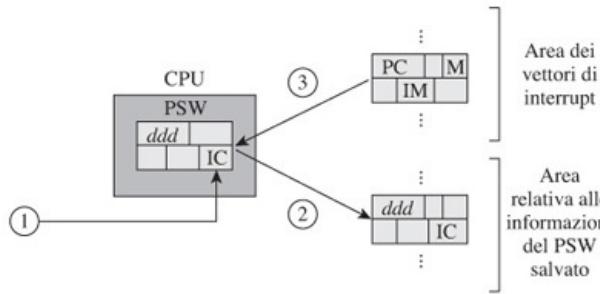
### **Mascheramento degli interrupt**

Il campo *maschera degli interrupt* (IM) del PSW indica quali interrupt sono consentiti in un dato momento. Il campo IM può contenere un intero  $m$  a indicare che verranno rilevati solo gli interrupt con priorità maggiore o uguale a  $m$ . Alternativamente, il campo IM può contenere un valore codificato a bit, in cui ogni bit indica se uno specifico interrupt potrà essere rilevato. Gli interrupt rilevabili sono detti *abilitati* e gli altri sono *mascherati o disabilitati*. Se si verifica un evento corrispondente a un interrupt disabilitato, l'interrupt generato non è perso, ma rimane pendente finché non viene abilitato e può dunque essere rilevato.

### **L'azione di interruzione**

Dopo l'esecuzione di ogni istruzione, la CPU controlla se sono stati generati interrupt. In caso affermativo, la CPU esegue il gestore dell'interrupt (o interrupt handler), che salva lo stato della CPU e passa a eseguire la routine di servizio dell'interrupt nel kernel.

Come mostrato nello schema di [Figura 2.6](#), il gestore di interrupt si compone di tre passi. Il primo passo imposta il codice dell'interrupt nel campo IC del PSW in base alla causa dell'interrupt. Il secondo passo salva il contenuto del PSW in memoria in modo tale che il kernel possa ricreare lo stato della CPU del programma interrotto ([Figura 2.3](#)) nel momento in cui si dovrà ripristinare l'esecuzione del programma. L'area delle *informazioni salvate del PSW*, dove viene conservato il PSW del programma interrotto, è un'area di memoria riservata oppure un'area sullo stack. Il terzo passo esegue l'appropriata routine di servizio dell'interrupt nel kernel come segue: l'area del *vettore degli interrupt* contiene diversi vettori degli interrupt; ogni vettore è utilizzato per controllare la gestione degli interrupt di una determinata classe. In base alla classe di appartenenza di un interrupt, il gestore di interrupt sceglie il vettore degli interrupt corretto e carica il suo contenuto nei campi del PSW. Un vettore degli interrupt contiene le seguenti informazioni:



| Passo                                   | Descrizione   |
|---|---|
| 1. Impostare il codice interrupt        | L'hardware per la gestione dell'interrupt crea un codice che descrive la causa dell'interrupt. Questo codice viene memorizzato nel campo interrupt code (IC) del PSW.   |
| 2. Salvare il PSW                       | Il PSW viene copiato nell'area relativa alle informazioni del PSW salvato. In alcuni computer, questa azione salva anche i registri general purpose.  |
| 3. Caricamento del vettore di interrupt | Si accede al vettore di interrupt corrispondente alla classe di interrupt. L'informazione contenuta nel vettore di interrupt viene caricata nel campo corrispondente del PSW. Questa azione commuta la CPU per l'esecuzione dell'appropriata routine di servizio dell'interrupt del kernel. |

**Figura 2.6** La gestione degli interrupt.

1. l'indirizzo della routine di servizio dell'interrupt;
2. una maschera degli interrupt che indica quali altri interrupt potranno essere gestiti durante l'elaborazione dell'interrupt corrente;
3. uno 0 o un 1 per indicare se la CPU deve trovarsi in modalità kernel o utente durante l'esecuzione della routine di servizio dell'interrupt. Generalmente è scelto lo 0 in modo da consentire alla routine di servizio dell'interrupt, che è una parte del kernel, di utilizzare istruzioni privilegiate.

Per semplicità, si assume che un vettore degli interrupt abbia lo stesso formato di un PSW e contenga questi tre elementi di informazione, rispettivamente, nei campi *program counter* (PC), *maschera degli interrupt* (IM) e *modalità* (M). In questo modo, il terzo passo dalla gestione degli interrupt carica le informazioni dal vettore degli interrupt nei campi *program counter*, *maschera degli interrupt* e *modalità* del PSW, cosa che consente alla CPU di eseguire in modalità kernel la routine di servizio dell'interrupt.

## 2.3 Interazione del SO con il computer e i programmi utente

Per rispondere velocemente agli eventi, un SO utilizza un'organizzazione in cui ogni evento causa un interrupt. In questo paragrafo, si discuterà di come il SO interagisce con il computer per garantire che lo stato del programma interrotto sia salvato, in modo da poter riprenderne l'esecuzione successivamente, e di come una routine di servizio dell'interrupt ottenga le informazioni relative all'evento che ha causato un interrupt, in modo da poter effettuare le azioni appropriate. I programmi hanno bisogno di usare i servizi messi a disposizione dal sistema operativo, come per esempio un'operazione di I/O. Per questo motivo è necessario un metodo con il quale possano generare un interrupt e passare le loro richieste al SO. Una *system call* è un nome generico dato a tali metodi.

### 2.3.1 Controllare l'esecuzione dei programmi

Per controllare l'esecuzione dei programmi utente, il SO deve assicurarsi che quando i programmi utente sono in esecuzione, i vari campi del PSW contengano, in ogni istante, le informazioni appropriate, che includono il momento in cui è iniziata l'esecuzione di un nuovo programma utente e il momento in cui la sua esecuzione è stata ripristinata dopo un'interruzione. Dalla discussione del Paragrafo 2.2, i punti chiave di questa funzione sono:

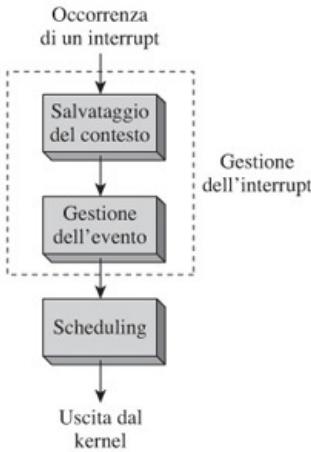
1. all'inizio dell'esecuzione di un programma utente, il PSW dovrebbe contenere le seguenti informazioni:
  - a. il campo *program counter* (PC) dovrebbe contenere l'indirizzo della prima istruzione del programma;
  - b. il campo *modalità* (M) dovrebbe contenere un 1 in modo tale che la CPU operi in modalità utente;
  - c. il campo *informazioni di protezione della memoria* (MPI) dovrebbe contenere informazioni circa l'indirizzo base e la dimensione dell'area di memoria allocata al programma;
  - d. il campo *maschera degli interrupt* (IM) dovrebbe essere impostato in modo da abilitare tutti gli interrupt;
2. quando l'esecuzione di un programma utente viene interrotta, lo stato della CPU - composto dal contenuto del PSW e dei registri general-purpose - dovrebbe essere salvato;
3. quando deve essere ripristinata l'esecuzione di un programma interrotto, lo stato salvato della CPU dovrebbe essere caricato nel PSW e nei registri general-purpose.

Il SO conserva una tabella delle informazioni rilevanti per questa funzione. Per ora, verrà utilizzato il nome generico *tabella dei programmi* e nei capitoli successivi verranno discussi metodi specifici per l'organizzazione di queste informazioni come per esempio il *process control block* (PCB). Ogni elemento della tabella contiene informazioni relative a un programma utente ed è creato quando il programma deve essere avviato. Un campo di questo elemento della tabella è utilizzato per conservare informazioni circa lo stato della CPU. Il kernel scrive le informazioni menzionate nel punto 1 in questo campo quando si deve iniziare l'esecuzione del programma, e salva lo stato della CPU in questo campo quando l'esecuzione del programma viene interrotta - ciò è ottenuto copiando le informazioni dall'area *informazioni salvate del PSW* quando il programma viene interrotto. Le informazioni memorizzate in questo campo sono riutilizzate quando il programma viene mandato nuovamente in esecuzione. Di fatto, i campi rilevanti del PSW conterrebbero le informazioni menzionate nei campi 1(b)-1(d) ogni volta che la CPU sta eseguendo le istruzioni del programma.

### 2.3.2 Servire gli interrupt

Come detto nel Paragrafo 2.2.5, per semplicità, si assume che un vettore degli interrupt abbia lo stesso formato del PSW. Il kernel costruisce il vettore degli interrupt per diverse classi di interrupt all'avvio del sistema operativo. Ogni vettore di interrupt contiene le seguenti informazioni: uno 0 nel campo *modalità* (M) per indicare che la CPU dovrebbe essere utilizzata in modalità kernel, l'indirizzo della prima istruzione della routine di servizio dell'interrupt nel campo *program counter* (PC), uno 0 e la dimensione della memoria nel campo *memory protection information* (MPI) - in modo che la routine di servizio dell'interrupt possa avere accesso all'intera memoria - e una maschera degli interrupt nel campo *maschera degli interrupt* (IM) che o impedisce agli altri interrupt di potersi verificare o consente solo agli interrupt a più alta priorità di potersi verificare, in base alla politica di servizio degli interrupt annidati adottata dal sistema operativo (i dettagli di questa politica verranno discussi successivamente in questo paragrafo).

La [Figura 2.7](#) contiene uno schema del funzionamento del kernel che prende il controllo solo quando si verifica un interrupt; pertanto il suo funzionamento è detto *interrupt-driven* (guidato dagli interrupt). La *routine di servizio dell'interrupt* innanzitutto salva le informazioni riguardanti il programma interrotto nella *tabella dei programmi*, per utilizzarle quando il programma viene schedulato nuovamente. Queste informazioni consistono nel PSW salvato dall'azione di interruzione, nel contenuto dei GPR e nelle informazioni riguardanti la memoria e le risorse utilizzate dal programma. Queste informazioni vengono chiamate *contesto* dell'esecuzione o semplicemente *contesto* di un programma; l'azione che lo salva si chiama azione di *salvataggio del contesto*. La routine di servizio dell'interrupt effettua le azioni necessarie per gestire l'evento che ha causato l'interrupt. Come detto nel Paragrafo 2.2.5, il campo *codice dell'interrupt* del PSW salvato fornisce informazioni utili a questo scopo. La [Tabella 2.3](#) riassume queste azioni, che chiameremo azioni di *gestione dell'evento* del kernel.



**Figura 2.7** Funzionamento del kernel.

| Interrupt                              | Azione di gestione dell'evento  |
|--|---|
| Eccezioni aritmetiche                  | Terminazione del programma.   |
| Violazione di protezione della memoria | Terminazione del programma.   |
| Interrupt software                     | Soddisfare la richiesta del programma se possibile; in caso contrario, annotarlo per un'azione futura.  |
| Fine di un'operazione di I/O           | Trovare quale programma aveva iniziato l'operazione di I/O e annotare che può ora essere preso in considerazione per essere schedulato. Iniziare un'operazione di I/O pendente, se presente, sul dispositivo. |
| Timer interrupt                        | (1) Aggiornare l'orologio di sistema. (2) Effettuare l'azione appropriata nel caso in cui sia trascorso uno specificato intervallo di tempo.  |

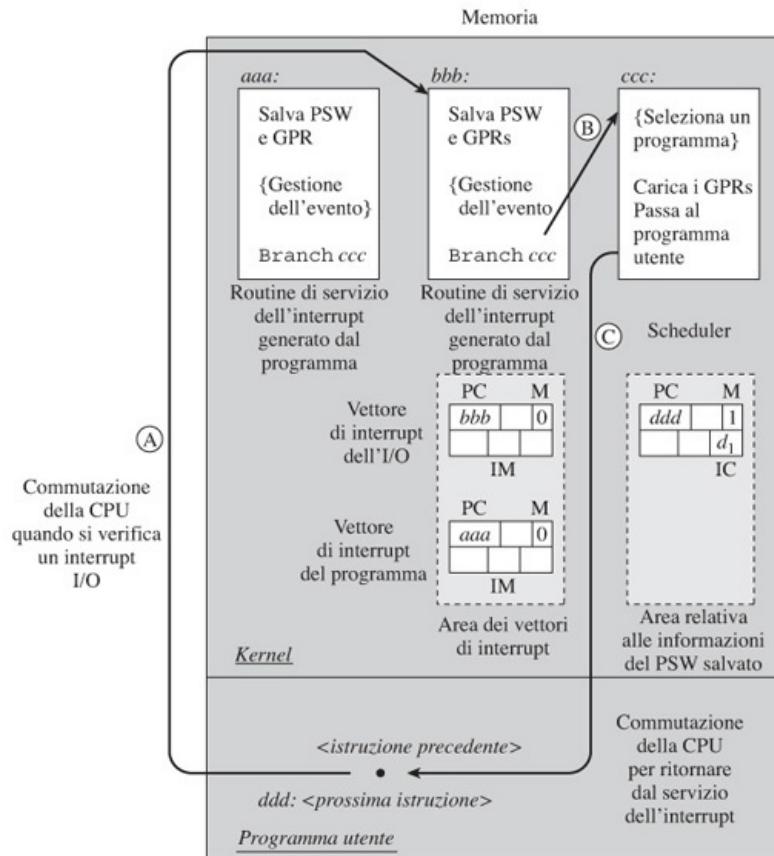
**Tabella 2.3** Azioni di gestione dell'evento del kernel.

La routine di *scheduling* seleziona un programma e lo manda in esecuzione da parte della CPU caricando il PSW salvato e i GPR del programma nei registri della CPU. In base all'evento che ha causato l'interrupt e allo stato degli altri programmi, il programma selezionato per l'esecuzione può essere lo stesso programma che era in esecuzione quando si è verificato un interrupt o qualche altro programma.

L'Esempio 2.3 illustra la gestione dell'interrupt e lo scheduling quando si verifica un interrupt che segnala la fine di una operazione di I/O.

### Esempio 2.3 Gestione dell'interrupt in un ipotetico kernel

La [Figura 2.8\(a\)](#) mostra l'organizzazione del vettore degli interrupt e le routine di servizio degli interrupt in memoria, mentre la [Figura 2.8\(b\)](#) mostra il contenuto del PSW in vari momenti durante la gestione di un interrupt di I/O. I vettori degli interrupt sono creati dalla procedura di boot del SO. Ogni vettore di interrupt contiene l'indirizzo di una routine di servizio degli interrupt, una maschera di interrupt e uno 0 nel campo *modalità*. Un programma utente sta per eseguire l'istruzione che si trova all'indirizzo *ddd* in memoria quando si verifica un interrupt che segnala la conclusione di un'operazione di I/O sulla periferica *d*<sub>1</sub>. La parte più a sinistra della [Figura 2.8\(b\)](#) mostra il contenuto del PSW in quel momento.



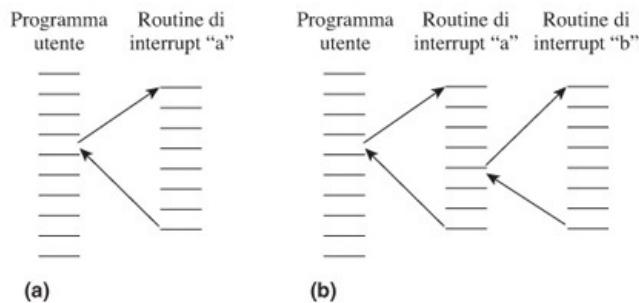
**Figura 2.8** Gestione di un interrupt di I/O e ritorno allo stesso programma utente.

Il primo passo della gestione dell'interrupt memorizza  $d_1$  nel campo IC del PSW e salva il PSW nell'area *informazioni salvate del PSW*. Il PSW salvato contiene un 1 nel campo *modalità*,  $ddd$  nel campo PC e  $d_1$  nel campo IC. Il contenuto del vettore degli interrupt per l'interrupt relativo al completamento di un'operazione di I/O è caricato nel PSW. La CPU è impostata in modalità kernel e il controllo è trasferito alla routine che ha inizio all'indirizzo  $bbb$ , ovvero la routine di servizio dell'interupt di I/O (vedere la freccia segnata ① in [Figura 2.8\(a\)](#) e il contenuto del PSW mostrato in [Figura 2.8\(b\)](#)).

La routine di servizio degli interrupt di I/O salva il PSW e il contenuto dei GPR nella *tabella dei programmi*. Successivamente esamina il campo IC del PSW salvato, riscontra che la periferica  $d_1$  ha completato la sua operazione di I/O e annota che il programma che aveva iniziato l'operazione di I/O può essere preso in considerazione per essere schedulato. Infine trasferisce il controllo allo scheduler (vedere la freccia segnata ② in [Figura 2.8\(a\)](#)). Lo scheduler seleziona per l'esecuzione il programma interrotto e il kernel trasferisce l'uso della CPU al programma ricaricando il contenuto salvato del PSW e dei GPR (vedere la freccia segnata ③ in [Figura 2.8\(a\)](#)). Il programma riprende l'esecuzione all'istruzione con indirizzo  $ddd$  (vedere il contenuto del PSW nella parte più a destra della [Figura 2.8\(b\)](#)).

### Gestione degli interrupt annidati

La [Figura 2.9\(a\)](#) mostra le azioni di gestione dell'interrupt dell'Esempio 2.3 nella forma più semplice: la routine di servizio dell'interrupt "a" gestisce l'interrupt e lo scheduler seleziona per l'esecuzione lo stesso programma interrotto. Se si verifica un altro interrupt mentre la routine di servizio dell'interrupt "a" sta gestendo il primo interrupt, questo porterà ad azioni identiche nell'hardware e nel software. Questa volta, l'esecuzione della routine di servizio dell'interrupt "a" è il "programma" che sarà interrotto; la CPU passerà all'esecuzione di un'altra routine di servizio di interrupt, per esempio, la routine di servizio dell'interrupt "b" ([Figura 2.9\(b\)](#)). Questa situazione ritarda la gestione del primo interrupt e inoltre richiede attenzione nella codifica del kernel per evitare confusione nel caso in cui lo stesso tipo di interrupt fosse generato di nuovo (Problema 2.6). Tuttavia, questo meccanismo consente al kernel di poter rispondere facilmente a interrupt con priorità alta.



**Figura 2.9** Gestione di un interrupt semplice e annidato.

I sistemi operativi hanno utilizzato due approcci per la gestione degli interrupt annidati. Alcuni sistemi operativi usano il campo della maschera degli interrupt (IM) nel vettore degli interrupt per mascherare *tutti* gli interrupt mentre è in esecuzione una routine di gestione dell'interrupt ([Figura 2.8](#)). Questo approccio rende il kernel non prelazionabile, il che semplifica la sua progettazione poiché il kernel sarebbe impegnato nella gestione di un solo interrupt alla volta. Tuttavia nei kernel non preemptive si possono verificare dei ritardi nella gestione degli interrupt a più alta priorità. In un approccio alternativo, il kernel imposta la maschera degli interrupt in ogni vettore degli interrupt per mascherare gli interrupt meno critici, continuando a poter servire gli interrupt più critici in maniera annidata. Tali kernel sono chiamati *kernel prelazionabili o preemptive kernel*. Problemi relativi alla consistenza dei dati si presenterebbero nel caso in cui due o più routine di gestione degli interrupt attivate in maniera annidata aggiornassero gli stessi dati nel kernel, per cui il kernel deve utilizzare uno schema di sincronizzazione per assicurare che solo una routine di servizio dell'interrupt possa avere accesso a tali dati in ogni istante.

#### **Prelazione dei programmi utente**

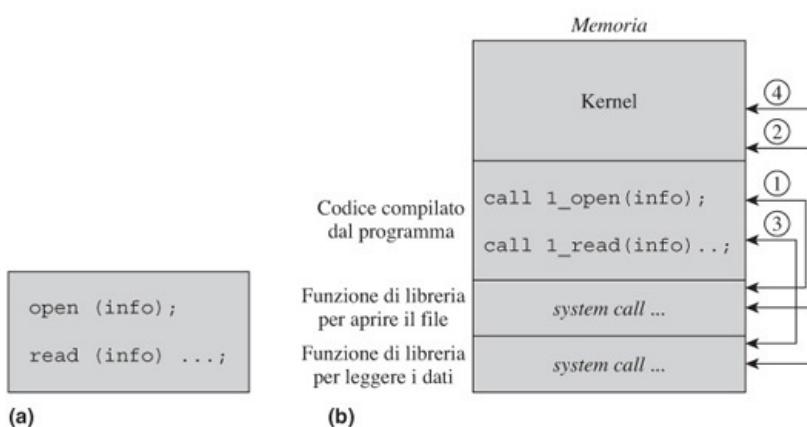
Nello schema di [Figura 2.7](#), la prelazione di un programma utente si verifica implicitamente quando viene generato un interrupt durante la sua esecuzione e il kernel decide di commutare la CPU all'esecuzione di qualche altro programma. Va ricordato dall'Esempio 2.3 che il contesto dei programmi interrotti è conservato nella *tabella dei programmi* e dunque non c'è nessuna difficoltà nel ripristinare l'esecuzione di un programma prelazionato quando viene schedulato nuovamente.

### **2.3.3 System call**

Un programma, durante la sua esecuzione, ha bisogno di utilizzare le risorse del computer, come per esempio le periferiche di I/O. Tuttavia, le risorse sono condivise tra i programmi utente e dunque è necessario prevenire interferenze reciproche nel loro utilizzo. Per facilitare la gestione delle risorse, le istruzioni che allocano o hanno accesso a risorse critiche sono implementate come istruzioni privilegiate. In questo modo, queste istruzioni non possono essere eseguite a meno che la CPU non si trovi in modalità kernel, e quindi i programmi utente non possono accedere direttamente alle risorse, ma devono effettuare una richiesta al kernel e il kernel deve accedere alle risorse per loro conto. Il kernel fornisce un insieme di servizi per questo scopo.

Dal punto di vista di un programmatore, un programma utilizza le risorse di un computer attraverso le istruzioni di un linguaggio di programmazione. Il compilatore implementa il punto di vista del programmatore come segue: mentre compila un programma, sostituisce le istruzioni riguardanti l'uso delle risorse del computer con chiamate alle funzioni di libreria che implementano l'uso delle risorse. Queste funzioni di libreria sono successivamente linkate con il programma utente. Durante l'esecuzione, il programma utente richiama una funzione di libreria e la funzione di libreria usa la risorsa attraverso un servizio del kernel. C'è ancora bisogno di un metodo attraverso il quale una funzione di libreria possa invocare il kernel per utilizzare uno dei suoi servizi. A tale metodo viene dato il termine generico di *system call* (chiamata di sistema).

La [Figura 2.10](#) mostra uno schema di questa organizzazione. Il programma mostrato in [Figura 2.10\(a\)](#) apre il file info e legge alcuni dati. Il programma compilato ha la forma mostrata in [Figura 2.10\(b\)](#). Il programma chiama una funzione di libreria per aprire il file; questa funzione è mostrata dalla freccia ①. La funzione di libreria richiama il servizio del kernel per l'apertura di un file attraverso una system call (freccia ②). Il servizio del kernel restituisce il controllo alla funzione di libreria dopo aver aperto il file, che restituisce il controllo al programma utente. Il programma legge il file in maniera analoga attraverso una chiamata a una funzione di libreria, che porta a una system call (frecce ③ e ④).



**Figura 2.10** Uno schema di chiamata di sistema: (a) un programma e (b) una organizzazione al momento di esecuzione.

Una system call è di fatto implementata attraverso il metodo degli interrupt descritto precedentemente, dunque la si può definire come segue:

**Definizione 2.1 System call** Una richiesta che un programma fa al kernel attraverso un interrupt software.

Si assume che l'istruzione di interrupt software menzionata nel [Paragrafo 2.2.5](#) abbia la forma

SI *<int\_code>*

dove il valore *<int\_code>*, che tipicamente è un intero nell'intervallo 0-255, indica quale servizio del kernel è stato richiesto. Un interrupt da programma si verifica quando un programma esegue questa istruzione e il primo passo della gestione dell'interrupt, come mostrato nella [Figura 2.6](#), copia *<int\_code>* nel campo *codice dell'interrupt* (IC) del PSW. La routine di servizio dell'interrupt per l'interrupt di programma analizza il campo del codice di interrupt nell'area delle informazioni salvate del PSW per conoscere la richiesta fatta dal programma. Una system call può prendere parametri che forniscono informazioni importanti al servizio del kernel invocato. Per esempio, la system call per aprire un file in [Figura 2.10](#) riceverebbe l'informazione del nome del file come parametro e la system call per leggere i dati riceverebbe i parametri che indicano il nome del file, il numero di byte di dati da leggere, l'indirizzo in memoria dove i dati letti devono essere

memorizzati, ecc. Differenti metodi possono essere usati per passare i parametri: i parametri possono essere caricati nei registri prima che la system call sia invocata, possono essere copiati sullo stack, oppure possono essere memorizzati in un'area di memoria e l'indirizzo di inizio dell'area di memoria può essere passato attraverso un registro o lo stack.

Il prossimo esempio descrive l'esecuzione di una system call per ottenere l'ora corrente.

#### Esempio 2.4 System call in un ipotetico SO

Un ipotetico SO fornisce una system call per ottenere l'ora corrente. Sia 78 il codice per questo servizio. Quando un programma desidera conoscere l'ora corrente, esegue l'istruzione SI 78, che causa un interrupt software. Il numero 78 è inserito nel campo del codice dell'interrupt del PSW prima che il PSW sia salvato nell'area di memoria del PSW salvato. In questo modo il valore  $d_1$  nel campo IC del PSW salvato in [Figura 2.8](#) sarebbe 78. Come mostrato in [Figura 2.8](#), il vettore degli interrupt per i program interrupt contiene *aaa* nel suo campo PC. Dunque la CPU è trasferita all'esecuzione della routine che inizia all'indirizzo *aaa*. La routine scopre che il codice interrupt è 78 e deduce che il programma vuole conoscere l'ora. In base alle convenzioni adottate nel SO, l'informazione relativa all'ora deve essere restituita al programma in una locazione standard, solitamente in un registro dati. Dunque il kernel memorizza questo valore nella locazione di memoria in cui il contenuto del registro dati viene salvato quando si verifica un interrupt. Questo valore viene caricato nel registro dati quando la CPU è trasferita per l'esecuzione nuovamente al programma interrotto.

Secondo lo schema di [Figura 2.10](#), si assumerà che un programma scritto in un linguaggio di programmazione come C, C++ o Java richiami la funzione di libreria quando ha bisogno di un servizio del SO e che la funzione di libreria effettui la chiamata a una system call per richiedere il servizio. Inoltre si userà la convenzione che il nome della funzione di libreria è anche il nome della system call. Per esempio, nell'Esempio 2.4, un programma C chiamerebbe una funzione di libreria `gettimeofday` per ottenere l'ora del giorno e questa funzione effettuerebbe la system call `gettimeofday` attraverso l'istruzione SI 78 come descritto nell'Esempio 2.4.

Un sistema operativo fornisce system call per vari scopi come l'avvio e la terminazione dei programmi, per la sincronizzazione dei programmi, per le operazioni su file e per ottenere informazioni circa il sistema. Il sistema operativo Linux mette a disposizione quasi 200 system call; alcune di queste chiamate sono elencate nella [Tabella 2.4](#). Queste system call possono anche essere richiamate in un programma C o C++ attraverso i nomi riportati nella [Tabella 2.4](#); un programma in linguaggio assembly può richiamarle direttamente attraverso l'istruzione SI.

| Numero della system call | Nome della system call | Descrizione                                      |
|--------------------------|------------------------|--|
| 1                        | exit                   | Termina l'esecuzione di questo programma         |
| 3                        | read                   | Legge i dati da un file                          |
| 4                        | write                  | Scrive i dati in un file                         |
| 5                        | open                   | Apre un file                                     |
| 6                        | close                  | Chiude un file                                   |
| 7                        | waitpid                | Attende che l'esecuzione di un programma termini |
| 11                       | execve                 | Esegue un programma                              |
| 12                       | chdir                  | Cambia la directory di lavoro                    |
| 14                       | chmod                  | Cambia i permessi del file                       |
| 39                       | mkdir                  | Crea una nuova directory                         |
| 74                       | sethostname            | Imposta il nome dell'host                        |
| 78                       | gettimeofday           | Recupera l'ora corrente                          |
| 79                       | settimeofday           | Imposta l'ora corrente                           |

**Tabella 2.4** System call.

## Riepilogo

Come accennato nel primo capitolo, un SO moderno può servire diversi programmi utente simultaneamente. Il SO raggiunge questo risultato interagendo con il computer e i programmi utente per effettuare diverse funzioni di controllo. In questo capitolo sono state descritte alcune caratteristiche rilevanti di un computer ed è stato discusso come queste sono utilizzate dal SO e dai programmi utente.

Il sistema operativo è una collezione di routine. Le istruzioni nelle routine devono essere eseguite dalla CPU per realizzare le funzioni di controllo. In questo modo la CPU dovrebbe eseguire le istruzioni contenute nel SO quando si verifica una situazione che richiede l'attenzione del sistema operativo, mentre dovrebbe eseguire le istruzioni contenute nei programmi utente negli altri casi. Questo è ottenuto inviando un segnale speciale, chiamato *interrupt*, alla CPU. Gli interrupt sono inviati al verificarsi di una situazione come, per esempio, il completamento di un'operazione di I/O o in caso di errori. Un interrupt software conosciuto come *system call* è inviato da un programma che richiede un servizio del kernel come, per esempio, l'allocazione di una risorsa o l'apertura di un file. La CPU contiene un insieme di *registri di controllo* il cui contenuto governa il suo funzionamento. La *program status word* (PSW) è una collezione di registri di controllo della CPU; ci si riferisce a ogni registro di controllo come a un campo del PSW. Un programma la cui esecuzione è stata interrotta deve essere riattivato successivamente. A tal fine, quando si verifica un interrupt, il kernel salva lo stato della CPU. Lo stato della CPU consiste del PSW e dei registri accessibili dal programma, che vengono chiamati registri general purpose (GPR). L'esecuzione del programma interrotto è ripristinata ricaricando lo stato salvato della CPU nel PSW e nei GPR.

La CPU ha due modalità operative controllate dal campo *modalità* (M) del PSW. Quando la CPU è in *modalità utente*, non può eseguire istruzioni sensibili come quelle che caricano le informazioni nei campi del PSW, per esempio il campo *modalità*, mentre può eseguire tutte le istruzioni quando si trova in *modalità kernel*. Il SO impone la CPU in modalità utente quando esegue un programma utente e impone la CPU in modalità kernel quando esegue le istruzioni nel kernel. Questa organizzazione previene la situazione in cui un programma esegua delle istruzioni che potrebbero interferire con altri programmi nel sistema.

La gerarchia di memoria di un computer fornisce lo stesso effetto di una memoria veloce e capiente, sebbene a un basso costo. Essa contiene una memoria molto veloce e molto piccola chiamata *cache*, una memoria ad accesso casuale (RAM) più lenta e capiente - che viene chiamata semplicemente memoria - e un disco. La CPU accede solo alla cache. Tuttavia, la cache contiene solo alcune parti delle istruzioni e dei dati di un programma. Le altre parti risiedono in memoria; queste verranno caricate dall'hardware della cache quando la CPU cerca di accedervi. Il tempo di accesso effettivo dipende da quale frazione di istruzioni e dati referenziati dalla CPU viene trovata nella cache; questa frazione è chiamata *hit ratio*.

Il sistema di input-output è il più lento di un computer; la CPU può eseguire milioni di istruzioni nella quantità di tempo richiesta per effettuare un'operazione di I/O. Alcuni metodi per effettuare un'operazione di I/O richiedono la partecipazione della CPU, che in questo modo spreca del tempo. Dunque il sistema di input-output di un computer utilizza la tecnologia *direct memory access* (DMA) per consentire alla CPU e al sistema di I/O di operare in maniera indipendente. Il sistema operativo sfrutta queste caratteristiche per consentire alla CPU di eseguire le istruzioni in un programma mentre sono in esecuzione operazioni di I/O dello stesso programma o di programmi differenti. Questa tecnica riduce il tempo idle (o di inattività) della CPU e migliora le prestazioni del sistema.

## Domande

2.1. Classificare ognuna delle seguenti affermazioni come vera o falsa:

- a. Il codice di condizione (flag) impostato da un'istruzione non è parte dello stato della CPU.
- b. Lo stato della CPU cambia quando un programma esegue un'istruzione no-op (nessuna operazione).

- c. L'istruzione di interrupt software (SI) cambia la modalità della CPU in modalità kernel.
  - d. Le istruzioni di salto in un programma possono portare a una bassa località spaziale, ma possono fornire un'alta località temporale.
  - e. Quando si utilizza il DMA, la CPU è coinvolta nei trasferimenti verso una periferica di I/O durante un'operazione di I/O.
  - f. Una violazione di protezione della memoria provoca un interrupt da programma.
  - g. Il kernel viene a conoscenza del fatto che un'operazione di I/O è stata completata quando un programma effettua una system call per informarlo che l'operazione di I/O è stata conclusa.
- 2.2. Quale delle seguenti istruzioni dovrebbe essere privilegiata? Spiegare perché.
- a. Impostare la CPU in modalità kernel.
  - b. Caricare il registro limite.
  - c. Caricare un valore in un registro general-purpose.
  - d. Mascherare alcuni interrupt.
  - e. Terminare forzatamente un'operazione di I/O.

## Problemi

- 2.1. Che uso fa il kernel del codice di interrupt nel PSW?
- 2.2. La CPU dovrebbe essere in modalità kernel mentre sta eseguendo il codice kernel e in modalità utente quando esegue un programma utente. Spiegare come ciò è ottenuto durante il funzionamento di un SO.
- 2.3. Il kernel di un SO maschera tutti gli interrupt durante la gestione dell'interrupt. Discutere i vantaggi e gli svantaggi di tale mascheramento.
- 2.4. Un computer utilizza timer basati sui tick del clock come descritto nel Paragrafo 2.2.5. Spiegare come questa organizzazione può essere usata per mantenere l'ora del giorno. Quali sono le limitazioni a questo approccio?
- 2.5. Un SO supporta una system call sleep, che sospende l'esecuzione del programma chiamante per il numero di secondi specificato nell'argomento della *sleep*. Spiegare come può essere implementata questa system call.
- 2.6. Un computer organizza l'area delle informazioni salvate del PSW come uno stack. Inserisce il contenuto del PSW nello stack durante il secondo passo della gestione dell'interrupt ([Figura 2.6](#)). Spiegare i vantaggi della gestione degli interrupt mediante uno stack.
- 2.7. Se la richiesta fatta da un programma attraverso una system call non può essere soddisfatta direttamente, il kernel informa lo scheduler che il programma dovrebbe non essere selezionato per l'esecuzione finché la sua richiesta non è soddisfatta. Fare un esempio di questa richiesta.
- 2.8. Un ipotetico SO fornisce una system call per richiedere l'allocazione della memoria. Un programmatore esperto offre il seguente consiglio: "Se il tuo programma contiene molte richieste di memoria, puoi velocizzare la sua esecuzione combinando tutte queste richieste e fare una singola system call." Spiegare perché.
- 2.9. Un computer ha due livelli di memoria cache, che consentono tempi di accesso che sono 0.01 e 0.1 volte il tempo di accesso alla memoria. Se l'hit ratio in ogni cache è 0.9, la memoria ha un tempo di accesso di 10 microsecondi e il tempo richiesto per caricare un blocco di cache è 5 volte il tempo di accesso della memoria più lenta, calcolare il tempo effettivo di accesso alla memoria.
- 2.10. Un computer ha una CPU che può eseguire 10 milioni di istruzioni al secondo e una memoria che ha un tasso di trasferimento di 100 Mbyte/secondo. Quando viene lanciato un interrupt I/O, la routine di servizio dell'interrupt deve eseguire 50 istruzioni per trasferire 1 byte tra la memoria e una periferica di I/O. Qual è il massimo tasso di trasferimento durante le operazioni di I/O implementate utilizzando le seguenti modalità di I/O: (a) interrupt di I/O e (b) I/O basato su DMA.
- 2.11. Diverse unità di una periferica di I/O, che ha un tasso di trasferimento di picco di 10 Kbyte/secondo e opera in modalità interrupt di I/O, sono connesse al computer nel Problema 2.10. Quante di queste unità operano a piena velocità allo stesso tempo?
- 2.12. Un ipotetico SO supporta due system call per effettuare le operazioni di I/O. La system call *init\_io* avvia un'operazione di I/O e la system call *await\_io* assicura che il programma venga eseguito nuovamente solo dopo che l'operazione di I/O è stata

completata. Spiegare tutte le azioni che hanno luogo quando il programma effettua queste due system call. (*Suggerimento:* quando nessun programma del SO può andare in esecuzione sulla CPU, il SO può impostare la CPU in un ciclo infinito nel quale non fa niente. Uscirà dal ciclo quando si verifica un interrupt.)

## Note bibliografiche

Smith (1982) e Handy (1998) descrivono l'organizzazione della memoria cache. Przybylski (1990) discute la progettazione della gerarchia della cache e della memoria. La gerarchia della memoria e l'organizzazione dell'I/O sono anche affrontate in molti libri sulle architetture e l'organizzazione dei computer, per esempio, Hayes (1997), Patterson e Hennessy (2005), Hennessy e Patterson (2002), Hamacher et al. (2002), e Stallings (2003).

Molti libri sui sistemi operativi discutono l'interfaccia delle chiamate di sistema. Bach (1986) contiene un'utile sintassi delle system call di Unix. O'Gorman (2003) descrive la gestione degli interrupt in Linux. Beck et al. (2002), Bovet e Cesati (2005), e Love (2005) contengono ampie discussioni sulle system call di Linux. Mauro e McDougall (2006) descrivono le system call in Solaris, mentre Russinovich e Solomon (2005) descrivono le system call in Windows.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice-Hall, Englewood Cliffs, N.J.
2. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
3. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
4. O'Gorman, J. (2003): *Linux Process Manager: The internals of Scheduling, Interrupts and Signals*, John Wiley, New York.
5. Hamacher, C., Z. Vranesic, and S. Zaky (2002): *Computer Organization*, 5th ed., McGraw-Hill, New York.
6. Handy, J. (1998): *The Cache Memory Book*, 2nd ed., Academic Press, New York.
7. Hayes, J. (1997): *Computer Architecture and Organization*, 3rd ed., McGraw-Hill, New York.
8. Hennessy, J., and D. Patterson (2002): *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, San Mateo, Calif.
9. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
10. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
11. Patterson, D., and J. Hennessy (2005): *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed., Morgan Kaufman, San Mateo, Calif.
12. Przybylski, A. (1990): *Cache and Memory Hierarchy Design: A Performance-Directed Approach*, Morgan Kaufmann, San Mateo, Calif.
13. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
14. Smith, A. J. (1982): "Cache memories," *ACM Computing Surveys*, 14, 473-530.
15. Stallings, W. (2003): *Computer Organization and Architecture*, 6th ed., Prentice Hall, Upper Saddle River, N.J.
16. Tanenbaum, A. (1998): *Structured Computer Organization*, 4th ed., Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 3

## Panoramica dei sistemi operativi

---

### **Obiettivi di apprendimento**

- Ambienti di elaborazione
- Classi dei sistemi operativi
- Caratteristiche di un sistema operativo: efficienza, prestazioni e servizio per l'utente
- Tipi di sistema operativo: multiprogrammato, time-sharing, real-time e distribuiti
- I moderni sistemi operativi

Volendo descrivere un *ambiente di elaborazione*, bisogna prendere in considerazione sia il sistema che i suoi utenti: Come è costruito il sistema? Come è installato per interagire con altri sistemi? Quali servizi mette a disposizione degli utenti? Tutte queste caratteristiche dell'ambiente di elaborazione influenzano la progettazione di un sistema operativo perché il SO deve fornire un'appropriata combinazione di uso efficiente delle risorse del computer e di convenienza per gli utenti, per la quale nel [Capitolo 1](#) abbiamo utilizzato la nozione di *utilizzo efficace*, e inoltre deve prevenire le interferenze nelle attività dei suoi utenti.

Nel corso della storia dell'elaborazione, gli ambienti di elaborazione si sono evoluti con i cambiamenti delle architetture dei computer e delle aspettative degli utenti. Nuove richieste di utilizzo sono emerse con ogni nuovo ambiente di elaborazione, per cui è stata sviluppata una nuova classe di sistemi operativi sfruttando nuovi concetti e tecniche per ottenere l'utilizzo efficace.

I moderni ambienti di elaborazione supportano diverse applicazioni e dunque possiedono le caratteristiche di molti classici ambienti di elaborazione. Di conseguenza, molti concetti e tecniche dei classici ambienti di elaborazione possono essere ritrovati nelle strategie utilizzate dai moderni sistemi operativi. Per semplificare lo studio dei moderni sistemi operativi, in questo capitolo presentiamo una panoramica dei concetti e delle tecniche dei classici ambienti di elaborazione e discutiamo quali trovano posto in un moderno sistema operativo.

### **3.1 Ambienti di elaborazione e natura delle elaborazioni**

Un *ambiente di elaborazione* consiste di un computer, delle sue interfacce con altri sistemi e dei servizi messi a disposizione dal sistema operativo agli utenti e ai loro programmi. Gli ambienti di elaborazione evolvono di continuo per fornire una migliore qualità del servizio agli utenti; tuttavia, il sistema operativo deve svolgere compiti più complessi al crescere della potenza computazionale dei computer, della complessità delle interfacce con le periferiche di I/O e con altri sistemi, e della richiesta da parte degli utenti di nuovi servizi.

La natura delle elaborazioni nell'ambiente di elaborazione e il modo in cui gli utenti le realizzano, dipende dalle caratteristiche dell'ambiente di elaborazione. In un tipico ambiente di elaborazione moderno, un utente avvia diverse attività simultaneamente, per esempio può lanciare un gestore della posta, modificare file, avviare elaborazioni, ascoltare musica o guardare video e navigare in Internet allo stesso tempo. Il sistema operativo deve fornire le risorse necessarie richieste da ognuna di questa attività, come la CPU, la memoria e dispositivi di I/O presenti nello stesso computer o in un altro computer cui si ha accesso attraverso Internet, in modo tale che le attività progrediscano con soddisfazione per l'utente.

Cominceremo la discussione relativa ai sistemi operativi dando uno sguardo veloce a come gli ambienti di elaborazione si sono evoluti fino alla loro configurazione attuale.

#### **Ambienti di elaborazione non interattivi**

Rappresentano le forme più vecchie di ambienti di elaborazione. In questi ambienti, un utente fornisce al sistema operativo sia il programma che i dati. L'elaborazione viene effettuata dal sistema operativo e i risultati vengono restituiti all'utente. L'utente non può intervenire nell'elaborazione durante l'esecuzione. Dunque queste elaborazioni possono essere considerate come entità passive interpretate e realizzate dal sistema operativo. Esempi di elaborazioni non interattive sono le elaborazioni scientifiche, che implicano operazioni sui numeri, e gli aggiornamenti di database. In questi ambienti di elaborazione, l'obiettivo del sistema operativo è l'uso efficiente delle risorse.

Le elaborazioni utilizzate negli ambienti non interattivi sono programmi o *job*. Un programma è un insieme di funzioni o moduli che possono essere eseguiti indipendentemente. Un *job* è una sequenza di programmi che insieme raggiungono l'obiettivo desiderato; un programma di un *job* viene eseguito solo se i programmi precedenti del *job* sono stati eseguiti con successo. Per esempio, si consideri il procedimento di compilazione, di linking ed esecuzione di un programma C++. Un *job* per effettuare queste azioni dovrebbe eseguire un compilatore C++, eseguire un linker per collegare il programma alle funzioni di libreria, e infine eseguire il programma completo. In questo caso, la fase di linking ha senso solo se il programma è stato compilato correttamente e l'esecuzione ha senso solo se la fase di linking è stata eseguita con successo.

### **Ambienti di elaborazione interattivi**

In questi ambienti di elaborazione, un utente può interagire con l'elaborazione mentre è in esecuzione. La natura di una interazione tra un utente e la sua elaborazione dipende da come l'elaborazione è stata codificata; per esempio, un utente, durante l'esecuzione, può inserire il nome di un file contenente i dati a necessari per l'elaborazione, o può inserirli direttamente. L'obiettivo del sistema operativo è di ridurre il tempo medio richiesto per implementare l'interazione tra un utente e la sua elaborazione.

Un utente interagisce anche con il SO per avviare un'elaborazione; tipicamente ogni comando dell'utente risulta nell'esecuzione di un programma. In questo caso la nozione di *job* non è importante poiché un utente considererebbe la dipendenza dei programmi prima di eseguire il comando successivo. Per esempio, se bisogna compilare, linkare ed eseguire un programma C++, un utente cercherebbe di linkare solo un programma correttamente compilato. Dunque i sistemi operativi per l'elaborazione interattiva gestiscono esclusivamente l'esecuzione di programmi e non di *job*. Nella letteratura riguardante i SO si utilizza il termine *processo* riferito all'esecuzione di un programma in un ambiente di elaborazione interattivo. In linea di principio, il termine *processo* è applicabile sia negli ambienti di elaborazione interattivi che non interattivi. Tuttavia, seguiremo la convenzione e utilizzeremo il termine *processo* solo nel contesto degli ambienti di elaborazione interattivi.

L'interazione con un *processo* consiste, per un utente, nel presentare una richiesta di elaborazione, una *sottorichiesta*, al *processo* e per il *processo* nel restituire una risposta. In base al tipo di *sottorichiesta*, la risposta può essere nella forma di un insieme di risultati, o un insieme di azioni, come operazioni su file o aggiornamenti di database. La [Tabella 3.1](#) descrive le elaborazioni del programma, del *job*, del *processo* e della *sottorichiesta*.

| <b>Elaborazione</b> | <b>Descrizione</b>  |
|---------------------|---|
| Programma           | Un <i>programma</i> è un insieme di funzioni o moduli, che includono alcune funzioni o moduli contenuti nelle librerie.   |
| Job                 | Un <i>job</i> è una sequenza di programmi che insieme raggiungono un obiettivo comune. Non ha senso eseguire un programma in un <i>job</i> se il programma precedente del <i>job</i> non è stato eseguito con successo. |
| Processo            | Un <i>processo</i> è l'esecuzione di un programma.  |
| Sottorichiesta      | Una <i>sottorichiesta</i> è una richiesta di elaborazione effettuata da un utente a un <i>processo</i> . Ogni <i>sottorichiesta</i> produce una singola risposta, che consiste di un insieme di risultati o azioni.     |

**Tabella 3.1** Elaborazioni in un SO.

### **Ambienti real-time, distribuiti ed embedded**

Alcune elaborazioni hanno delle necessità speciali, per le quali sono stati sviluppati speciali ambienti di elaborazione. Un'*elaborazione real-time* deve rispettare specifici vincoli temporali, per cui le sue azioni sono efficaci solo se completate in uno specifico intervallo di tempo. Per esempio, un'elaborazione che periodicamente campiona i dati di uno strumento e memorizza i campioni in un file, deve completare la memorizzazione di un campione prima di prendere il prossimo campione. Il sistema operativo in un ambiente real-time utilizza tecniche speciali per assicurare che le elaborazioni siano completate rispettando i vincoli temporali. L'*ambiente di elaborazione distribuito* consente di utilizzare le risorse presenti in diversi sistemi attraverso una rete. Nell'*ambiente di elaborazione embedded*, il computer è parte di uno specifico sistema hardware, come un elettrodomestico, il sottosistema di una automobile, o un personal digital assistant (PDA) ed esegue elaborazioni che effettivamente controllano il sistema. Il computer con una configurazione minimale generalmente non è costoso e il suo SO deve soddisfare i vincoli temporali derivanti dalla natura del sistema controllato.

### **I moderni ambienti di elaborazione**

Per supportare diverse applicazioni, l'ambiente di elaborazione di un moderno computer implementa le caratteristiche dei diversi ambienti di elaborazione precedentemente descritti. Di conseguenza, il sistema operativo deve adottare complesse strategie per gestire le elaborazioni degli utenti e le risorse; per esempio, deve ridurre il tempo medio richiesto per implementare l'interazione tra un utente e l'applicazione e anche assicurare un uso efficiente delle risorse.

Affrontiamo lo studio delle strategie utilizzate nei moderni sistemi operativi in due fasi: in questo capitolo, studieremo le strategie utilizzate dai sistemi operativi in ognuno degli ambienti di elaborazione menzionati precedentemente, e successivamente vedremo quali di questi risultano utili in un moderno ambiente di elaborazione. Nei capitoli successivi, discuteremo la progettazione delle strategie utilizzate nei moderni sistemi operativi.

## **3.2 Classi di sistemi operativi**

Le classi di sistemi operativi si sono evolute nel tempo con l'evoluzione dei computer e delle aspettative degli utenti, ovvero con l'evoluzione degli ambienti di elaborazione. Studiando alcune delle prime classi di sistemi operativi, è necessario comprendere che ogni SO è stato progettato per operare con i sistemi disponibili in un determinato periodo storico; in questo modo avremo una visione delle caratteristiche delle architetture rappresentative dei computer relativamente al periodo in esame.

La **Tabella 3.2** elenca cinque classi fondamentali di sistemi operativi il cui nome rispecchia le loro caratteristiche peculiari. La tabella mostra il periodo in cui i sistemi operativi di ogni classe hanno avuto diffusione, quale obiettivo principale ha motivato il loro sviluppo, e quali concetti chiave furono sviluppati per ottenere questo obiettivo.

| Classe del SO       | Periodo  | Obiettivo principale           | Concetti chiave                        |
|---------------------|----------|--------------------------------|--|
| Elaborazione batch  | anni '60 | Tempo idle della CPU           | Automatizzare la transizione tra i job |
| Multiprogrammazione | anni '60 | Utilizzo delle risorse         | Proprietà del programma, prelazione    |
| Time-sharing        | anni '70 | Buon tempo di risposta         | Scheduling, time slice, round-robin    |
| Real time           | anni '80 | Rispettare i vincoli temporali | Scheduling real-time                   |
| Distribuiti         | anni '90 | Condivisione delle risorse     | Controllo distribuito, trasparenza     |

**Tabella 3.2** Caratteristiche chiave delle classi di sistemi operativi.

L'hardware di elaborazione era costoso nei primi anni, pertanto l'elaborazione batch e i sistemi operativi multiprogrammati si focalizzavano sull'uso efficiente della CPU e delle altre risorse del computer. Gli ambienti di elaborazione, a quei tempi, non erano interattivi. Negli anni '70, l'hardware dei computer divenne più economico, così l'uso efficiente di un computer non fu più il primo obiettivo, che si spostò verso la produttività degli utenti. Furono sviluppati gli ambienti di elaborazione interattivi e i sistemi operativi

time-sharing facilitarono una migliore produttività fornendo risposte rapide alle sottorichieste fatte ai processi. Gli anni '80 videro l'avvento delle applicazioni real-time per il controllo o il tracciamento delle attività del mondo reale, e di conseguenza i sistemi operativi si focalizzarono sul rispetto dei vincoli temporali di queste applicazioni. Negli anni '90, l'ulteriore calo dei prezzi dell'hardware condusse allo sviluppo dei sistemi distribuiti, grazie ai quali diversi computer condividevano le proprie risorse attraverso la rete.

Il paragrafo seguente elabora i concetti chiave delle cinque classi di sistemi operativi menzionati nella [Tabella 3.2](#).

### **Sistemi di elaborazione batch**

In un sistema operativo per l'elaborazione batch, l'obiettivo principale è l'uso efficiente della CPU. Il sistema di elaborazione batch opera elaborando un job alla volta; all'interno del job, esegue i programmi uno dopo l'altro. In questo modo solo un programma è in esecuzione in un dato momento. Il miglioramento dell'efficienza della CPU è vincolato dalla velocità con cui viene lanciato un programma o un job quando termina il precedente in modo tale che la CPU non rimanga inutilizzata.

### **Sistemi multiprogrammati**

Un sistema operativo multiprogrammato si focalizza sull'uso efficiente sia della CPU che dei dispositivi di I/O. Il sistema gestisce diversi programmi in uno stato di parziale completamento per ogni istante di tempo. Il SO utilizza le *priorità del programma* e concede l'utilizzo della CPU al programma, pronto per l'esecuzione, con priorità più alta. La CPU viene commutata a un programma con priorità bassa quando un programma a più alta priorità inizia un'operazione di I/O e viene commutata al programma a più alta priorità alla fine dell'operazione di I/O. Queste azioni consentono di ottenere l'uso simultaneo dei dispositivi di I/O e della CPU.

### **Sistemi time-sharing**

Un sistema operativo time-sharing si focalizza sul miglioramento della velocità di risposta alle sottorichieste fatte da *tutti* i processi, che si traduce in un beneficio tangibile per gli utenti. Questo obiettivo è ottenuto dando un'equa opportunità di esecuzione a ogni processo attraverso due metodi: il SO serve tutti i processi a turno (*scheduling round-robin*) ed evita che un processo possa utilizzare troppo tempo di CPU quando schedulato per l'esecuzione (*time-slicing*). La combinazione di queste due tecniche assicura che nessun processo aspetti troppo a lungo per ottenere la CPU.

### **Sistemi real-time**

Un sistema operativo real-time viene utilizzato per implementare un'applicazione dedicata al controllo o al tracciamento di attività del mondo reale. L'applicazione ha la necessità di completare le sue attività di elaborazione in maniera cadenzata per rispondere agli eventi esterni relativi all'attività controllata. Per facilitare questo compito, il SO consente a un utente di creare diversi processi all'interno di un programma e utilizza lo *scheduling real-time* per alternare l'esecuzione dei processi in modo che l'applicazione possa completare la sua esecuzione rispettando i vincoli temporali.

### **Sistemi distribuiti**

Un sistema operativo distribuito consente a un utente di avere accesso alle risorse presenti in altri computer in modo conveniente ed efficace. Per migliorare la convenienza, il sistema operativo non richiede all'utente di conoscere dove si trovano le risorse; questa caratteristica è chiamata *trasparenza*. Per migliorare l'efficienza, il sistema operativo può eseguire parti di una elaborazione su computer differenti allo stesso tempo. Utilizza il *controllo distribuito*, ovvero distribuisce le azioni di decisione a differenti computer nel sistema in modo che i malfunzionamenti dei singoli computer o della rete non bloccino il suo funzionamento.

Nei Paragrafi 3.4-3.8, ognuna delle cinque classi fondamentali di SO verrà esaminata in maggior dettaglio.

## **3.3 Efficienza, prestazioni del sistema e servizio per l'utente**

Le misure forniscono un metodo per valutare aspetti selezionati del funzionamento di un sistema operativo. Nel [Capitolo 1](#), abbiamo definito l'efficienza nell'utilizzo e la convenienza per l'utente come due degli obiettivi fondamentali di un SO. Tuttavia, per un amministratore le prestazioni di un sistema nel suo ambiente sono più importanti della semplice efficienza nell'utilizzo, quindi in questo paragrafo discuteremo le misure dell'efficienza, delle prestazioni del sistema e del servizio per l'utente. La [Tabella 3.3](#) riassume queste misure.

| Aspetto                 | Misura                   | Descrizione                                       |
|-------------------------|--------------------------|---|
| Efficienza d'uso        | Efficienza della CPU     | Percentuale di utilizzo della CPU                 |
|                         | Efficienza della memoria | Percentuale di utilizzo della memoria             |
| Prestazioni del sistema | Throughput               | Quantità di lavoro svolto per unità di tempo      |
| Servizio per l'utente   | Tempo di turnaround      | Tempo di completamento di un job o di un processo |
|                         | Tempo di risposta        | Tempo di risposta a una sotrichiesta              |

**Tabella 3.3** Misure dell'efficienza, delle prestazioni del sistema e del servizio per l'utente.

### **Efficienza**

Valutare l'efficienza nell'uso di una risorsa equivale a vedere quanto la risorsa non è utilizzata o sprecata e, relativamente all'utilizzo della risorsa, quanto è stata produttiva. Come esempio di efficienza, si consideri l'uso della CPU. Una certa quantità di tempo di CPU è sprecata poiché la CPU non ha abbastanza lavoro da svolgere. Questo si verifica quando tutti i processi utente nel sistema o stanno eseguendo operazioni di I/O oppure sono in attesa che gli utenti forniscano dei dati. Una parte del tempo di CPU è utilizzata dal SO per la gestione degli interrupt e per lo scheduling. Questi costituiscono *l'overhead* del funzionamento del SO. Il rimanente tempo di CPU è utilizzato per eseguire i processi degli utenti. Per valutare l'efficienza dell'utilizzo della CPU, è necessario considerare quale frazione o percentuale del tempo totale di CPU è usato per eseguire i processi degli utenti. L'efficienza nell'uso delle altre risorse, come la memoria e i dispositivi di I/O, può essere determinata in modo analogo: sottrarre la quantità di risorsa non utilizzata e l'overhead del SO dal totale della risorsa e valutare la frazione o percentuale del risultato rispetto al totale della risorsa.

Utilizzando la nozione di efficienza, affrontiamo brevemente il compromesso fondamentale tra efficienza nell'uso e convenienza per l'utente: un sistema multiprogrammato esegue diversi programmi utente in ogni istante e ne alterna l'esecuzione per ottenere un utilizzo efficiente sia della CPU che dei dispositivi di I/O. La CPU è assegnata al programma attivo nel sistema con la priorità più alta nel momento in cui la richiede e può utilizzare la CPU per tutto il tempo che desidera. Un sistema time-sharing, tuttavia, riduce la quantità di tempo di CPU che un processo schedulato può utilizzare. Il SO prelaziona un processo che utilizza troppo tempo di CPU e lo schedula successivamente. Questa caratteristica aumenta l'overhead del SO relativamente alla gestione degli interrupt e allo scheduling, influenzando di conseguenza l'efficienza dell'utilizzo della CPU. Tuttavia, consente di ottenere buoni tempi di risposta per tutti i processi, caratteristica utile per gli utenti.

### **Prestazioni del sistema**

Stabilità la giusta combinazione di efficienza della CPU e servizio per l'utente, è importante poter misurare le prestazioni del SO. La nozione di prestazione dipende dall'ambiente di elaborazione e indica il tasso con il quale il computer effettua il lavoro durante il suo funzionamento.

Un sistema operativo generalmente utilizza una misura dell'efficienza per mettere a punto il suo funzionamento, così da ottenere prestazioni migliori. Per esempio, se l'efficienza della memoria è bassa, il sistema operativo può caricare più programmi degli utenti in memoria. Di conseguenza, ciò può portare a migliori prestazioni del sistema aumentando il tasso con il quale il sistema completa le elaborazioni degli utenti. Se l'efficienza della CPU è bassa, il sistema operativo può cercarne le cause (per esempio, pochi programmi in memoria o programmi che trascorrono troppo tempo ad aspettare il completamento di operazioni di I/O), in modo da mettere in atto azioni correttive dove

possibile.

Le prestazioni del sistema sono caratterizzate dalla quantità di lavoro svolto per unità di tempo e sono generalmente misurate dal *throughput*.

**Definizione 3.1 Throughput** Il numero di job, programmi, processi o sottorichieste completati da un sistema in una unità di tempo.

L'unità di lavoro utilizzata per misurare il throughput dipende dall'ambiente di elaborazione. In un ambiente non interattivo, il throughput di un SO è misurato in termini di numero di job o programmi completati in una unità di tempo. In un ambiente interattivo, può essere misurato in termini di numero di sottorichieste completate in una unità di tempo. In un ambiente di elaborazione specializzato, le prestazioni possono essere misurate in termini significativi per l'applicazione; per esempio, in un ambiente bancario, potrebbe essere il numero di transazioni per unità di tempo. Il throughput può essere anche usato come misura delle prestazioni dei dispositivi di I/O. Per esempio, il throughput di un disco può essere misurato come il numero di operazioni di I/O completate o come il numero di byte trasferiti in unità di tempo.

### **Servizio per l'utente**

Alcuni aspetti della convenienza per l'utente sono intangibili e dunque impossibili da misurare numericamente; per esempio, una caratteristica come l'usabilità delle interfacce non può essere quantificata. Tuttavia, ci sono alcuni aspetti misurabili della convenienza per l'utente e dunque per questi è possibile definire misure appropriate. Uno di questi aspetti è il *servizio per l'utente*, che indica quanto velocemente un'elaborazione dell'utente è stata completata dal SO. Definiamo due misure del servizio per l'utente, *tempo di turnaround* negli ambienti di elaborazione non interattivi e *tempo di risposta* negli ambienti di elaborazione interattivi. Un tempo di turnaround o un tempo di risposta piccolo implica un migliore servizio per l'utente.

**Definizione 3.2 Tempo di turnaround** Il tempo trascorso dalla sottomissione da parte di un utente di un job, di un programma o di un processo fino al momento in cui i risultati sono restituiti all'utente.

**Definizione 3.3 Tempo di risposta** Il tempo trascorso dalla sottomissione di una sottorichiesta da parte di un utente fino al momento in cui il processo risponde a essa.

Misure specifiche del servizio per l'utente possono essere definite per l'utilizzo in specifici ambienti di elaborazione. Due esempi sono il *deadline overrun* in un sistema operativo real-time e lo *speedup dell'elaborazione* in un sistema operativo distribuito. Il deadline overrun indica il ritardo del SO nel completare l'esecuzione di un'elaborazione con vincoli temporali, così che un deadline overrun negativo indica un buon servizio per l'utente. Lo speedup dell'elaborazione indica il fattore di incremento della velocità nell'esecuzione di un programma a causa del fatto che i suoi processi sono stati eseguiti allo stesso tempo su differenti computer di un sistema distribuito; un maggior valore di speedup dell'elaborazione implica un miglior servizio per l'utente.

## **3.4 Sistemi di elaborazione batch**

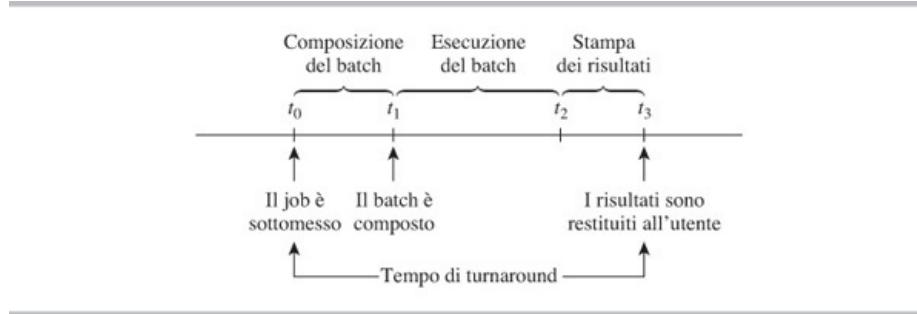
I computer degli anni '60 non erano interattivi. Le schede perforate erano il mezzo di input principale e dunque un job e i suoi dati consistevano in un gruppo di schede. Un operatore caricava le schede in un lettore per impostare l'esecuzione di un job. Questa azione causava una perdita di tempo prezioso di CPU; l'elaborazione batch fu introdotta per prevenire questo spreco.

Un *batch* è una sequenza di job utente assemblati per essere elaborati dal sistema operativo. Un operatore assemblava il batch organizzando alcuni job degli utenti in una sequenza e inserendo alcune speciali schede per indicare l'inizio e la fine del batch. Quando l'operatore forniva un comando per iniziare l'elaborazione di un batch, il

*batching kernel* impostava l'elaborazione del primo job del batch. Alla termine del job, cominciava l'esecuzione del job seguente e così via fino alla fine del batch. In questo modo l'operatore doveva intervenire solo all'inizio e alla fine del batch.

I lettori di schede e le stampanti rappresentavano un collo di bottiglia negli anni '60, così i sistemi di elaborazione batch utilizzarono il concetto di lettori di schede e stampanti virtuali (Paragrafo 1.3.2) attraverso i nastri magnetici, per migliorare il throughput del sistema. Un batch di job veniva preventivamente memorizzato su di un nastro magnetico, utilizzando un computer meno potente e costoso. Il sistema di elaborazione batch elaborava questi job dal nastro e scriveva i risultati su un altro nastro magnetico. Questi venivano successivamente stampati e restituiti agli utenti.

La [Figura 3.1](#) mostra i fattori che costituiscono il tempo di turnaround di un job.



**Figura 3.1** Tempo di turnaround in un sistema di elaborazione batch.

I job degli utenti non potevano interferire direttamente gli uni con gli altri perché non coesistevano nella memoria del computer. Tuttavia, poiché il lettore di schede era l'unico dispositivo di input disponibile per gli utenti, i comandi, i programmi degli utenti e i dati erano tutti smistati dal lettore di schede, cosicché se un programma in un job tentava di leggere più dati di quelli forniti nel job, avrebbe letto alcune schede del job seguente!

Per evitare il verificarsi di tali interferenze tra i job, un sistema di elaborazione batch richiedeva a un utente di inserire un insieme di *istruzioni di controllo* nel gruppo di schede che costituivano un job. L'*interprete dei comandi*, un componente del kernel batch, leggeva una scheda quando il programma, nel job attualmente in esecuzione, richiedeva la scheda successiva. Se la scheda conteneva un'istruzione di controllo, analizzava l'istruzione di controllo ed effettuava le azioni appropriate; in caso contrario, passava la scheda al programma attualmente in esecuzione. La [Figura 3.2](#) mostra un insieme semplificato di istruzioni di controllo usate per compilare ed eseguire un programma Fortran. Se un programma tentava di leggere più dati di quelli forniti, l'interprete dei comandi avrebbe letto le schede /\*, /& e // JOB. Analizzando una di queste schede, avrebbe realizzato che il programma stava cercando di leggere più schede di quelle fornite.



**Figura 3.2** Istruzioni di controllo nei sistemi IBM 360/370.

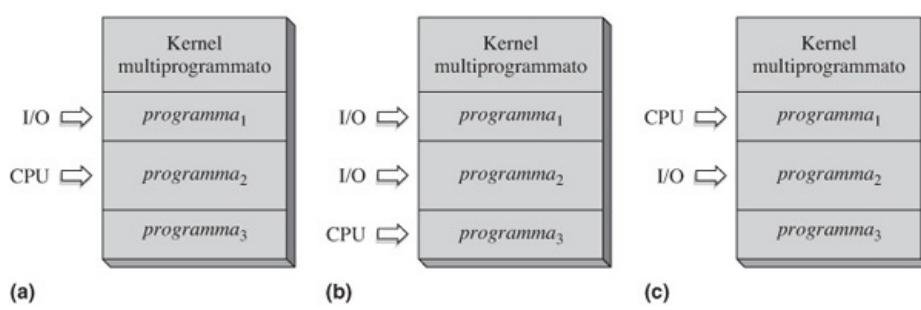
Un moderno SO non è *progettato* per l'elaborazione batch, ma la tecnica è ancora utile nelle elaborazioni finanziarie e scientifiche dove lo stesso tipo di elaborazione o analisi devono essere eseguite su diversi insiemi di dati. L'uso dell'elaborazione batch in tali ambienti eliminerebbe il processo, costoso in termini di tempo, dell'inizializzazione

separata di ogni insieme di dati delle analisi finanziarie o scientifiche.

### 3.5 Sistemi multiprogrammati

I sistemi operativi multiprogrammati furono sviluppati per fornire un utilizzo efficiente delle risorse in un ambiente di elaborazione non interattivo. Un SO multiprogrammato mantiene molti programmi utente in memoria, da cui il nome *multiprogrammazione*. Utilizza il DMA per le operazioni di I/O (Paragrafo 2.2.4), in modo da poter eseguire le operazioni di I/O di alcuni programmi mentre utilizza la CPU per eseguire altri programmi. Questa organizzazione fa un uso efficiente sia della CPU che dei dispositivi di I/O. L'I/O e le attività di elaborazione di diversi programmi sono in esecuzione allo stesso tempo, cosa che consente di ottenere elevate prestazioni dal sistema. Questo aspetto verrà affrontato nel Paragrafo 3.5.1.

La Figura 3.3 illustra il funzionamento di un SO multiprogrammato. La memoria contiene tre programmi. Un'operazione di I/O è in esecuzione per il *programma<sub>1</sub>*, mentre la CPU sta eseguendo il *programma<sub>2</sub>*. La CPU viene commutata al *programma<sub>3</sub>* quando *programma<sub>2</sub>* avvia un'operazione di I/O ed è commutata al *programma<sub>1</sub>* quando l'operazione di I/O viene completata. Il kernel multiprogrammato effettua lo scheduling, la gestione della memoria e la gestione dell'I/O. Utilizza una semplice politica di scheduling, che verrà spiegata nel Paragrafo 3.5.1 ed esegue semplici allocazioni della memoria e dei dispositivi di I/O basate sul partizionamento o sul pool. Poiché diversi programmi sono in memoria allo stesso tempo, le istruzioni, i dati e le operazioni di I/O di un programma dovrebbero essere protette dall'interferenza di altri programmi. Vedremo brevemente come viene raggiunto questo risultato.



**Figura 3.3** Funzionamento di un sistema multiprogrammato: (a) *programma<sub>2</sub>* è in esecuzione; (b) *programma<sub>2</sub>* avvia un'operazione di I/O, *programma<sub>3</sub>* viene schedulato; (c) l'operazione di I/O del *programma<sub>1</sub>* viene completata e il programma viene schedulato.

Un computer deve possedere le caratteristiche riassunte nella Tabella 3.4 per supportare la multiprogrammazione (Paragrafo 2.2). Il DMA rende possibile la multiprogrammazione consentendo operazioni concorrenti della CPU e dei dispositivi di I/O. La protezione della memoria previene l'accesso a locazioni di memoria che risiedono al di fuori del suo spazio di indirizzamento definito dal contenuto del *registro base* e del *registro size* della CPU. Le modalità utente e kernel della CPU forniscono un metodo effettivo per prevenire le interferenze tra programmi. Va ricordato dal Paragrafo 2.2 che il SO imposta la CPU in modalità utente quando esegue i programmi utente e che le istruzioni che caricano un indirizzo nel registro base e un numero nel registro limite della CPU sono istruzioni privilegiate. Se un programma cercasse di compromettere la protezione della memoria cambiando il contenuto dei registri base e limite attraverso queste istruzioni, sarebbe generato un interrupt da programma poiché la CPU è in modalità utente; il kernel terminerebbe il programma durante la gestione di questo interrupt.

| Caratteristica                     | Descrizione   |
|------------------------------------|---|
| DMA                                | La CPU avvia un'operazione di I/O quando viene eseguita un'istruzione di I/O. Il DMA implementa il trasferimento dei dati senza coinvolgere la CPU e genera un interrupt di I/O quando il trasferimento è completato.   |
| Protezione della memoria           | Un programma può accedere solo alla parte della memoria definita dal contenuto del <i>registro base</i> e del <i>registro size</i> .  |
| Modalità kernel e utente della CPU | Alcune istruzioni, chiamate <i>istruzioni privilegiate</i> , possono essere eseguite solo quando la CPU si trova in modalità kernel. Un interrupt di programma si verifica se un programma tenta di eseguire un'istruzione privilegiata quando la CPU è in modalità utente. |

**Tabella 3.4** Supporto dell'architettura per la multiprogrammazione.

Il tempo di turnaround di un programma è la misura appropriata del servizio per l'utente in un sistema multiprogrammato. Dipende dal numero totale di programmi in un sistema multiprogrammato, dal modo in cui il kernel condivide la CPU tra i programmi e dalle richieste di esecuzione proprie di ogni programma.

### 3.5.1 Priorità dei programmi

Una misura appropriata delle prestazioni di un SO multiprogrammato è il *throughput*, ovvero il rapporto del numero di programmi elaborati e il tempo totale impiegato per elaborarli. Il throughput di un SO multiprogrammato che elabora  $n$  programmi nell'intervallo di tempo tra  $t_0$  e  $t_f$  è  $n/(t_f - t_0)$ . Può essere maggiore del throughput di un sistema di elaborazione batch poiché le attività in diversi programmi possono aver luogo simultaneamente: un programma può eseguire istruzioni sulla CPU, mentre altri programmi effettuano operazioni di I/O. Tuttavia il throughput dipende dalla natura dei programmi eseguiti, ovvero da quanta elaborazione e da quanto I/O eseguono e quanto bene il kernel riesce a sovrapporre le loro attività nel tempo.

Il SO mantiene sempre un numero sufficiente di programmi in memoria, in modo che la CPU e i dispositivi di I/O abbiano lavoro sufficiente da effettuare. Questo numero è denominato *grado di multiprogrammazione*. Tuttavia un elevato grado di multiprogrammazione non può garantire un buon utilizzo sia della CPU sia dei dispositivi di I/O, perché la CPU resterebbe idle se ognuno dei programmi eseguisse operazioni di I/O per la maggior parte del tempo, oppure i dispositivi di I/O rimarrebbero idle se ognuno dei programmi eseguisse elaborazioni per la maggior parte del tempo. Dunque il SO multiprogrammato adotta le due tecniche descritte nella **Tabella 3.5** per assicurare una sovrapposizione della CPU e dei dispositivi di I/O nelle attività dei programmi: utilizza un appropriato *mix di programmi*, così che alcuni dei programmi in memoria siano *programmi CPU-bound*, ovvero programmi che necessitano di molta elaborazione ma poche operazioni di I/O, e altri siano *programmi I/O-bound*, che richiedono poca elaborazione ma effettuano molte operazioni di I/O. In questo modo, i programmi da eseguire hanno la possibilità di impegnare la CPU e i dispositivi di I/O simultaneamente. Il SO utilizza la nozione di *scheduling a priorità con prelazione* per condividere la CPU tra i programmi in modo tale da assicurare una buona sovrapposizione delle rispettive attività di CPU e I/O. Tali tecniche vengono spiegate di seguito.

| Tecniche                             | Descrizione   |
|--------------------------------------|---|
| Appropriato mix di programmi         | <p>Il kernel seleziona un mix di programmi CPU-bound e I/O-bound in memoria, dove</p> <ul style="list-style-type: none"> <li>• Un <i>programma CPU-bound</i> è un programma che coinvolge molta elaborazione e poco I/O. Utilizza la CPU per lunghi burst, ovvero utilizza la CPU per lunghi periodi di tempo prima di iniziare un'operazione di I/O.</li> <li>• Un <i>programma I/O-bound</i> effettua poca elaborazione e molto I/O. Utilizza la CPU in piccoli burst.</li> </ul> |
| Scheduling a priorità con prelazione | <p>A ogni programma è assegnata una priorità. La CPU è sempre allocata al programma con la priorità più alta pronto per l'esecuzione. Un programma a bassa priorità in esecuzione viene prelazionato se un programma a più alta priorità richiede l'uso della CPU.</p>  |

**Tabella 3.5** Tecniche di multiprogrammazione.

**Definizione 3.4 Priorità** Un criterio mediante il quale lo scheduler decide quale richiesta debba essere schedulata quando molte richieste sono in attesa di essere servite.

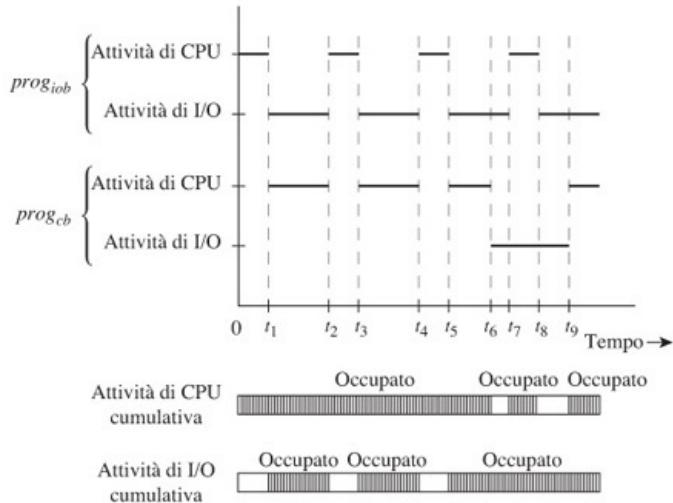
Il kernel assegna priorità numeriche ai programmi. Assumiamo che le priorità siano interi positivi e che un valore elevato implichi una priorità alta. Quando molti programmi hanno bisogno della CPU allo stesso tempo, il kernel assegna l'utilizzo della CPU al programma con la priorità più alta. Il kernel utilizza la priorità con prelazione, cioè prelaziona un programma a bassa priorità in esecuzione dalla CPU se un programma ad alta priorità richiede la CPU. In questo modo, la CPU esegue sempre il programma a più alta priorità che la richiede. Per comprendere le implicazioni dello scheduling con prelazione basato dalle priorità, si consideri cosa accadrebbe se un programma ad alta priorità stesse eseguendo un'operazione di I/O, un programma a bassa priorità fosse in esecuzione dalla CPU e l'operazione di I/O del programma a più alta priorità venisse completata: il kernel commuterebbe immediatamente la CPU al programma a più alta priorità.

L'assegnazione delle priorità ai programmi è una decisione cruciale che può influenzare il throughput del sistema. I sistemi multiprogrammati usano la seguente regola per l'assegnazione delle priorità: *un programma I/O-bound dovrebbe avere una priorità più alta di un programma CPU-bound*.

L'Esempio 3.1 illustra il funzionamento di questa regola.

#### **Esempio 3.1 Esecuzione di un programma in un sistema multiprogrammato**

Un sistema multiprogrammato ha  $prog_{iob}$ , un programma I/O-bound e  $prog_{cb}$ , un programma CPU-bound. Il suo funzionamento comincia al tempo 0. In [Figura 3.4](#), le attività di CPU e I/O di questi programmi sono tracciati nella forma di grafici temporali in cui l'asse x mostra il tempo e l'asse y mostra le attività di CPU e I/O dei due programmi. Le attività cumulative di CPU e I/O sono mostrate nella parte bassa del grafico. Bisogna notare che il grafico non è scalato; l'attività della CPU del  $prog_{iob}$  è stata ampliata per chiarezza.



**Figura 3.4** Grafico del tempo nel caso in cui il programma I/O-bound ha priorità più alta.

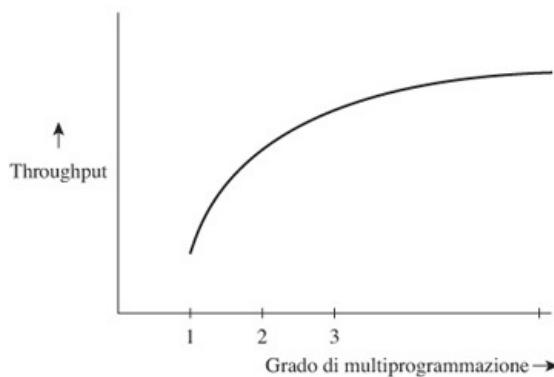
Il programma *prog<sub>io</sub>* è il programma a più alta priorità. Dunque comincia l'esecuzione al tempo 0. Dopo un breve burst di attività di CPU, avvia una operazione di I/O (istante di tempo  $t_1$ ). La CPU è ora commutata al *prog<sub>cb</sub>*. L'esecuzione del *prog<sub>cb</sub>* è in questo modo concorrente con l'operazione di I/O del *prog<sub>io</sub>*. Poiché è un programma CPU-bound, *prog<sub>cb</sub>* mantiene la CPU occupata finché *prog<sub>io</sub>* completa l'operazione di I/O al tempo  $t_2$ , quando *prog<sub>cb</sub>* viene prelazionato poiché *prog<sub>io</sub>* ha una priorità più alta. Questa sequenza di eventi si ripete nel periodo 0 -  $t_6$ . Differenze rispetto a questo comportamento si verificano quando *prog<sub>cb</sub>* avvia un'operazione di I/O. Ora entrambi i programmi sono impegnati in operazioni di I/O, che procedono simultaneamente poiché i programmi utilizzano differenti dispositivi di I/O e la CPU resta idle finché uno di loro non completa la sua operazione di I/O. Questo spiega i periodi idle della CPU  $t_6 - t_7$  e  $t_8 - t_9$  nell'attività cumulativa della CPU. I periodi idle di I/O si verificano ogni volta che *prog<sub>io</sub>* utilizza la CPU e *prog<sub>cb</sub>* non sta eseguendo operazioni di I/O (vedi gli intervalli  $0 - t_1, t_2 - t_3$  e  $t_4 - t_5$ ). Ma la CPU e il sottosistema di I/O sono concorrentemente occupati negli intervalli  $t_1 - t_2, t_3 - t_4, t_5 - t_6$  e  $t_7 - t_8$ .

È possibile fare alcune considerazioni riguardanti l'Esempio 3.1. L'utilizzo della CPU è buono così come l'utilizzo dell'I/O tuttavia periodi idle dell'I/O potrebbero verificarsi se il sistema fosse dotato di molti dispositivi in grado di operare in modalità DMA. Sono frequenti i periodi di attività di CPU e I/O concorrenti. *prog<sub>io</sub>* fa molti progressi poiché è il programma a più alta priorità. Fa poco uso della CPU e in questo modo anche *prog<sub>cb</sub>* fa molti progressi. Il throughput è sostanzialmente più alto rispetto alla situazione in cui i programmi sono eseguiti uno dopo l'altro come, per esempio, in un sistema di elaborazione batch. Un'altra importante caratteristica di questa assegnazione delle priorità è che il throughput del sistema può essere migliorato aggiungendo più programmi. La Tabella 3.6 descrive come l'aggiunta di un programma CPU-bound può ridurre i periodi di inattività della CPU senza compromettere l'esecuzione degli altri programmi, mentre l'aggiunta di un programma I/O-bound può migliorare l'utilizzo ma compromettendo marginalmente l'esecuzione dei programmi CPU-bound. Il kernel può aggiungere giudiziosamente programmi CPU-bound o I/O-bound per assicurare un uso efficiente delle risorse.

| Azione                            | Effetto  |
|-----------------------------------|--|
| Aggiungere un programma CPU-bound | Un programma CPU-bound ( $prog_3$ ) può essere introdotto per utilizzare tempo di CPU non utilizzato nell'Esempio 3.1 (per esempio gli intervalli $t_6 - t_7$ e $t_8 - t_9$ ). $prog_3$ avrebbe la priorità più bassa. Dunque la presenza del nuovo programma non inficerrebbe il progresso di $prog_{cb}$ e $prog_{job}$ .  |
| Aggiungere un programma I/O-bound | Un programma I/O-bound ( $prog_4$ ) può essere introdotto. La sua priorità risulterebbe tra le priorità di $prog_{job}$ e $prog_{cb}$ . La presenza di $prog_4$ aumenterebbe l'utilizzo dell'I/O. Non inficerrebbe il progresso di $prog_{job}$ , poiché $prog_{job}$ , ha la priorità più alta, mentre ridurrebbe il progresso di $prog_{cb}$ solo marginalmente poiché $prog_4$ non utilizza una quantità significativa di tempo di CPU. |

**Tabella 3.6** Effetti dell'aumento del grado di multiprogrammazione.

Quando si mantiene un appropriato mix di programmi, ci si può aspettare che un aumento del grado di multiprogrammazione risulti in un aumento del throughput. La **Figura 3.5** mostra come varia il throughput di un sistema al variare del grado di multiprogrammazione. Quando il grado di multiprogrammazione è 1, il throughput è determinato dal tempo trascorso dall'unico programma nel sistema. Quando più programmi sono attivi nel sistema, anche i programmi a più bassa priorità contribuiscono al throughput. Tuttavia, il loro contributo è limitato dalle loro opportunità di utilizzo della CPU. Il throughput non varia al crescere del grado di multiprogrammazione se i programmi a bassa priorità non hanno possibilità di essere eseguiti.



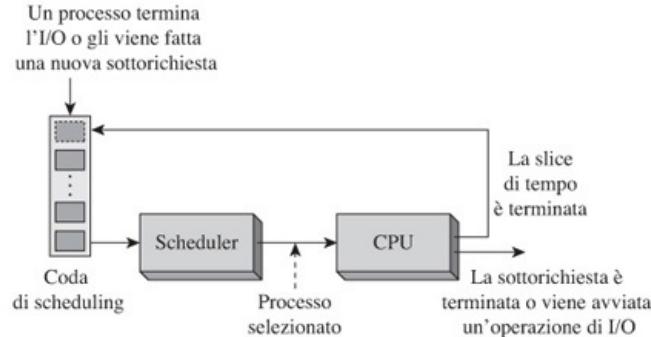
**Figura 3.5** Variazione del throughput al variare del grado di multiprogrammazione.

### 3.6 Sistemi time-sharing

In un ambiente di elaborazione interattivo, un utente sottomette una richiesta di elaborazione, o una sottorichiesta, a un processo ed esamina il risultato sul monitor. Un sistema operativo time-sharing è progettato per fornire una risposta veloce alle sottorichieste effettuate dagli utenti. L'obiettivo è ottenuto condividendo il tempo di CPU tra i processi in modo tale che ogni processo al quale è stata fatta una sottorichiesta ottenga un tempo della CPU senza attendere troppo.

La tecnica di scheduling utilizzata nei kernel time-sharing viene chiamata *scheduling round-robin con time-slicing*. Il suo funzionamento è descritto di seguito (**Figura 3.6**): il kernel mantiene una *coda di scheduling* dei processi che richiedono l'uso della CPU e schedula il processo in testa alla coda. Quando un processo schedulato termina l'esecuzione di una sottorichiesta o avvia un'operazione di I/O, il kernel lo rimuove dalla coda e schedula un altro processo. Questo processo viene aggiunto alla fine della coda quando riceve una nuova sottorichiesta o quando viene completata l'operazione di I/O. Questa organizzazione assicura che per tutti i processi si verifichino ritardi comparabili prima di ottenere l'uso della CPU. Tuttavia, il tempo di risposta dei processi

degraderebbe se un processo utilizzasse troppo tempo di CPU nel servire una sottorichiesta. Il kernel utilizza la nozione di *time slice* per evitare questa situazione, indicata d'ora in avanti come  $\delta$  e definita come segue:



**Figura 3.6** Uno schema dello scheduler round-robin con time-slice.

**Definizione 3.5 Time slice** La più grande porzione di tempo di CPU che ogni processo può utilizzare quando viene schedulato per essere eseguito dalla CPU.

Se la porzione di tempo termina prima che il processo completi l'esecuzione di una sottorichiesta, il kernel prelaziona il processo, lo sposta alla fine della coda di scheduling e seleziona un altro processo. Il processo prelazionato verrà schedulato quando raggiungerà nuovamente la testa della coda. In questo modo, un processo può dover essere schedulato diverse volte prima che completi il servizio di una sottorichiesta. Il kernel utilizza un interrupt timer per implementare il time-slicing (Paragrafo 2.2.5 e Tabella 2.2).

La misura appropriata per misurare il servizio per l'utente in un sistema time-sharing è il tempo utilizzato per eseguire una sottorichiesta, ovvero il tempo di risposta ( $rt$ ). Può essere stimato nel seguente modo: sia  $n$  il numero di utenti che stanno utilizzando il sistema in un dato momento. Sia  $\delta$  secondi il tempo di CPU necessario per il completamento della sottorichiesta di ogni utente e sia  $\sigma$  l'*overhead dello scheduling*, ovvero il tempo di CPU utilizzato dal kernel per eseguire la schedulazione. Se assumiamo che un'operazione di I/O venga completata istantaneamente e che un utente sottometta la prossima sottorichiesta immediatamente dopo aver ricevuto il risultato della precedente sottorichiesta, il tempo di risposta ( $rt$ ) e l'efficienza della CPU ( $\eta$ ) sono dati da

$$rt = n \times (\delta + \sigma) \quad (3.1)$$

$$\eta = \frac{\delta}{\delta + \sigma} \quad (3.2)$$

Il tempo di risposta attuale può essere differente dal valore di  $rt$  predetto dall'Equazione 3.1, per due motivi. Primo, tutti gli utenti possono non aver fatto sottorichieste ai loro processi. Dunque  $rt$  non dipenderebbe da  $n$ , il numero totale degli utenti nel sistema, mentre dipenderebbe dal numero di utenti attivi. Secondo, le sottorichieste degli utenti non richiedono esattamente  $\delta$  secondi di CPU per produrre un risultato. Dunque la relazione tra  $rt$  ed  $\eta$  con  $\delta$  è più complessa di quella mostrata nelle Equazioni 3.1 e 3.2.

L'Esempio 3.2 illustra lo scheduling round-robin con time-slicing e come esso risulti in un'esecuzione intrecciata dei processi.

### Esempio 3.2 Esecuzione dei processi in un sistema time-sharing

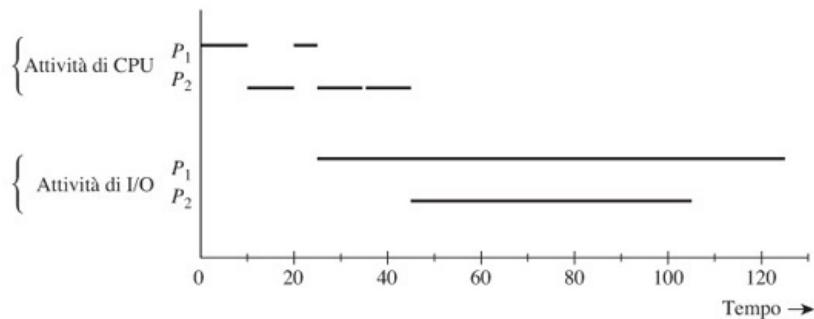
I processi  $P_1$  e  $P_2$  hanno un comportamento ciclico. Ogni ciclo contiene un burst di attività di CPU per l'esecuzione di una sottorichiesta e un burst di attività di I/O per mostrare i risultati, seguiti da una pausa finché non viene sottomessa la prossima sottorichiesta. I burst di CPU dei processi  $P_1$  e  $P_2$  sono, rispettivamente, di 15 e 30 ms,

mentre burst di I/O sono, rispettivamente, di 100 e 60 ms.

La [Figura 3.7](#) mostra l'esecuzione dei processi in un sistema time-sharing che utilizza una time-slice di 10 ms. La tabella nella parte alta della [Figura 3.7](#) mostra la lista delle schedulazioni e le decisioni dello scheduler, mentre il grafico dei tempi mostra le attività di CPU e I/O dei processi. Entrambi i processi devono essere schedulati più volte prima di completare i burst di CPU del ciclo di esecuzione e poter iniziare l'I/O. Il processo  $P_1$  utilizza la CPU dal tempo 0 fino a 10 ms e  $P_2$  utilizza la CPU da 10 a 20 ms senza completare i burst di CPU dei propri cicli di esecuzione.  $P_1$  è schedulato nuovamente a 20 ms e avvia un'operazione di I/O a 25 ms. Ora  $P_2$  ottiene due timeslice consecutive. Tuttavia, queste porzioni di tempo sono separate dall'overhead dello scheduling poiché il SO prelaziona il processo  $P_2$  a 35 ms e lo schedula di nuovo, poiché non ci sono processi che necessitano della CPU nel sistema. L'operazione di I/O di  $P_1$  termina a 125 ms.  $P_2$  avvia un'operazione di I/O a 45 ms, che termina a 105 ms. In questo modo, i tempi di risposta sono, rispettivamente, 125 ms e 105 ms.

### 3.6.1 Swapping dei programmi

Il throughput delle sottorichieste è la misura appropriata delle prestazioni di un sistema operativo time-sharing. Il SO time-sharing dell'Esempio 3.2 completa due sottorichieste in 125 ms, dunque il suo throughput è di 8 sottorichieste al secondo per il periodo 0-125 ms. Tuttavia, il throughput diminuirebbe dopo 125 ms se gli utenti non effettuassero immediatamente le successive sottorichieste a questi processi.



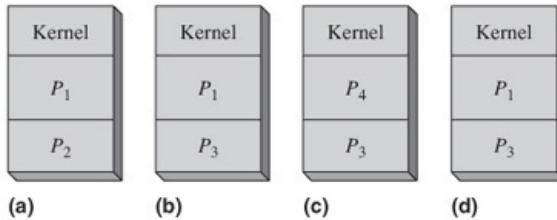
**Figura 3.7** Esecuzione dei processi  $P_1$  e  $P_2$  in un sistema time-sharing.

La CPU è idle dopo 45 ms perché non ha lavoro da svolgere. Avrebbe potuto servire qualche altra sottorichiesta, se fossero stati presenti più processi attivi nel sistema. Ma cosa succederebbe se ci fosse spazio in memoria solo per due processi? Il throughput del sistema sarebbe basso e i tempi di risposta dei processi oltre  $P_1$  e  $P_2$  ne risentirebbero. La tecnica dello *swapping* viene utilizzata per servire un maggior numero di processi rispetto a quelli che possono essere effettivamente presenti in memoria. Lo swapping ha la potenzialità di migliorare sia le prestazioni del sistema sia i tempi di risposta dei processi.

**Definizione 3.6 Swapping** La tecnica di rimuovere temporaneamente un processo dalla memoria di un computer.

Il kernel effettua un'operazione *swap-out* su un processo che verosimilmente non sarà schedulato nel prossimo futuro, copiando le sue istruzioni e i dati sul disco. Questa operazione libera l'area di memoria che era allocata al processo. Il kernel ora carica un altro processo in questa area di memoria attraverso un'operazione *swap-in*. Il kernel sovrappone le operazioni di *swap-out* e *swap-in* con l'esecuzione di altri processi da parte della CPU e, nel frattempo, un processo riportato in memoria viene schedulato. In questo modo, il kernel può eseguire più processi di quelli allocabili in memoria. La [Figura 3.8](#) illustra come il kernel utilizza lo swapping. Inizialmente, i processi  $P_1$  e  $P_2$  sono presenti in memoria. Questi processi sono swappati sul disco quando completano la gestione delle

loro sottorichieste e sono sostituiti, rispettivamente, dai processi  $P_3$  e  $P_4$ , che a loro volta potrebbero essere stati swappati in precedenza. Un processo swappato sul disco viene riportato in memoria prima di essere nuovamente schedulato, cioè quando si avvicina alla testa della coda di schedulazione in [Figura 3.6](#).



**Figura 3.8** Swapping: (a) i processi in memoria tra 0 e 105 ms; (b)  $P_2$  è sostituito da  $P_3$  a 105 ms; (c)  $P_1$  è sostituito da  $P_4$  a 125 ms; (d)  $P_1$  viene swappato per servire la successiva sottorichiesta.

### 3.7 Sistemi operativi real-time

In una classe di applicazioni chiamate *applicazioni real-time*, gli utenti hanno bisogno del computer per effettuare azioni scandite dal tempo in modo da controllare le attività in un sistema esterno o partecipare a esse. La temporizzazione delle azioni è determinata dai vincoli temporali del sistema esterno. Di conseguenza, definiamo un'applicazione real-time come segue:

**Definizione 3.7 Applicazione real-time** Un programma che risponde alle attività in un sistema esterno entro un intervallo di tempo massimo determinato dal sistema esterno.

Se l'applicazione impiega troppo tempo per rispondere a una attività, si verifica un malfunzionamento del sistema esterno. Usiamo il termine *requisito di risposta* di un sistema per indicare il più grande valore di tempo di risposta per il quale il sistema può funzionare correttamente; un risposta in tempo è quella il cui tempo di risposta non è maggiore del requisito di risposta del sistema.

Si consideri un sistema che registra i dati ricevuti da un sensore remoto posto su un satellite. Il satellite invia campioni digitalizzati alla stazione terrestre a un tasso di 500 campioni al secondo. Si richiede che il processo applicativo memorizzi semplicemente questi campioni in un file. Poiché arriva un nuovo campione ogni due millesimi di secondo, ovvero 2 ms, il computer deve rispondere a ogni richiesta di "memorizzare il campione" in meno di 2 ms, in caso contrario l'arrivo di un nuovo campione cancellerebbe il precedente campione dalla memoria del computer. Tale applicazione real-time poiché un campione deve essere memorizzato in meno di 2 ms per prevenire un malfunzionamento. Il suo requisito di risposta è 1.99 ms. La *deadline* di un'azione in un'applicazione real-time è il tempo entro il quale l'azione dovrebbe essere effettuata. Nell'esempio precedente, se un nuovo campione è ricevuto dal satellite al tempo  $t$ , la deadline per memorizzarlo sul disco è di  $t + 1.99$  ms. Altri esempi di applicazioni real-time possono essere ritrovati nella guida dei missili, nelle applicazioni di comando e controllo come il controllo dei processi e del traffico aereo, nel campionamento dei dati e nei sistemi di acquisizione dei dati come i sistemi di visione nelle automobili, nei sistemi multimediali e nelle applicazioni come prenotazioni o sistemi bancari che utilizzano grandi database. I requisiti di risposta di questi sistemi variano da pochi microsecondi o millisecondi per i sistemi di controllo e di guida a pochi secondi per i sistemi di prenotazione e quelli bancari.

#### 3.7.1 Sistemi hard e soft real-time

Per sfruttare al massimo le caratteristiche dei sistemi real-time minimizzando i costi, sono stati sviluppati due tipi di sistemi real-time. Un *sistema hard real-time* è tipicamente *dedicato* a elaborare applicazioni real-time e soddisfa il requisito di risposta di

un'applicazione real-time in ogni condizione. Un *sistema soft real-time* fa del suo meglio per soddisfare il requisito di risposta di un'applicazione real-time ma non può garantire che lo farà in ogni condizione. Tipicamente, soddisfa i requisiti di risposta in maniera probabilistica, per esempio, il 98% delle volte. Le applicazioni di guida e di controllo falliscono se non soddisfano il requisito di risposta, e pertanto sono serviti da sistemi hard real-time. Le applicazioni che puntano a fornire una buona qualità del servizio (QoS), come per esempio le applicazioni multimediali, e le applicazioni per le prenotazioni e bancarie, non comportano risultati di insuccesso e dunque possono essere gestite da sistemi soft real-time (la qualità dell'immagine fornita da un sistema di video-on-demand può occasionalmente peggiorare, ma il video può ancora essere fruibile!).

### 3.7.2 Caratteristiche di un sistema operativo real-time

Le caratteristiche di un SO real-time sono riassunte nella [Tabella 3.7](#). Le prime tre caratteristiche consentono a un'applicazione di soddisfare il vincolo di risposta di un sistema come segue: un'applicazione real-time può essere progettata in modo tale che il SO possa eseguire le sue parti in maniera concorrente, ovvero come processi separati. Quando a queste parti sono assegnate delle priorità e viene impiegato uno scheduling basato su priorità, si verifica una situazione analoga alla multiprogrammazione *all'interno* dell'applicazione; se un'applicazione avvia un'operazione di I/O, il SO schedulerebbe un'altra applicazione. In questo modo, le attività di CPU e di I/O dell'applicazione possono essere sovrapposte l'una all'altra, cosa che contribuisce a ridurre il tempo di esecuzione delle applicazioni. La *schedulazione basata su deadline* è una tecnica usata nel kernel che schedula i processi in maniera tale che possano rispettare le rispettive deadline.

| Caratteristica                             | Spiegazione  |
|--|--|
| Concorrenza all'interno di un'applicazione | Un programmatore può indicare che alcune parti di un'applicazione debbano essere eseguite in maniera concorrente l'una con l'altra. Il SO considera l'esecuzione di ognuna di queste parti come un processo. |
| Priorità del processo                      | Un programmatore può assegnare delle priorità al processo.   |
| Scheduling                                 | Il SO utilizza politiche di scheduling basate su priorità o su deadline.   |
| Eventi specifici per il dominio, interrupt | Un programmatore può definire speciali situazioni nel sistema esterno come eventi, associare a essi degli interrupt e specificare azioni per la gestione di questi eventi.                                   |
| Predicibilità                              | Le politiche e l'overhead del SO dovrebbero essere predibili.  |
| Affidabilità                               | Il SO garantisce che un'applicazione possa continuare a funzionare anche quando si verificano malfunzionamenti nel computer.   |

**Tabella 3.7** Caratteristiche essenziali di un sistema operativo real-time.

La possibilità di specificare *eventi specifici per il dominio* e azioni per la gestione degli eventi consente a un'applicazione real-time di rispondere prontamente a speciali condizioni del sistema esterno. La *predicibilità* delle politiche e dell'overhead del SO consente a uno sviluppatore di calcolare il tempo di esecuzione di un'applicazione nel caso peggiore e decidere se il vincolo di risposta del sistema esterno possa essere soddisfatto. Il requisito di *predicibilità* vincola un SO hard real-time a non utilizzare caratteristiche quali la *memoria virtuale* le cui prestazioni non possono essere predette in maniera precisa ([Capitolo 12](#)). Il SO eviterebbe anche l'uso delle risorse condivise tra i processi, poiché questo può portare a ritardi che sono difficili da predire e non vincolati, ovvero arbitrariamente lunghi.

Un SO real-time adotta due tecniche per assicurare la continuità del funzionamento quando si verifica un malfunzionamento: *fault tolerance* e *graceful degradation*. Un computer fault-tolerant utilizza risorse ridondanti per assicurare che il sistema continui a funzionare anche in caso di malfunzionamento; per esempio, può avere due dischi sebbene l'applicazione necessiti di un solo disco. La *graceful degradation* è l'abilità di un sistema di passare a un ridotto livello di servizio quando si verifica un malfunzionamento e di tornare al funzionamento normale quando il problema viene risolto. Il

programmatore può assegnare priorità alte alle funzioni cruciali così che possano essere eseguite in maniera cadenzata anche quando il sistema opera in modalità ridotta.

### 3.8 Sistemi operativi distribuiti

Un sistema operativo distribuito si compone di diversi computer singoli connessi attraverso una rete. Ogni computer potrebbe essere un PC, un sistema multiprocessore ([Capitolo 10](#)), o un *cluster*, che a sua volta consiste in un gruppo di computer che lavorano insieme in maniera integrata ([Paragrafo 16.2](#)). In questo modo in un sistema distribuito esistono molte risorse di un determinato tipo, per esempio, molte memorie, CPU e dispositivi di I/O. Un sistema operativo distribuito sfrutta la molteplicità delle risorse e la presenza di una rete per fornire i benefici elencati nella [Tabella 3.8](#). Tuttavia, la possibilità di malfunzionamenti di rete o di singoli computer complica il funzionamento del sistema operativo e necessita di speciali tecniche di implementazione. Anche gli utenti hanno bisogno di utilizzare tecniche speciali per accedere alle risorse in rete. Questi aspetti verranno discussi nel [Paragrafo 3.8.1](#).

| Beneficio                                 | Descrizione  |
|---|--|
| Condivisione delle risorse                | Le risorse possono essere utilizzate attraverso i limiti dei singoli sistemi.                                  |
| Affidabilità                              | Il SO continua a funzionare anche quando i computer o le risorse in esso contenute non funzionano.             |
| Incremento della velocità di elaborazione | I processi di un'applicazione possono essere eseguiti su differenti sistemi per velocizzarne il completamento. |
| Comunicazione                             | Gli utenti possono comunicare tra di loro a prescindere dalle loro posizioni nel sistema.                      |

**Tabella 3.8** Benefici dei sistemi operativi distribuiti.

La *condivisione delle risorse* è stata la motivazione per lo sviluppo dei sistemi distribuiti. Un utente di un PC o di una workstation può usare le risorse come stampanti su di una rete locale (LAN), e accedere ad hardware specializzato o risorse software di un sistema geograficamente distante attraverso una wide area network (WAN).

Un sistema operativo distribuito fornisce *affidabilità* attraverso la ridondanza dei sistemi, delle risorse e dei percorsi di comunicazione; se un computer o una risorsa usata in un'applicazione si guasta, il SO può commutare l'applicazione a un altro computer o a un'altra risorsa e se il percorso per una risorsa è guasto, può utilizzare un altro percorso per quella risorsa. L'affidabilità può essere usata per offrire un'elevata *disponibilità* delle risorse e dei servizi, e può essere calcolata come la frazione di tempo in cui la risorsa o il servizio è disponibile. L'elevata disponibilità di una risorsa di dati, per esempio un file, può essere ottenuta conservando copie del file in varie parti del sistema.

L'*incremento della velocità di elaborazione* implica una riduzione della durata delle applicazioni, cioè del tempo di esecuzione. Questo obiettivo è ottenuto allocando i processi di un'applicazione su diversi computer nel sistema distribuito, cosicché possano essere eseguiti allo stesso tempo e terminare in un tempo inferiore rispetto a quello impiegato se fossero eseguiti da un SO convenzionale.

Gli utenti di un sistema operativo distribuito hanno degli identificativi utente e password che sono validi in tutto il sistema. Questa caratteristica facilita notevolmente la *comunicazione* tra gli utenti in due modi. Primo, la comunicazione mediante l'id utente invoca automaticamente i meccanismi di sicurezza del SO assicurando in questo modo l'autenticità della comunicazione. Secondo, gli utenti possono muoversi all'interno del sistema distribuito ed essere ancora in grado di comunicare con altri utenti attraverso il sistema.

#### 3.8.1 Tecniche speciali di sistemi operativi distribuiti

Un sistema distribuito è più di una semplice collezione di computer connessi a una rete; il funzionamento dei singoli computer deve esser integrato per ottenere i benefici elencati nella [Tabella 3.8](#). Questo obiettivo è ottenuto attraverso la partecipazione di tutti

i computer alle funzioni di controllo del sistema operativo. Di conseguenza, possiamo definire un sistema distribuito come segue:

**Definizione 3.8 Sistema distribuito** Un sistema composto di due o più nodi, in cui ogni nodo è un computer con un proprio clock e una propria memoria, dell'hardware di rete con la capacità di effettuare alcune funzioni di controllo di un SO.

La [Tabella 3.9](#) riassume tre concetti e tecniche utilizzate in un SO distribuito. Il *controllo distribuito* è l'opposto del controllo centralizzato; implica che le funzioni di controllo del sistema distribuito siano effettuate da diversi computer nel sistema secondo la Definizione 3.8, invece di essere effettuate da un singolo computer. Il controllo distribuito è essenziale per assicurare che il malfunzionamento di un singolo computer, o di un gruppo di computer, non arresti l'esecuzione dell'intero sistema. La *trasparenza* di una risorsa o di un servizio implica che un utente dovrebbe essere in grado di accedervi senza dover conoscere quale nodo nel sistema distribuito la contiene. Questa caratteristica consente al SO di cambiare la posizione di una risorsa software o di un servizio per ottimizzare il suo utilizzo da parte delle applicazioni. Per esempio, in un sistema che fornisce la trasparenza, un file system distribuito potrebbe spostare un file in un nodo in cui è in esecuzione un processo che necessita di questo file, in modo da eliminare i ritardi coinvolti nell'accesso al file attraverso la rete. La *remote procedure call* (RPC) richiama una procedura che viene eseguita in un altro computer del sistema distribuito. Un'applicazione può utilizzare la caratteristica della RPC sia per effettuare una parte della sua elaborazione su un altro computer, il che contribuirebbe ad aumentare la velocità dell'elaborazione, sia per accedere a una risorsa presente in quel computer.

| Concetto/tecnica            | Descrizione  |
|-----------------------------|--|
| Controllo distribuito       | Una funzione di controllo viene effettuata attraverso la collaborazione di diversi nodi, possibilmente <i>tutti</i> i nodi, in un sistema distribuito.   |
| Trasparenza                 | Si può accedere a una risorsa o a un servizio senza dover conoscere la sua posizione nel sistema distribuito.  |
| Remote procedure call (RPC) | Un processo chiama una procedura che è messa a disposizione da un computer remoto. La RPC è analoga a una procedura o una chiamata di funzione in un linguaggio di programmazione, fatta eccezione per il fatto che il SO passa i parametri alla procedura remota attraverso la rete e restituisce i risultati attraverso la rete. |

**Tabella 3.9** Concetti chiave e tecniche usate in un SO distribuito.

### 3.9 Moderni sistemi operativi

Gli utenti sono coinvolti in diverse attività in un moderno ambiente di elaborazione. Dunque, un moderno sistema operativo non può usare una strategia uniforme per tutti i processi; deve usare una strategia appropriata per ogni singolo processo. Per esempio, come menzionato nel Paragrafo 3.1, un utente può aprire un programma per leggere le e-mail, editare un file, eseguire alcuni programmi, includendo alcuni programmi in modalità background e allo stesso tempo guardare un video. In questo caso, il funzionamento di alcuni programmi può essere interattivo o può coinvolgere attività su altri nodi di un sistema distribuito, dove il rendering di un video è un'attività soft real-time.

Pertanto il SO deve usare uno scheduling round-robin per l'esecuzione dei programmi, uno scheduling a priorità per i processi dell'applicazione video e implementare chiamate a procedura remota (RPC) per supportare le attività in un altro nodo. In questo modo, un moderno SO usa molti dei concetti e delle tecniche di cui si è discusso relativamente all'elaborazione batch e ai sistemi operativi multiprogrammati, time-sharing, real-time e distribuiti. La [Tabella 3.10](#) mostra un tipico esempio di come i concetti precedenti siano stati raggruppati.

| Concetto   | Tipico esempio d'uso   |
|--|--|
| Elaborazione batch                                 | Evita perdite di tempo dovute all'inizializzazione per ogni utilizzo di una risorsa; per esempio, le transazioni su database sono elaborate in modalità batch nelle elaborazioni di ufficio e le elaborazioni scientifiche sono eseguite in modalità batch nelle organizzazioni di ricerca e nei laboratori clinici. |
| Scheduling a priorità con prelazione               | Favoriscono le applicazioni ad alta priorità e consentono un uso efficiente delle risorse assegnando alte priorità ai processi interattivi e basse priorità ai processi non interattivi.   |
| Time-slicing                                       | Per prevenire che un processo monopolizzi la CPU; aiuta a fornire buoni tempi di risposta.   |
| Swapping   | Incrementa il numero di processi che possono essere serviti simultaneamente; consente di migliorare le prestazioni del sistema e i tempi di risposta dei processi.   |
| Creazione di processi multipli in una applicazione | Permette di ridurre la durata di un'applicazione; è più efficace quando l'applicazione contiene sostanziali attività di CPU e I/O.   |
| Condivisione delle risorse                         | Consente di condividere risorse come stampanti laser o servizi come file server in un ambiente LAN.  |

**Tabella 3.10** Utilizzo dei concetti classici dei SO nei moderni ambienti di elaborazione.

Per gestire diverse attività in maniera efficace, il SO utilizza le strategie che meglio si adattano alle situazioni riscontrate durante la loro esecuzione. Alcuni esempi di tali strategie sono:

- il kernel utilizza scheduling a priorità; tuttavia, invece di assegnare priorità fisse a tutti i processi in un sistema multiprogrammato, assegna elevate priorità fisse solo ai processi con vincoli real-time e modifica le priorità degli altri processi per adattarsi al loro comportamento recente: aumenta la priorità di un processo se di recente è stato coinvolto in una interazione o in un'operazione di I/O e riduce la sua priorità in caso contrario;
- un moderno SO tipicamente utilizza la tecnica denominata *memoria virtuale*, secondo la quale solo alcune parti dei processi sono mantenute in memoria a ogni istante di tempo mentre altre parti sono caricate quando necessarie. Il kernel osserva il comportamento di un processo per decidere quanta memoria deve allocare al processo; alloca meno memoria se il processo ha usato solo poche parti di recente e alloca più memoria se il processo ha utilizzato molte delle sue parti;
- il kernel fornisce una funzione di *plug-and-play* con la quale i dispositivi di I/O possono essere connessi al computer durante il suo funzionamento e seleziona il metodo appropriato per gestirli.

Vedremo diverse istanze di strategie adattive nei capitoli successivi.

## Riepilogo

Un *ambiente di elaborazione* si compone di un computer, di interfacce con altri sistemi e dei servizi forniti dal suo sistema operativo agli utenti e ai programmi. Gli ambienti di elaborazione si sono evoluti con i progressi della tecnologia dei computer e delle applicazioni. Ogni ambiente è progettato per fornire una differente combinazione di uso efficiente e servizio per l'utente, per cui è adottato da differenti sistemi operativi. In questo capitolo, abbiamo affrontato i concetti e le tecniche utilizzate nei fondamentali sistemi operativi.

I sistemi operativi per l'elaborazione batch si focalizzano sul processo di automatizzazione di una collezione di programmi, in modo da ridurre i tempi idle della CPU. Lo sviluppo della tecnologia direct memory access (DMA) consente alla CPU di eseguire istruzioni mentre è in esecuzione un'operazione di I/O. I sistemi operativi sfruttano questa caratteristica per fornire servizio a diversi programmi simultaneamente sovrapponendo un'operazione di I/O in un programma con

l'esecuzione di istruzioni in un altro programma. Un sistema operativo multiprogrammato assegna alte *priorità* ai programmi I/O-bound ed effettua lo *scheduling a priorità* per ottenere buone prestazioni del sistema.

La convenienza per l'utente divenne importante quando il costo dell'hardware iniziò a diminuire. Di conseguenza, i sistemi operativi time-sharing tendevano a fornire tempi di risposta veloci ai programmi degli utenti. Questo obiettivo fu ottenuto mediante l'uso dello scheduling *round-robin* con *time-slicing*, che serviva tutti i programmi a turno e limitava la quantità di tempo di CPU che un programma poteva utilizzare.

Un'applicazione real-time deve soddisfare vincoli temporali imposti da un sistema esterno. I *sistemi hard real-time*, come per esempio i sistemi per il controllo della missione, richiedono che i loro vincoli temporali siano rispettati in maniera garantita, invece i *sistemi soft real-time*, come per esempio i sistemi multimediali, possono tollerare che in modo occasionale non vengano soddisfatti i loro vincoli temporali. I sistemi operativi real-time supportano la concorrenza *all'interno* di un programma e utilizzano tecniche come gli *scheduling a priorità* e gli *scheduling basati su deadline* per consentire il rispetto dei vincoli temporali.

Un sistema operativo distribuito controlla un gruppo di sistemi connessi attraverso una rete; il SO esegue le sue funzioni di controllo in diversi di questi computer. Ciò consente l'uso efficiente delle risorse di tutti i computer consentendo ai programmi di condividerle attraverso la rete, l'aumento della velocità di esecuzione di un programma eseguendo le sue parti su differenti computer allo stesso tempo e di fornire affidabilità attraverso la ridondanza delle risorse e dei servizi.

Un moderno sistema operativo controlla un diverso ambiente di elaborazione che ha elementi di tutti i classici ambienti di elaborazione, e deve pertanto utilizzare tecniche differenti per differenti applicazioni. Utilizza una strategia adattiva che seleziona le tecniche più appropriate per ogni applicazione in base alla sua natura.

## Domande

3.1. I programmi A, B, C e D hanno una struttura simile, ognuno consiste di un singolo ciclo composto da  $n$  istruzioni che effettuano alcune elaborazioni su ogni elemento di un array monodimensionale Z. Altre caratteristiche di questi programmi sono le seguenti:

- Il programma A:  $n = 4$  e Z è un grande array.
- Il programma B:  $n = 100$  e Z è un grande array.
- Il programma C:  $n = 4$  e Z è un piccolo array.
- Il programma D:  $n = 100$  e Z è un grande array.

Questi programmi sono eseguiti in un sistema di elaborazione batch. Elencare questi programmi in ordine decrescente di hit ratio della cache.

3.2. Un sistema multiprogrammato viene utilizzato per eseguire un insieme di programmi C. Il sistema ha abbastanza memoria per memorizzare un grande numero di programmi. I programmi in C sono eseguiti diverse volte, ogni volta con un differente grado di multiprogrammazione. Il throughput del sistema e l'efficienza della CPU sono plottati rispetto al grado di multiprogrammazione. In ognuno dei seguenti casi, quale conclusione può essere tratta in base alla natura dei programmi in C?

- a.** Il throughput cambia solo marginalmente con il grado di multiprogrammazione.
- b.** Il throughput aumenta quasi linearmente con il grado di multiprogrammazione.
- c.** L'efficienza della CPU cambia solo marginalmente con il grado di multiprogrammazione.
- d.** L'efficienza della CPU aumenta linearmente con il grado di multiprogrammazione.

3.3. Classificare ognuna delle seguenti affermazioni come vere o false.

- a.** A causa della presenza della memoria cache, un programma impiega più tempo di CPU per l'esecuzione in un sistema multiprogrammato o time-sharing di quanto ne richiederebbe se fosse eseguito in un sistema batch.
- b.** Per ottenere un maggior throughput, un SO multiprogrammato assegna priorità maggiore ai programmi CPU-bound.
- c.** Se un kernel multiprogrammato scopre che l'efficienza della CPU è bassa, dovrebbe rimuovere dalla memoria un programma I/O-bound.
- d.** Se la time-slice in un sistema timesharing è troppo lunga, i processi termineranno

- la loro esecuzione nello stesso ordine in cui erano state avviate.
- e. Due persone che utilizzano lo stesso sistema time-sharing allo stesso momento potrebbero ricevere tempi di risposta molto differenti.
- f. Non è corretto utilizzare il mascheramento degli interrupt in un sistema operativo real-time.

## Problemi

- 3.1. Un sistema è descritto come sovraccarico se a esso è diretto più lavoro di quanto ne possa svolgere. È considerato non sfruttato a pieno se parte della sua capacità viene sprecata. La seguente politica è proposta per migliorare il throughput di un sistema di elaborazione batch: a) classificare i job in brevi e lunghi in considerazione del tempo di CPU richiesto; b) creare batch separati di job brevi e lunghi, c) eseguire un batch di job lunghi solo se non ci sono batch di job brevi. Questa politica migliora il throughput di un sistema di elaborazione batch che sia: (a) sovraccarico? (b) non sfruttato a pieno?
- 3.2. Il kernel di un sistema multiprogrammato classifica un programma come CPU-bound o I/O-bound e gli assegna una priorità appropriata. Quale sarebbe la conseguenza di una errata classificazione dei programmi per i tempi di throughput e turnaround in un sistema multiprogrammato? Quale sarebbe l'effetto di una errata classificazione nel plot del throughput *rispetto* al grado di multiprogrammazione di [Figura 3.5](#)?
- 3.3. La CPU di un sistema multiprogrammato sta eseguendo un programma ad alta priorità quando viene generato un interrupt che segnala il completamento di un'operazione di I/O. Mostrare tutte le azioni e le attività nel SO che si verificano successivamente all'interrupt se
- l'operazione di I/O è stata lanciata da un programma a bassa priorità;
  - l'operazione di I/O è stata lanciata da un programma ad alta priorità.
- Illustrare ogni caso con l'aiuto di un grafico dei tempi.
- 3.4. Un SO multiprogrammato ha i programmi  $prog_{iob}$  e  $prog_{cb}$  in memoria, con  $prog_{cb}$ , ad alta priorità. Disegnare un grafico dei tempi per il sistema analogo a quello di [Figura 3.4](#) e mostrare che il throughput è minore di quello per il sistema di [Figura 3.4](#).
- 3.5. Disegnare un grafico dei tempi per un sistema in cui sono in esecuzione due programmi CPU-bound e 2 programmi I/O-bound nel caso in cui la priorità più alta venga assegnata (a) ai programmi CPU-bound, (b) ai programmi I/O-bound.
- 3.6. Un programma si compone di un singolo ciclo che viene eseguito 50 volte. Il ciclo contiene una elaborazione che impiega 50 ms di tempo di CPU, seguita da una operazione di I/O che dura 200 ms. Il programma viene eseguito in un SO multiprogrammato con overhead trascurabile. Realizzare un grafico dei tempi mostrando le attività di CPU e I/O del programma e calcolare il tempo trascorso nei seguenti casi:
- il programma ha la priorità più alta nel sistema;
  - il programma è suddiviso in  $n$  sottoprogrammi con identiche caratteristiche e ha la più bassa priorità. Considerare i casi (i)  $n = 3$ , (ii)  $n = 4$  e (iii)  $n = 5$ .
- 3.7. Un sistema operativo multiprogrammato ha un overhead trascurabile e serve programmi con la medesima dimensione. Ogni programma contiene un ciclo ripetuto  $n$  volte e ogni iterazione esegue un'elaborazione che necessita di  $t_c$  ms di tempo di CPU, seguita da un'operazione di I/O che richiede  $t_{io}$  ms. I programmi si dividono in due classi; i valori di  $n$ ,  $t_c$  e  $t_{io}$  per queste due classi sono i seguenti:

| Classe | $n$ | $t_c$ | $t_{io}$ |
|--------|-----|-------|----------|
| A      | 5   | 15    | 100      |
| B      | 6   | 200   | 80       |

Il sistema ha memoria sufficiente per contenere solo due programmi. Al tempo 0 arrivano nel sistema 10 programmi, cinque per ogni classe A e B. Disegnare il grafico dei tempi che mostri l'esecuzione dei programmi nel sistema finché due programmi non completano la loro esecuzione e calcolare i loro tempi di turnaround.

- 3.8. Si dice che un programma "fa progressi" se la CPU sta eseguendo le sue istruzioni o

- se è in esecuzione una sua operazione di I/O. Il *coefficiente di progresso* di un programma è la frazione del suo tempo di presenza nel sistema durante il quale fa progressi. Calcolare i coefficienti di progresso dei programmi del Problema 3.6(b).
- 3.9. Commentare la correttezza della seguente affermazione: "Un programma CPU-bound ha sempre un coefficiente di progresso molto basso in un sistema multiprogrammato".
- 3.10. Un sistema multiprogrammato utilizza un grado di multiprogrammazione ( $m$ )  $>> 1$ . Viene proposto di raddoppiare il throughput del sistema migliorando/sostituendo le sue componenti hardware. Il risultato desiderato verrebbe raggiunto con qualcuna delle seguenti proposte?
- Sostituire la CPU con una che abbia velocità doppia rispetto alla prima.
  - Raddoppiare la quantità di memoria.
  - Sostituire la CPU con una che abbia velocità doppia rispetto alla prima e raddoppiare la quantità di memoria.
- 3.11. Alcuni programmi in esecuzione in un sistema multiprogrammato sono chiamati  $P_1, \dots, P_m$ , dove  $m$  è il grado di multiprogrammazione, tale che la priorità del programma  $P_i, >$  della priorità del programma  $P_{i+1}$ . Tutti i programmi hanno natura ciclica e ogni ciclo contiene una attività di burst e un'operazione di I/O. Siano  $b_{cpu}^i$  e  $b_{io}^i$  i burst di CPU e I/O del programma  $P_i$ . Commentare la validità di ognuna delle seguenti operazioni.
- La CPU è idle se  $b_{io}^h > \sum_{j \neq h} (b_{cpu}^j)$ , dove  $P_h$  è il programma a più alta priorità.
  - Al programma  $P_m$  è garantito l'uso della CPU se  $b_{io}^i < (b_{cpu}^{i+1} + b_{io}^{i+1})$  e  $b_{io}^i > \sum_{j=i+1 \dots m} (b_{cpu}^j) \forall i = 1, \dots, m - 1$ .
- 3.12. Si dice che un programma è in una condizione di *starvation* se non ottiene mai l'uso della CPU. Quale delle seguenti condizioni implica starvation del programma a più bassa priorità in un sistema multiprogrammato? (La notazione è la stessa del Problema 3.11.)
- Per qualche programma  $P_i$ ,  $b_{io}^i < \sum_{j=i+1 \dots m} (b_{cpu}^j)$ .
  - Per qualche programma  $P_i$ ,  $b_{io}^i < \sum_{j=i+1 \dots m} (b_{cpu}^j)$  e  $b_{cpu}^i > b_{io}^i$  per ogni  $j > i$ .
- 3.13. Un sistema time-sharing contiene  $n$  processi identici, ognuno dei quali esegue un ciclo che contiene un'elaborazione di  $t_p$  secondi di CPU e un'operazione di  $t_{io}$  secondi. Disegnare un grafico che mostri le variazioni del tempo di risposta al variare della time slice. (Suggerimento: Considerare i casi  $t_p < \delta$ ,  $\delta < t_p < 2 \times \delta$  e  $t_p > 2 \times \delta$ .)
- 3.14. Commentare la correttezza della seguente affermazione: "Il funzionamento di un sistema time-sharing è identico al funzionamento di un sistema multiprogrammato che esegue gli stessi programmi se  $\delta$  è maggiore del CPU burst di ogni programma."
- 3.15. Rispondere alle seguenti domande argomentando le risposte:
- Lo swapping aumenta o degrada l'efficienza dell'utilizzo del sistema?
  - Lo swapping può essere usato in un sistema multiprogrammato?
- 3.16. Un computer esegue un SO time-sharing. Viene proposto di aggiungere una seconda CPU al computer per migliorare il suo throughput. A quali condizioni, l'aggiunta della seconda CPU al computer migliorerebbe il throughput se la memoria fosse aumentata? A quali condizioni migliorerebbe il throughput se la memoria non fosse aumentata?
- 3.17. Un sistema time-sharing utilizza lo swapping come tecnica fondamentale per la gestione della memoria. Utilizza le seguenti liste per gestire le sue azioni: una lista di scheduling, una lista di swap-out contenente tutti i processi swapped sul disco, una lista di processi da swappare sul disco e una lista di processi da riportare in memoria. Spiegare quando e perché il kernel time-sharing dovrebbe inserire un nuovo elemento nelle liste dei processi da swappare sul disco o da riportare in memoria.
- 3.18. Un sistema time-sharing utilizza una time slice di 100 ms. Ogni processo ha un comportamento ciclico. Ogni ciclo richiede in media 50 ms di tempo di CPU per calcolare il risultato di una sottorichiesta e 150 ms per mostrare il risultato sul monitor. Un processo riceve una sottorichiesta un secondo dopo che ha terminato la visualizzazione dei risultati della precedente sottorichiesta. Il sistema operativo può

- mantenere in memoria 10 processi; tuttavia, ha a disposizione un numero di dispositivi di I/O per 25 processi. I tempi di swap-in e swap-out per ogni processo sono  $t_s$  ms ognuno. Calcolare il throughput medio del sistema in un periodo di 10 secondi per ognuno dei seguenti casi.
- Nel sistema sono attivi 10 processi.
  - Nel sistema sono attivi 20 processi e  $t_s = 750$  ms.
  - Nel sistema sono attivi 20 processi e  $t_s = 250$  ms.
- 3.19. Un'applicazione real-time richiede un tempo di risposta di 2 secondi. Discutere la fattibilità dell'uso di un sistema time-sharing per l'applicazione real-time se il tempo di risposta medio nel sistema time-sharing è di (a) 20 secondi, (b) 2 secondi o (c) 0.2 secondi.
- 3.20. Un sistema time-sharing serve  $n$  processi. Utilizza una time slice di  $\delta$  secondi di CPU e richiede  $t_s$  secondi di CPU per la commutazione dei processi. Un'applicazione real-time richiede  $t_c$  secondi di tempo di CPU, seguiti da un'operazione di I/O che dura  $t_{io}$  secondi. Qual è il più grande valore di  $\delta$  per il quale il sistema time-sharing può soddisfare le richieste di risposta dell'applicazione real-time?
- 3.21. Un programma è stato sviluppato per un controller multiprocessore di un'automobile. Si richiede che l'applicazione esegua le seguenti funzioni.
- Monitorare e mostrare la velocità dell'automobile.
  - Monitorare il livello di carburante e generare un allarme se necessario.
  - Mostrare l'efficienza del carburante, ovvero Km/l alla velocità attuale.
  - Monitorare la condizione del motore e generare un allarme se si verifica una condizione anomala.
  - Registrare periodicamente alcune informazioni ausiliarie come la velocità e il livello del carburante (ovvero implementa una "black box" come negli aerei).
- Rispondere alle seguenti domande riguardanti l'applicazione.
- Si tratta di un'applicazione real-time? Giustificare la risposta.
  - La creazione di processi multipli ridurrebbe il tempo di risposta dell'applicazione? In questo caso quali dovrebbero essere i processi? Quali proprietà dovrebbero possedere?
  - È necessario definire eventi e interrupt per uno specifico dominio? In caso affermativo, specificare le loro proprietà.
- 3.22. Se due eventi indipendenti  $e_1$  e  $e_2$  hanno probabilità di verificarsi  $pr_1$  e  $pr_2$ , dove entrambe  $pr_1$  e  $pr_2 < 1$ , la probabilità che entrambi gli eventi si verifichino allo stesso tempo  $pr_1 \times pr_2$ . Un sistema distribuito è dotato di due dischi. Si supponga che la probabilità che entrambi i dischi abbiano un malfunzionamento sia  $< 0.0001$ . Quale dovrebbe essere la probabilità di malfunzionamento di un disco?
- 3.23. Per ottenere lo speedup in un sistema distribuito, un'applicazione è codificata in tre parti da eseguire su tre sistemi diversi sotto il controllo di un sistema operativo distribuito. Tuttavia, lo speedup ottenuto è  $< 3$ . Elencare tutte le possibili ragioni che hanno causato lo scarso speedup.

## Note bibliografiche

La letteratura riguardante l'elaborazione batch, la multiprogrammazione e i sistemi timesharing risale agli anni '70. Zhao (1989) e Liu (2000) rappresentano buone fonti per i sistemi real-time. Molti testi di sistemi operativi affrontano le classi di sistemi operativi descritte in questo capitolo; alcuni testi recenti sono Tanenbaum (2001), Bic e Shaw (2003), Nutt (2004), Silberschatz et al. (2005) e Stallings (2005). Diverse bibliografie riguardanti i sistemi operativi sono disponibili in Internet.

Tanenbaum e Renesse (1985) è un buon punto di partenza per lo studio dei sistemi operativi. Affronta le questioni più importanti riguardanti i sistemi operativi distribuiti e contiene una disamina di alcuni sistemi operativi distribuiti. Tanenbaum (1995) affronta nel dettaglio alcuni famosi sistemi operativi distribuiti. Coulouris et al. (2001) affronta i concetti e la progettazione dei sistemi distribuiti.

Diversi libri descrivono specifici sistemi operativi moderni. Bach (1986) e Vahalia (1996) descrivono il sistema operativo Unix. Beck et al. (2002), Bovet e Cesati (2005) e Love (2005) esaminano il sistema operativo Linux, mentre Stevens e Rago (2005)

descrivono i sistemi operativi Unix, Linux e BSD. Mauro e McDougall (2006) discutono di Solaris. Russinovich e Solomon (2005) descrivono i sistemi operativi Windows.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
2. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
3. Bic, L., and A.C. Shaw (2003): *Operating Systems Principles*, Prentice Hall, Englewood Cliffs, N.J.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
5. Coulouris, G., J. Dollimore, and T. Kindberg (2001): *Distributed Systems-Concepts and Design*, 3rd ed., Addison-Wesley, New York.
6. Crowley, C. (1997): *Operating Systems-A Design Oriented Approach*, McGraw-Hill, New York.
7. Denning, P.J. (1971): "Third generation operating systems", *Computing Surveys*, **4** (1), 175-216.
8. Fortier, P.J. (1988): *Design of Distributed Operating Systems*, McGraw-Hill, New York.
9. Goscinski, A. (1991): *Distributed Operating Systems-The Logical Design*, Addison-Wesley, New York.
10. Liu, J.W.S. (2000): *Real-Time systems*, Pearson Education, New York.
11. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
12. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall.
13. Nutt, G. (2004): *Operating Systems-A Modern Perspective*, 3rd ed., Addison-Wesley, Reading, Mass.
14. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
15. Silberschatz, A., P.B. Galvin, and G. Gagne (2005): *Operating System Principles*, 7th ed., John Wiley, New York.
16. Singhal, M., and N.G. Shivaratri (1994): *Advanced Concepts in Operating Systems*, McGraw-Hill, New York.
17. Sinha, P.K. (1997): *Distributed Operating Systems*, IEEE Press, New York.
18. Smith, A.J. (1980): "Multiprogramming and memory contention", *Software-Practice and Experience*, **10** (7), 531-552.
19. Stallings, W. (2005): *Operating Systems-Internals and Design Principles*, 5th ed., Pearson Education, New York.
20. Stevens, W.R., and S.A. Rago (2005): *Advanced Programming in the Unix Environment*, 2nd ed., Addison-Wesley Professional.
21. Tanenbaum, A.S. (2003): *Computer Networks*, 4th ed., Prentice Hall, Englewood Cliffs, N.J.
22. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
23. Tanenbaum, A.S., and R. Van Renesse (1985): "Distributed Operating Systems", *Computing Surveys*, **17** (1), 419-470.
24. Tanenbaum, A.S. (1995): *Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, N.J.
25. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.
26. Wirth, N. (1969): "On multiprogramming, machine coding, and computer organization", *Communications of the ACM*, **12** (9), 489-491.
27. Zhao, W. (1989): "Special issue on real-time operating systems", *Operating System Review*, **23**, 7.

---

## CAPITOLO 4

# Struttura dei sistemi operativi

---

### Obiettivi di apprendimento

- Struttura di un sistema operativo
- Tipologie di kernel
- Sistemi operativi con kernel monolitico
- Sistemi operativi strutturati a livelli
- Sistemi operativi basati su virtual machine
- Sistemi operativi basati su microkernel
- Architettura dei sistemi operativi: Unix, Linux, Solaris, Windows

Durante il ciclo di vita di un sistema operativo, si possono verificare diversi cambiamenti nei computer e negli ambienti di elaborazione. Per meglio adattare un sistema operativo a questi cambiamenti, dovrebbe essere semplice implementare il sistema operativo su un nuovo computer e successivamente aggiungervi nuove funzionalità. Si parla pertanto rispettivamente di *portabilità* ed *espandibilità* di un sistema operativo.

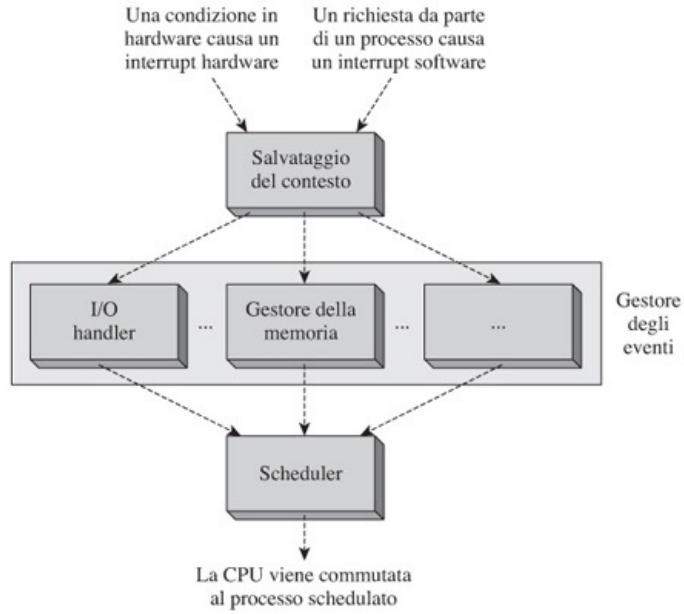
I primi sistemi operativi erano strettamente integrati con l'architettura di uno specifico computer. Questa caratteristica influenzava la loro portabilità. I moderni sistemi operativi sono implementati nella forma di un nucleo, detto *kernel* o *microkernel*, e costruiscono il resto del sistema operativo utilizzando i servizi offerti dal nucleo. Questa struttura restringe le dipendenze dall'architettura al solo nucleo del sistema operativo, per cui la portabilità è determinata dalle proprietà del suo kernel o microkernel. L'espandibilità di un sistema operativo è invece determinata dalla natura dei servizi offerti dal nucleo.

La struttura di un sistema operativo riguarda la natura del nucleo del SO e delle altre parti del sistema e le interazioni reciproche. Verranno descritte le diverse strutture di un sistema operativo e la loro influenza sulla portabilità e espandibilità.

### 4.1 Funzionamento di un SO

All'accensione di un computer, la *procedura di boot* analizza la sua configurazione: il tipo di CPU, la quantità di memoria, i dispositivi di I/O e i dettagli dell'hardware presente nel sistema (Paragrafo 1.3). Successivamente carica una parte del SO in memoria, inizializza le sue strutture dati con le informazioni ottenute e gli passa il controllo del sistema.

La [Figura 4.1](#) rappresenta un diagramma schematico del funzionamento di un SO (Paragrafo 2.3). Un evento, quale per esempio il completamento di un'operazione di I/O o la terminazione di una time slice, causa un interrupt. Quando un processo esegue una *system call*, per esempio per richiedere delle risorse o per avviare un'operazione di I/O, viene generato un interrupt denominato *interrupt software*, dal fatto che è generato tramite il software. L'azione dell'interrupt è quella di commutare la CPU a una procedura di servizio dell'interrupt. La procedura di servizio dell'interrupt esegue un'azione di *memorizzazione del contesto* per salvare le informazioni relative al programma interrotto e attiva un gestore dell'evento (*event handler*), che esegue le azioni appropriate per gestire l'evento. Lo scheduler successivamente seleziona un processo e commuta la CPU. La commutazione della CPU si verifica due volte durante l'elaborazione di un evento: prima al kernel per eseguire il gestore dell'evento e poi al processo selezionato dallo scheduler.



**Figura 4.1** Visione d'insieme del funzionamento di un SO.

Le funzioni di un SO sono dunque implementate da gestori di eventi e sono attivate dalle procedure di servizio degli interrupt. La [Tabella 4.1](#) riassume queste funzioni, che riguardano principalmente la gestione dei processi e delle risorse e la prevenzione dalle interferenze.

| Funzione                               | Descrizione  |
|--|--|
| Gestione dei processi                  | Avvio e terminazione dei processi, scheduling.   |
| Gestione della memoria                 | Allocazione e deallocazione della memoria, swapping, gestione della memoria virtuale.                                    |
| Gestione dell'I/O                      | Servizio degli interrupt di I/O, avvio delle operazioni di I/O, ottimizzazione delle prestazioni dei dispositivi di I/O. |
| Gestione dei file                      | Creazione, memorizzazione e accesso ai file.   |
| Sicurezza e protezione                 | Prevenzione dalle interferenze tra processi e risorse.   |
| Gestione delle comunicazioni (network) | Inviare e ricevere dati attraverso la rete   |

**Tabella 4.1** Funzione di un SO.

## 4.2 Struttura di un sistema operativo

### 4.2.1 Politiche e meccanismi

Nel determinare come un sistema operativo debba svolgere una sua funzione, il progettista del SO deve considerare a due livelli distinti:

- *politica*: una politica è un principio guida in base al quale il sistema operativo svolgerà la funzione;
- *meccanismo*: un meccanismo è un'azione specifica necessaria per implementare una politica.

Una politica decide *cosa* dovrebbe essere fatto, mentre un meccanismo determina *come*

dovrebbe esser fatto e in effetti lo fa. Una politica viene implementata come un modulo decisionale che decide quale modulo, che implementa un meccanismo, va richiamato e in quali circostanze. Un meccanismo viene implementato come modulo che esegue un'azione specifica. L'esempio seguente identifica le politiche e i meccanismi nello scheduling round-robin.

#### **Esempio 4.1 - Politiche e meccanismi nello scheduling round-robin**

A proposito di scheduling, considereremo come *politica* la tecnica round-robin (Paragrafo 3.6). I seguenti meccanismi sono necessari per implementare la politica di scheduling round-robin:

Mantenere un coda dei processi pronti

- Comutare la CPU per eseguire il processo selezionato (operazione di *dispatching*).

La politica di scheduling basata sulla priorità, che viene utilizzata nei sistemi multiprogrammati (Paragrafo 3.5.1), richiederebbe anche un meccanismo per conservare le informazioni sui processi pronti; tuttavia, sarebbe differente dal meccanismo utilizzato nello scheduling round-robin poiché organizzerebbe le informazioni in base alla priorità dei processi e non all'ordine di arrivo. Il meccanismo di dispatching, però, sarebbe comune a tutte le politiche di scheduling.

Fatta eccezione per i meccanismi di implementazione delle politiche di gestione di specifici processi o risorse, il SO dispone anche di meccanismi per eseguire azioni di gestione interna. L'azione di memorizzazione del contesto menzionato nel Paragrafo 4.1 è implementata come un meccanismo.

### **4.2.2 Portabilità ed espandibilità dei sistemi operativi**

La progettazione e l'implementazione dei sistemi operativi coinvolgono grandi investimenti finanziari. Per proteggere tali investimenti, la progettazione del sistema operativo dovrebbe avere un ciclo di vita pari a più di un decennio. Ma durante tale periodo di tempo, ci sarebbero molti cambiamenti nell'architettura dei computer, nella tecnologia dei dispositivi di I/O e negli ambienti di applicazione, e dovrebbe essere possibile adattare un SO a questi cambiamenti. Due caratteristiche sono importanti in questo contesto: portabilità ed espandibilità.

Per *porting* si intende l'atto di adattare un software all'utilizzo su un altro computer. La *portabilità* si riferisce alla facilità con la quale un software può essere portato; è inversamente proporzionale allo sforzo per il porting. L'*espandibilità* si riferisce alla facilità con la quale nuove funzionalità possono essere aggiunte al software di sistema.

Il porting di un SO implica il cambiamento di parti del codice che sono dipendenti dall'architettura in modo tale che il SO possa essere eseguito sul nuovo hardware. Alcuni esempi di dati e istruzioni nel SO dipendenti dall'architettura sono:

- un vettore di interrupt contenente le informazioni che dovrebbero essere caricate nei vari campi del PSW per commutare la CPU a una procedura di servizio dell'interrupt (Paragrafo 2.2.5). Questa informazione è specifica per ogni architettura;
- le informazioni riguardanti la protezione della memoria e le informazioni da fornire all'unità di gestione della memoria (memory management unit - MMU) sono specifiche per ogni architettura (Paragrafi 2.2.2 e 2.2.3);
- le istruzioni di I/O utilizzate per effettuare un'operazione di I/O sono specifiche per ogni architettura.

La parte dipendente dall'architettura del codice di un sistema operativo è generalmente associata ai meccanismi piuttosto che alle politiche. Un SO sarebbe caratterizzato da una elevata portabilità se il suo codice dipendente dall'architettura fosse di dimensione ridotta e il suo codice completo fosse strutturato in modo tale che lo sforzo per il porting fosse determinato dalla dimensione del codice dipendente dall'architettura, piuttosto che dal codice completo. Per questo motivo il problema della portabilità del SO è affrontato separando le parti dipendenti dall'architettura da quelle indipendenti, e fornendo interfacce ben definite tra le due parti.

L'espandibilità di un SO è necessaria per due scopi: per incorporare nuovo hardware in un computer (generalmente nuovi dispositivi hardware o adattatori di rete) e per fornire nuove funzionalità in risposta a nuove aspettative degli utenti. I primi sistemi operativi

non fornivano nessun tipo di espandibilità. Dunque anche l'aggiunta di un nuovo dispositivo di I/O richiedeva modifiche al SO. I sistemi operativi successivi risolsero questo problema aggiungendo una funzionalità alla procedura di boot. La procedura di boot cerca il nuovo hardware oppure richiede all'utente di selezionare il software appropriato per gestire il nuovo hardware, chiamato driver di dispositivo (*device driver*).

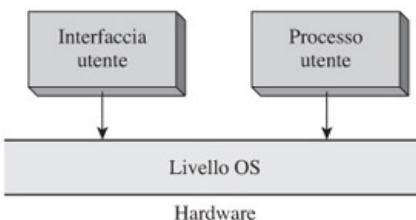
Il nuovo software viene poi caricato e integrato con il kernel così che possa essere richiamato e utilizzato in modo appropriato. I moderni sistemi operativi utilizzano la funzionalità *plug-and-play*, con la qual è possibile aggiungere un nuovo hardware anche quando il SO è in esecuzione. Il SO gestisce l'interrupt causato dall'aggiunta del nuovo hardware, seleziona il software appropriato e lo integra nel kernel.

La mancanza di espandibilità porta a difficoltà nell'adattare un SO alle nuove aspettative degli utenti. Diversi esempi di tali difficoltà possono essere trovate nella storia dei sistemi operativi. Negli anni '80 e '90, gli utenti dei PC desideravano una nuova funzione che permetesse di impostare, allo stesso tempo, più sessioni di un sistema operativo. Molti sistemi operativi ben noti in quel periodo, quale l'MS-DOS, avevano difficoltà a fornire questa funzionalità poiché non avevano sufficiente espandibilità. Una simile difficoltà fu sperimentata dal sistema operativo Unix nel supportare i sistemi multiprocessore. I dettagli riguardanti l'espandibilità verranno affrontati nel Paragrafo 4.7.

### 4.3 Sistemi operativi con struttura monolitica

Un SO è un software complesso che dispone di un gran numero di funzionalità e può contenere milioni di istruzioni. Viene progettato come un insieme di moduli e ogni modulo ha un'*interfaccia* ben definita che deve essere usata per accedere a una qualunque delle sue istruzioni o dati. Questo tipo di progettazione possiede la proprietà che un modulo non può "vedere" i dettagli di funzionamento interni ad altri moduli. Questa proprietà semplifica la progettazione, la codifica e il test di un SO.

I primi sistemi operativi avevano una struttura *monolitica*, secondo cui il SO formava un singolo strato software tra l'utente e la macchina, ovvero l'hardware del computer (**Figura 4.2**). L'interfaccia utente era costituita da un interprete dei comandi. Sia l'interprete dei comandi che i processi degli utenti richiamavano le funzioni e i servizi del SO attraverso le chiamate di sistema.



**Figura 4.2** SO monolitico.

Nel corso degli anni furono individuati due tipi di problemi relativi alla struttura monolitica. Lo strato base del SO aveva un'interfaccia con l'hardware, dunque il codice dipendente dall'architettura era distribuito in tutto il SO, per cui la portabilità era molto limitata. Inoltre, rendeva difficili le fasi di test e debug, comportando alti costi di manutenzione e aggiornamento. Questi problemi portarono alla ricerca di modi alternativi di strutturare un SO. Nei paragrafi successivi verranno discussi tre metodi per strutturare un SO, implementati come soluzioni a questi problemi.

- *Struttura a livelli*: la struttura a livelli affronta la complessità e il costo dello sviluppo e della manutenzione di un SO strutturandolo in un numero specifico di livelli (Paragrafo 4.4). Il sistema multiprogrammato THE del 1960 è un esempio di SO strutturato a livelli.
- *Struttura basata sul kernel*: la struttura basata sul kernel confina la dipendenza dall'architettura a una piccola sezione del codice del SO che costituisce il kernel (Paragrafo 4.6), migliorando la portabilità. Il SO Unix ha una struttura basata sul kernel.
- *Struttura di SO basato su microkernel*: il microkernel fornisce un insieme minimo di

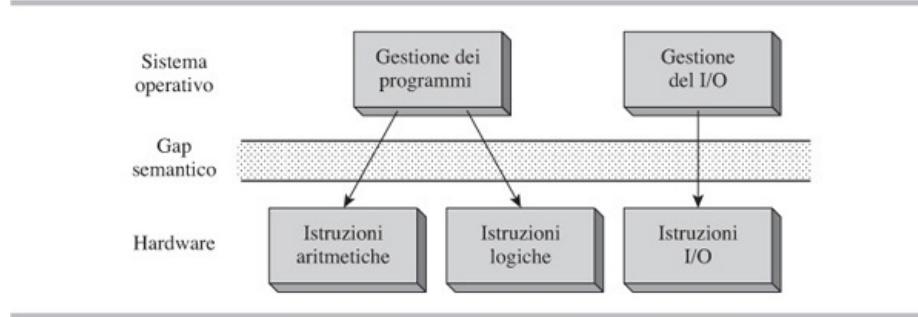
mezzi e servizi per implementare un SO, garantendo la portabilità. Inoltre, fornisce espandibilità poiché i cambiamenti possono essere apportati al SO senza richiedere modifiche al microkernel (Paragrafo 4.7).

#### 4.4 Sistemi operativi strutturati a livelli

La struttura monolitica presentava il problema che tutte le componenti del SO dovessero essere in grado di interagire con l'hardware. Questa caratteristica faceva aumentare il costo e l'impegno nello sviluppo di un SO a causa del grande *gap semantico* tra il sistema operativo e l'hardware.

**Definizione 4.1 Gap semantico** L'assenza di corrispondenza tra la natura delle operazioni necessarie all'applicazione e la natura delle operazioni fornite dall'hardware.

Il gap semantico può essere illustrato come segue: un'istruzione macchina implementa un'operazione primitiva a livello macchina, come per esempio la manipolazione aritmetica o logica degli operandi. Un modulo di SO può contenere un algoritmo, che utilizza delle operazioni primitive a livello di SO, come per esempio il salvataggio del contesto di un processo e l'inizializzazione di un'operazione di I/O. Tali operazioni sono più complesse delle operazioni primitive a livello macchina. Questa differenza porta a un grande gap semantico, che deve essere colmato attraverso la programmazione. Ogni operazione richiesta dal SO diventa una *sequenza* di istruzioni, possibilmente una routine (Figura 4.3). Questo comporta costi di programmazione elevati.



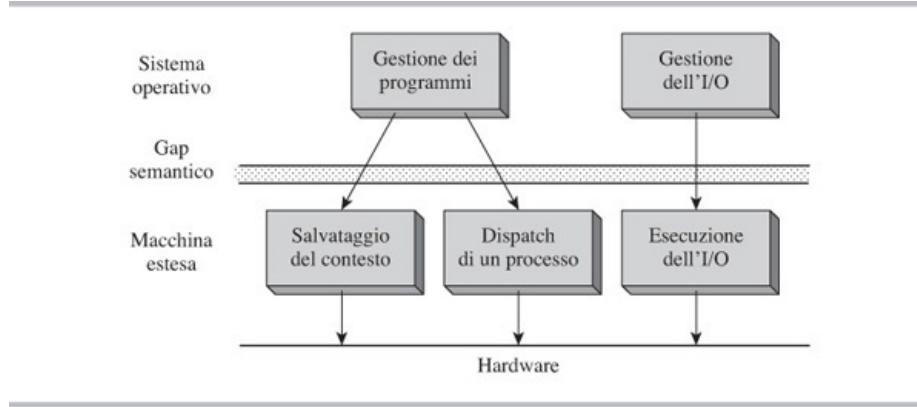
**Figura 4.3** Gap semantico.

Il gap semantico tra un SO e la macchina sulla quale opera può essere ridotto usando una macchina con più funzioni (un computer che mette a disposizione le istruzioni per effettuare alcune o tutte le operazioni che deve svolgere il sistema operativo) oppure *simulando* (*emulando*) via software una macchina dotata di più funzioni. Il primo approccio è costoso. Nel secondo approccio, tuttavia, il *simulatore* (*emulatore*), che è un programma, viene eseguito direttamente nell'hardware ed emula un macchina più potente che fornisce molte funzioni necessarie al SO. Questa nuova "macchina" è nota come *macchina estesa* e il suo simulatore (*emulatore*) è chiamato software della macchina estesa. In questo modo il SO si interfaccia con la macchina estesa piuttosto che con l'hardware; il software della macchina estesa costituisce pertanto un livello tra il SO e l'hardware.

La politica di base nella progettazione di un SO a livelli è che le routine di un livello devono usare solo le funzioni del livello direttamente sottostante, ovvero nessun livello può essere aggirato. Inoltre, l'accesso alle routine di livello inferiore deve aver luogo solo attraverso l'interfaccia tra i livelli. In questo modo, una routine situata a un livello non "conosce" gli indirizzi delle strutture dati o delle istruzioni presenti ai livelli inferiori, ma conosce esclusivamente il modo con il quale richiamare una routine appartenente al livello inferiore. Tale proprietà, che chiameremo *information hiding* previene l'uso scorretto o la corruzione dei dati di un livello da parte di routine situate ad altri livelli del SO. Durante il debugging, il rilevamento degli errori è più semplice poiché la causa di un errore a un livello, per esempio un valore non corretto in un dato, deve risiedere all'interno dello stesso livello. L'*information hiding* implica anche che un livello possa essere modificato senza influenzare gli altri livelli. Queste caratteristiche semplificano il

test e i debug di un SO.

La [Figura 4.4](#) mostra un SO organizzato a livelli. La macchina estesa fornisce le operazioni, quali il salvataggio del contesto, il dispatching, lo swapping e l'avvio dell'I/O. Il livello del sistema operativo è situato al di sopra del livello della macchina estesa. Questa organizzazione semplifica notevolmente la codifica e il test dei moduli del SO, separando una funzione dalla implementazione delle sue operazioni primitive. Ciò conduce a fasi di test, debug e modifica di un modulo più agevoli rispetto a un SO monolitico. Diremo che il livello più basso fornisce una *astrazione* ovvero la macchina estesa. Chiameremo il livello del sistema operativo il *livello top* di un SO.



**Figura 4.4** Progettazione di un SO a livelli.

Le strutture a livelli dei sistemi operativi si sono evolute in vari modi, usando differenti astrazioni e un numero differenti di livelli. L'Esempio 4.2 descrive il SO multiprogrammato THE, che utilizza una struttura multilivello e un processo come astrazione al livello più basso.

#### Esempio 4.2 - Struttura del sistema multiprogrammato THE

Il sistema multiprogrammato THE fu sviluppato alla Technische Hogeschool di Eindhoven in Olanda da Dijkstra e altri usando una progettazione a livelli. La [Tabella 4.2](#) mostra la gerarchia a livelli del sistema THE.

| Livello   | Descrizione                                       |
|-----------|---|
| Livello 0 | Allocazione del processore e multiprogrammazione. |
| Livello 1 | Gestione della memoria e del tamburo.             |
| Livello 2 | Comunicazione tra console e processi.             |
| Livello 3 | Gestione dell'I/O.                                |
| Livello 4 | Processi degli utenti.                            |

**Tabella 4.2** Livelli nel sistema multiprogrammato THE.

Il livello 0 del sistema gestisce l'allocazione del processore per implementare la multiprogrammazione. Questa funzione tiene traccia dello stato dei processi e gestisce la commutazione tra processi, utilizzando uno scheduling a priorità. I livelli al di sopra del livello 0 non devono occuparsi della gestione di queste problematiche e della presenza di più processi nel sistema.

Il livello 1 si occupa della gestione della memoria. Implementa una gerarchia di memoria che si compone della memoria a tamburo, che è un dispositivo di memorizzazione secondario (Paragrafo 2.2.3). I dettagli del trasferimento tra la memoria e il tamburo non interessano il resto del SO.

Il livello 2 implementa la comunicazione tra un processo e la console dell'operatore tramite l'allocazione di una console virtuale ad ogni processo. Il livello 3 effettua la gestione dell'I/O. In tal modo, i problemi inerenti la programmazione dell'I/O (Paragrafo 14.4) sono nascosti dal livello 4, che è occupato dai processi utente.

La progettazione a livelli di un SO causa tre problemi. Il funzionamento di un sistema

può essere rallentato dalla struttura a livelli. È bene, infatti, ricordare che ogni livello può interagire solo con i livelli adiacenti. Questo implica che una richiesta di un servizio fatta da un processo utente deve essere spostata dal livello col numero più alto al livello col numero più basso prima che l'azione richiesta venga effettuata dall'hardware. Questo modo di operare genera pertanto un elevato overhead.

Il secondo problema riguarda le difficoltà nello sviluppo di un progetto a livelli. Poiché un livello può accedere solo al livello immediatamente inferiore, tutte le caratteristiche e le funzionalità necessarie devono essere disponibili nei livelli inferiori. Questa esigenza pone un problema nell'ordinamento dei livelli che richiedono i rispettivi servizi. Il problema è spesso risolto dividendo un livello in due e ponendo altri livelli tra le due metà. Per esempio, un progettista può posizionare una funzione di gestione dei processi a un livello e la gestione della memoria al livello immediatamente successivo. Tuttavia, l'allocazione della memoria è richiesta come parte della creazione dei processi. Per risolvere tale problema, la gestione dei processi può essere suddivisa in due livelli. Uno dei due livelli effettuerebbe le funzioni di gestione dei processi quali il salvataggio del contesto, la commutazione, lo scheduling e la sincronizzazione dei processi. Il livello continuerebbe a essere al di sotto del livello per la gestione della memoria, mentre l'altro livello effettuerebbe la creazione dei processi e sarebbe posizionato al di sopra del livello per la gestione della memoria.

Il terzo problema riguarda la stratificazione delle funzionalità del SO. La stratificazione si verifica poiché ogni funzionalità deve essere divisa in parti che appartengano a differenti livelli del SO. Le parti devono usare le interfacce poste tra i vari livelli per comunicare l'una con l'altra. Per esempio, consideriamo una certa funzionalità  $F$  di un SO che si compone di due moduli,  $F_{l_1}$  e  $F_{l_2}$ , appartenenti rispettivamente ai livelli  $l_1$  e  $l_2$ . Se al livello  $l_2$  si può accedere solo attraverso un interrupt,  $l_1$  deve generare un interrupt per comunicare con  $l_2$ . Ciò può portare a un progetto complesso e alla perdita di efficienza nell'esecuzione. La stratificazione, inoltre, conduce a una scarsa espandibilità poiché l'aggiunta di una nuova funzionalità richiede che venga aggiunto nuovo codice all'interno di molti livelli del SO, il quale, a sua volta, può richiedere modifiche alle interfacce. Si può notare come il progetto di un SO multilivello non si focalizza sulla separazione delle parti del codice dipendenti dall'architettura; per esempio, quattro dei cinque livelli del sistema multiprogrammato THE descritto nella [Tabella 4.2](#) contengono parti dipendenti dall'architettura. In questo modo, una struttura a livelli non garantisce una elevata portabilità.

## 4.5 Macchina virtuale e sistemi operativi

Senza dubbio classi di utenti diverse hanno la necessità di differenti tipologie di servizi. Dunque, utilizzare un unico SO su di un computer può provocare una scarsa soddisfazione da parte di diversi utenti. Inoltre, far funzionare il computer con differenti SO in momenti differenti non costituisce una soluzione soddisfacente perché solo alcuni servizi di un sistema operativo sarebbero resi disponibili. Questo problema viene risolto utilizzando un *sistema operativo basato su macchine virtuali* (SO VM). Il SO VM crea differenti *macchine virtuali*. Ogni macchina virtuale viene allocata a un utente, che può utilizzare un SO a scelta ed eseguire i suoi programmi sul SO selezionato. In tal modo gli utenti del computer possono usare differenti sistemi operativi allo stesso tempo. Chiameremo ognuno di questi sistemi operativi *SO ospite* e chiameremo il SO della macchina virtuale *SO host*. Il computer utilizzato dal SO VM viene denominato generalmente *computer host*. Una *macchina virtuale* è una risorsa virtuale (Paragrafo 1.3.2). Consideriamo una macchina virtuale che ha la stessa architettura del computer host, cioè con una CPU virtuale in grado di eseguire le stesse istruzioni e con memoria e dispositivi di I/O simili. Tuttavia, può differire dalla macchina host in alcuni elementi della sua configurazione quali la dimensione della memoria e i dispositivi di I/O. Poiché l'architettura del computer host è identica a quella della macchina virtuale, non esistono gap semantici tra di loro, per cui il funzionamento della macchina virtuale non introduce nessuna perdita di prestazioni (confrontare questa configurazione con l'uso del livello macchina esteso descritto nel Paragrafo 4.4); inoltre, non sono necessarie modifiche al software per eseguire un SO ospite sulla macchina virtuale.

Il SO VM realizza il funzionamento concorrente dei sistemi operativi ospite attraverso un'azione simile alla commutazione dei processi: seleziona cioè una macchina virtuale e configura l'ambiente per consentire al SO ospite in esecuzione su di essa di eseguire le sue istruzioni sulla CPU. Il SO ospite in esecuzione ottiene il controllo completo

sull'ambiente della macchina host, inclusa la gestione degli interrupt. L'assenza di livello software tra la macchina host e il SO ospite assicura l'uso efficiente della macchina host. Un SO ospite controlla la macchina host finché il SO VM decide di passare a eseguire un'altra macchina virtuale, cosa che tipicamente accade in risposta a un interrupt. Il SO VM può utilizzare, per esempio, il timer per implementare gli scheduling basati su time-slice e round-robin dei SO ospite.

La gestione degli interrupt che vengono generati quando un SO ospite è in funzione richiede un'organizzazione abbastanza complessa. Alcuni interrupt verrebbero generati nello stesso dominio, per esempio un interrupt di I/O da un dispositivo incluso nella sua stessa macchina virtuale, mentre altri verrebbero generati nei domini di altri SO ospite. Il funzionamento del SO VM può essere organizzato in modo tale da prendere il controllo quando si verifica un interrupt, trovare il SO ospite al cui dominio appartiene l'interrupt e mandarlo in esecuzione per la gestione dell'interrupt. Tuttavia, questa organizzazione genera un elevato overhead a causa delle due operazioni di context switch, il primo context switch passa il controllo al SO VM e il secondo passa il controllo al SO ospite. Dunque il SO VM può adottare un'organizzazione in cui il SO ospite sarebbe invocato direttamente dagli interrupt che si verificano nel suo dominio. Il procedimento è implementato come segue: nel passare il controllo a un sistema operativo ospite, il SO VM sostituisce il suo vettore degli interrupt (Paragrafo 2.2.5) con quello definito nel SO ospite. Questa azione assicura che un interrupt commuta la CPU a una routine di gestione dell'interrupt del SO ospite. Se il SO ospite rileva che l'interrupt non si è verificato nel suo dominio, passa il controllo al SO VM richiamando una speciale system call denominata "invoke SO VM". Il SO VM passa l'interrupt al SO ospite destinatario. Naturalmente, nel caso di un numero elevato di macchine virtuali in esecuzione, la gestione degli interrupt può causare un intreccio eccessivo tra le macchine virtuali, pertanto il SO VM può non attivare immediatamente il SO ospite nel cui dominio si è verificato l'interrupt. Può quindi semplicemente annotare l'occorrenza degli interrupt che si sono verificati nel dominio di un SO ospite e segnalarli la prossima volta che viene "schedulato".

L'Esempio 4.3 descrive il funzionamento dell'IBM VM/370, un SO VM molto conosciuto negli anni '70.

### Esempio 4.3 - Struttura del VM/370

La [Figura 4.5](#) mostra tre SO ospite supportati dal VM/370. Il Conversational Monitor System (CMS) è un sistema operativo single-user, mentre il SO/370 e il DSO/370 sono sistemi operativi multiprogrammati. Un processo utente non è al corrente della presenza del VM/370, ma vede esclusivamente il SO ospite che utilizza. Per prevenire interferenze tra i SO ospite, la CPU viene posta in modalità utente mentre esegue un SO ospite. L'avvio di una operazione di I/O, che coinvolge l'uso di istruzioni privilegiate, viene gestita come segue: quando il kernel di un SO ospite esegue un'operazione di I/O, questa appare come un tentativo di eseguire un'istruzione privilegiata mentre la CPU è in modalità utente, tale da causare un interrupt da programma. L'interrupt è diretto al VM/370 piuttosto che al SO ospite. Il VM/370 avvia ora un'operazione di I/O eseguendo l'istruzione di I/O che ha causato l'interrupt.



**Figura 4.5** Il sistema operativo VM/370.

La distinzione tra modalità kernel e utente della CPU comporta alcune difficoltà nell'uso di un SO VM. Il SO VM deve infatti proteggersi dai SO ospite, per cui deve eseguire i SO ospite con la CPU in modalità utente. Tuttavia, in questo modo sia un SO ospite che i processi utente attivi al suo interno vengono eseguiti in modalità utente, cosa che rende vulnerabile il SO ospite a operazioni non legittime da parte di un processo utente. La famiglia di computer Intel 80x86 ha una funzionalità che fornisce un modo per superare questa difficoltà. I computer 80x86 supportano quattro modalità di esecuzione della CPU.

Dunque il SO ospite può essere eseguito con la CPU in modalità kernel, un SO ospite può eseguire processi nel suo contesto con la CPU in modalità utente ma può lui stesso andare in esecuzione con la CPU in una delle modalità intermedie. La *virtualizzazione* consiste nel realizzare l'associazione tra le interfacce e le risorse di una macchina virtuale e le interfacce e le risorse della macchina host. La virtualizzazione completa implica che la macchina host e la macchina virtuale abbiano identiche capacità, per cui un SO può operare allo stesso modo quando eseguito direttamente sull'hardware e sulla macchina virtuale supportata dal SO VM. Tuttavia, la virtualizzazione completa può indebolire la sicurezza. Nell'Esempio 4.3, abbiamo visto come il VM/370 non può determinare se l'uso di un'istruzione privilegiata è legittima: sarebbe legittima se fosse usata da un SO ospite, ma sarebbe illegittima se usata da un processo utente.

Gli ambienti delle moderne macchine virtuali adottano la tecnica della *quasi virtualizzazione* per superare i problemi che si presentano nella virtualizzazione completa. La quasi virtualizzazione sostituisce un'istruzione non virtualizzabile, ovvero un'istruzione che non può essere resa disponibile in una VM, mediante semplici istruzioni virtualizzate. Per esempio, il problema della sicurezza nel VM/370 potrebbe essere risolto con la quasi virtualizzazione nel modo seguente: le istruzioni privilegiate non sarebbero incluse in una macchina virtuale. D'altro canto, la macchina virtuale fornirebbe un'istruzione speciale utilizzabile da un SO ospite che volesse eseguire un'istruzione privilegiata. L'istruzione speciale causerebbe un interrupt software e passerebbe al SO VM l'informazione riguardante l'istruzione privilegiata che il SO ospite voleva eseguire e il SO VM eseguirebbe l'istruzione privilegiata al posto del SO ospite. Il SO host, il SO ospite e i processi utente utilizzerebbero pertanto differenti modalità di esecuzione della CPU in modo tale che il SO ospite possa riconoscere se l'istruzione speciale nella macchina virtuale era stata usata da un SO ospite o da un processo utente; nel secondo caso l'uso sarebbe considerato illegale. La quasi virtualizzazione è stata anche utilizzata per migliorare le prestazioni di un SO host. Quando nessuno dei processi utente attivi nel SO richiede l'uso della CPU il kernel del sistema operativo fa eseguire alla CPU un *ciclo idle*. Tuttavia, il tempo di CPU della macchina host sarebbe sprecato se un SO ospite entrasse in un ciclo idle. Dunque, la quasi virtualizzazione potrebbe essere utilizzata per fornire una istruzione speciale nella macchina virtuale per notificare questa condizione al SO ospite, cosicché il SO host possa sottrarre la CPU al SO ospite per un determinato periodo di tempo.

L'uso della quasi virtualizzazione implica che una macchina virtuale sia differente dalla macchina host, per cui il codice del SO ospite dovrebbe essere modificato per evitare l'uso di istruzioni non virtualizzabili. Questo può essere realizzato *portando* un SO ospite a operare nel SO VM. In alternativa, lo stesso risultato può essere ottenuto utilizzando la tecnica della *traduzione dinamica del codice binario* del kernel di un SO ospite; la traduzione consiste nel sostituire una parte del codice del kernel che contiene le istruzioni non virtualizzabili con codice che non contiene queste istruzioni. Per ridurre l'overhead di questa organizzazione, il codice del kernel modificato viene memorizzato in modo tale che la traduzione del codice binario non venga eseguita spesso.

Le macchine virtuali sono adattate per diversi scopi.

- Utilizzare un server esistente per una nuova applicazione che richiede l'uso di un sistema operativo differente. Tale situazione viene denominata *consolidamento del carico di lavoro*. L'obiettivo è ridurre il costo dell'hardware e del funzionamento dell'elaborazione riducendo il numero di server necessari.
- Fornire sicurezza e affidabilità per le applicazioni che usano lo stesso host e lo stesso SO. Questo beneficio deriva dal fatto che macchine virtuali di differenti applicazioni non possono avere accesso alle rispettive risorse.
- Testare un SO modificato (o una nuova versione di un'applicazione) su di un server parallelamente alla versione di produzione dello stesso SO.
- Fornire funzionalità di gestione dei malfunzionamenti trasferendo una macchina virtuale da un server a un altro server disponibile sulla rete.

Naturalmente, un SO VM è grande, complesso e costoso. Per rendere disponibili i benefici delle macchine virtuali a un costo inferiore, le macchine virtuali sono anche usate senza un SO VM. Due di queste organizzazioni sono descritte di seguito.

### **Virtual Machine Monitor (VMM)**

Un VMM, anche chiamato *hypervisor*, è uno strato software che opera al di sopra in un SO host. Virtualizza le risorse dell'host computer e supporta l'esecuzione concorrente di

più macchine virtuali. Quando un SO ospite è in esecuzione su ognuna delle macchine virtuali fornite dal VMM, il SO host e il VMM insieme forniscono una funzionalità equivalente a quella di un SO VM. VMware e XEN sono due VMM il cui scopo è di implementare centinaia di SO ospite su un computer host assicurando allo stesso tempo che un SO ospite comporti solo un marginale degrado delle prestazioni se confrontato alla sua implementazione diretta in hardware.

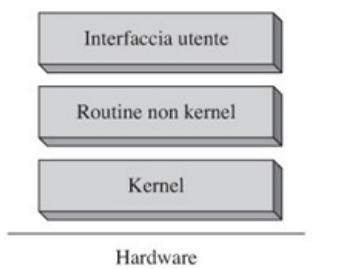
### **Linguaggi di programmazione per macchina virtuale**

Per ottenere alcuni dei benefici appena descritti, anche i linguaggi di programmazione hanno adottato il concetto di macchina virtuale. Negli anni '70, il linguaggio di programmazione Pascal adottò una macchina virtuale per fornire portabilità. La macchina virtuale aveva istruzioni chiamate *istruzioni P-code* che si adattavano bene all'esecuzione di programmi Pascal. La macchina virtuale era implementata via software come interprete delle istruzioni P-code. Un compilatore convertiva un programma Pascal in una sequenza di istruzioni P-code che potevano essere eseguite su qualunque computer dotato di un interprete P-code. La macchina virtuale conteneva un piccolo numero di istruzioni in modo tale che l'interprete fosse compatto e facilmente portabile. Questa caratteristica consentì la diffusione dell'uso del Pascal negli anni '70. Tuttavia, l'uso della VM provocava un sostanziale degrado delle prestazioni dovuta al gap semantico tra le istruzioni P-code e le istruzioni del computer host.

Il linguaggio di programmazione Java adotta una macchina virtuale per fornire sicurezza e affidabilità. Un programma Java si compone di oggetti, la cui struttura e comportamento sono specificate nelle classi. Ogni classe è compilata nella forma di un *bytecode*, ovvero una sequenza di istruzioni per la Java Virtual Machine (JVM). Durante l'esecuzione di un'applicazione Java, viene attivato il caricatore della classe (class loader) ogni qualvolta che si richiama un oggetto di una nuova classe. Il loader preleva il bytecode della classe da una libreria oppure da Internet e verifica che la classe sia conforme agli standard di sicurezza e affidabilità, ovvero che abbia una *firma digitale* valida (Paragrafo 20.3.2), e non utilizzi funzionalità come l'aritmetica dei puntatori. L'applicazione verrebbe terminata se una classe non superasse uno qualsiasi di questi controlli. Se diverse applicazioni Java fossero in esecuzione sullo stesso host, ognuna di esse sarebbe eseguita su una propria macchina virtuale; dunque il loro funzionamento non può causare reciproche interferenze. Il degrado delle prestazioni implicito nell'uso della macchina virtuale può essere compensata implementando la JVM in hardware.

## **4.6 Sistemi operativi basati su kernel**

La [Figura 4.6](#) rappresenta una vista astratta di un SO basato su kernel. Il *kernel* è il cuore di un SO e fornisce un insieme di istruzioni e servizi per supportare differenti funzioni. Il resto del SO è organizzato come un insieme di *routine non kernel*, che implementano operazioni sui processi e risorse di interesse per l'utente, e una *interfaccia grafica*. Si ricordi, dal Paragrafo 4.1 e dalla [Figura 4.1](#), che il funzionamento del kernel è guidato dagli interrupt. Il kernel prende il controllo quando un interrupt, come per esempio un interrupt generato da un timer o dal completamento di un'operazione di I/O, gli notifica l'occorrenza di un evento, o quando un'istruzione di interrupt viene eseguita per servire una *system call*. Quando viene generato l'interrupt, una procedura di servizio esegue la funzione di *salvataggio del contesto* e invoca un appropriato *gestore dell'evento*, che è una routine non kernel del SO.



**Figura 4.6** Struttura di un SO basato su kernel.

Una system call può essere richiamata dall’interfaccia utente per implementare un comando utente, oppure da un processo per richiedere un servizio al kernel, o ancora da una routine non kernel per richiedere una funzione del kernel. Per esempio, quando un utente lancia un comando per eseguire il programma memorizzato in qualche file, per esempio il file alpha, l’interfaccia utente effettua una system call e la procedura di servizio del kernel richiama una routine non kernel. La routine non kernel si occupa di eseguire le system call per allocare la memoria per l’esecuzione del programma, aprire il file alpha e caricare il suo contenuto nell’area di memoria allocata, seguita da un’altra system call per avviare l’esecuzione del processo che rappresenta l’esecuzione del programma. Se un processo richiede la creazione di un processo figlio per eseguire il programma nel file alpha, anch’esso dovrebbe effettuare una chiamata di sistema e seguire le stesse azioni.

Le motivazioni storiche della struttura del SO basato su kernel risiedono nella portabilità del SO e nella semplicità di progettazione e codifica delle routine non kernel. La portabilità del SO è ottenuta inserendo nel kernel le parti del codice del SO dipendenti dall’architettura, solitamente rappresentate dai *meccanismi*, e mantenendo le parti del codice indipendenti dall’architettura al di fuori, in modo tale che un eventuale porting sia limitato esclusivamente al kernel. Il kernel solitamente è monolitico per assicurare l’efficienza; la parte non kernel di un SO può essere monolitica, oppure può essere ulteriormente strutturata a livelli.

La [Tabella 4.3](#) contiene un esempio di funzioni e servizi offerti dal kernel per supportare diverse funzionalità del SO. Queste funzioni e servizi forniscono un insieme di astrazioni alle routine non kernel; il loro utilizzo semplifica la progettazione e la codifica delle routine non kernel riducendo il gap semantico (Paragrafo 4.4). Per esempio, le funzioni di I/O della [Tabella 4.3](#) implementano collettivamente l’astrazione dei dispositivi virtuali (Paragrafo 1.3.2). Un processo è un’altra astrazione fornita dal kernel.

| Funzionalità del SO        | Esempi di funzioni e servizi del kernel  |
|----------------------------|--|
| Gestione dei processi      | Salva il contesto del programma interrotto, effettua il dispatch di un processo, manipola le liste di scheduling.  |
| Comunicazioni tra processi | Invia e riceve messaggi dai processi.  |
| Gestione della memoria     | Imposta le informazioni relative alla protezione della memoria, effettua lo swap-in e lo swap-out, gestisce i page fault (ovvero l’interrupt “non presente in memoria”, detto missing from memory, del Paragrafo 1.4). |
| Gestione dell’I/O          | Avvia le operazioni di I/O, elabora l’interrupt generato dal completamento di un’operazione di I/O, effettua il recupero da errori di I/O.   |
| Gestione dei file          | Apre un file, legge/scrive dati.   |
| Sicurezza e protezione     | Aggiunge informazioni di autenticazione per un nuovo utente, conserva le informazioni per la protezione dei file.  |
| Gestione della rete        | Invia/riceve dati attraverso messaggi.   |

**Tabella 4.3** Funzioni e servizi tipici offerti dal kernel.

La progettazione basata su kernel può presentare problemi derivanti dalla stratificazione, in modo analogo alla progettazione del SO a livelli (Paragrafo 4.4) poiché il codice per implementare un comando può contenere una parte dipendente dall’architettura, tipicamente un *meccanismo* che sarebbe tenuto fuori dal kernel. Queste parti dovrebbero comunicare tra loro grazie all’uso delle system call, aumentando l’overhead del SO dovuto alle azioni di servizio degli interrupt. Si consideri il comando per avviare l’esecuzione di un codice in un programma chiamato alpha. Come discusso in precedenza, la routine non kernel che implementa il comando effettuerebbe quattro chiamate di sistema per: allocare la memoria, aprire il file alpha, caricare in memoria il programma in esso contenuto e avviare la sua esecuzione. Tutto ciò produrrebbe un considerevole overhead. Alcuni progetti di sistemi operativi riducono questo overhead includendo nel kernel la parte dipendente dall’architettura del codice di una funzione. In questo modo, le routine non kernel che avviano l’esecuzione di un programma

diventerebbero parte del kernel. Altri esempi di questo tipo sono le politiche di schedulazione dei processi e di gestione della memoria. Tali modifiche da un lato riducono l'overhead del SO, ma allo stesso tempo riducono anche la sua portabilità.

I sistemi operativi basati su kernel presentano una ridotta espandibilità poiché l'aggiunta di nuove funzionalità può richiedere cambiamenti nelle funzioni e nei servizi offerti dal kernel.

#### 4.6.1 Evoluzione della struttura basata su kernel

La struttura dei sistemi operativi basati su kernel si è evoluta per compensare alcuni dei suoi svantaggi. Gli elementi fondamentali di tale evoluzione sono i moduli del kernel caricabili dinamicamente e i driver dei dispositivi a livello utente.

Per rendere disponibile *il caricamento dinamico dei moduli del kernel*, il kernel è progettato come un insieme di moduli che interagiscono tra di loro attraverso interfacce ben definite. Un *kernel base* composto di un nucleo di moduli viene caricato durante la fase di boot del sistema. Gli altri moduli, che si conformano alle interfacce del kernel base, sono caricati quando le loro funzionalità sono richieste e sono rimossi dalla memoria quando non sono più necessari. L'utilizzo dei moduli caricabili preserva la memoria durante il funzionamento del SO poiché solo i moduli richiesti del kernel a un certo istante sono in memoria. Inoltre, questa tecnica fornisce l'espandibilità, dal momento che i moduli del kernel possono essere modificati e nuovi moduli possono essere facilmente aggiunti al kernel. L'uso dei moduli caricabili presenta anche alcuni svantaggi. Il caricamento e la rimozione dei moduli frammenta la memoria; in questo modo il kernel deve effettuare operazioni di gestione della memoria per ridurre le richieste di memoria. Un modulo contenente un bug può inoltre causare un blocco del sistema. I moduli del kernel caricabili sono utilizzati per implementare i driver di nuovi dispositivi di I/O, per gli adattatori di rete, o per nuovi file system, che in molti sistemi operativi sono semplicemente driver di dispositivo; inoltre, vengono utilizzati per aggiungere nuove system call al kernel. Per esempio, i sistemi Linux e Solaris hanno incluso il supporto per i moduli del kernel caricabili dinamicamente (Paragrafi 4.8.2 e 4.8.3).

Un *driver di dispositivo* gestisce una specifica classe di dispositivi di I/O. Come risultato dei rapidi cambiamenti nelle interfacce dei dispositivi di I/O, i driver di dispositivo rappresentano quella parte del SO che cambia più di continuo; come conseguenza, la facilità con cui possono essere testati e aggiunti a un SO determinerebbe l'affidabilità e l'espandibilità del SO stesso. I driver di dispositivo caricabili dinamicamente migliorano entrambi questi aspetti; tuttavia, un driver di dispositivo funzionerebbe con i privilegi del kernel, per cui un bug potrebbe interrompere il funzionamento del SO e causare frequenti riavvii. Consentire a un driver di dispositivo di operare in modalità utente consentirebbe di superare questa difficoltà. Un tale driver di dispositivo è chiamato *driver di dispositivo a livello utente*.

I driver di dispositivo a livello utente consentono facilità di sviluppo, debug, messa in opera e robustezza, poiché sia il codice del kernel che il suo funzionamento non risentono della presenza dei driver a livello utente. Tuttavia, essi presentano problemi di prestazioni. Le prime progettazioni di driver a livello utente causavano una caduta del throughput dell'I/O o un incremento del tempo di CPU utilizzato per le operazioni di I/O. Entrambi erano causati dall'elevato numero di chiamate di sistema necessarie per implementare un'operazione di I/O. Per esempio, il driver di dispositivo doveva effettuare le system call per impostare e dismettere il DMA per l'operazione di I/O, per risvegliare il processo utente in attesa del completamento dell'operazione di I/O e per restituire il controllo al kernel al termine della sua esecuzione. Gli sviluppi successivi di hardware e software hanno superato tali problemi di prestazioni in molti modi. Le azioni di impostazione e dismissione sono state semplificate dalla presenza dell'unità IOMMU e le system call sono state velocizzate attraverso il supporto specifico nel kernel.

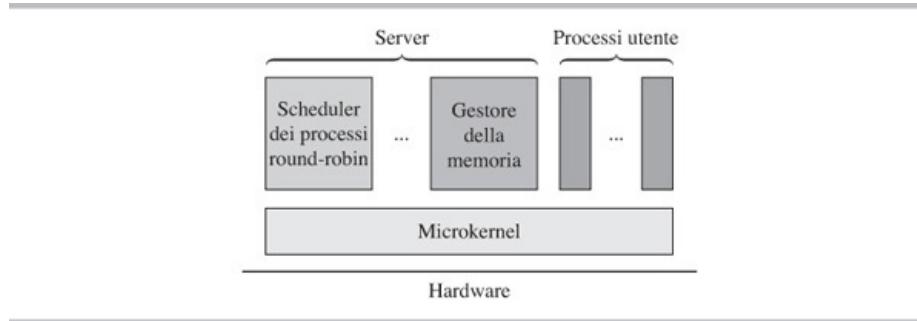
#### 4.7 Sistemi operativi basati su microkernel

Mettere tutto il codice del SO dipendente dall'architettura nel kernel fornisce una buona portabilità. Tuttavia, in pratica, anche i kernel contengono del codice indipendente dall'architettura. Questa caratteristica causa diversi problemi. I kernel sono di grandi dimensioni, cosa che allontana l'obiettivo della portabilità. Inoltre, per incorporare nuove caratteristiche spesso è necessario effettuare modifiche al kernel, che porta a poca

espandibilità. Un kernel di maggiore dimensione supporta un gran numero di chiamate di sistema. Alcune di queste chiamate possono essere usate raramente e, di conseguenza, le loro implementazioni in diverse versioni del kernel possono non essere accuratamente testate. Questo compromette l'affidabilità del SO.

Il *microkernel* fu sviluppato nei primi anni '90 per superare i problemi relativi alla portabilità, all'espandibilità e all'affidabilità dei kernel. Un microkernel è un nucleo essenziale del codice del SO, per cui contiene solo un sottoinsieme dei meccanismi tipicamente inclusi in un kernel e supporta solo un piccolo numero di system call, che sono ampiamente testate e utilizzate. Questa caratteristica migliora la portabilità e l'affidabilità del microkernel. Le parti meno essenziali del SO si trovano al di fuori del microkernel e utilizzano i suoi servizi; pertanto queste parti potrebbero essere modificate senza influenzare il kernel. In linea di principio, queste modifiche potrebbero essere fatte senza dover riavviare il sistema! I servizi messi a disposizione in un microkernel non propendono per una specifica caratteristica o politica, per cui nuove funzionalità e caratteristiche potrebbero essere aggiunte al SO per adattarsi a specifici ambienti di funzionamento.

La [Figura 4.7](#) illustra la struttura di un SO basato su microkernel. Il microkernel include i meccanismi per la schedulazione dei processi e la gestione della memoria, ecc., ma non include uno scheduler o un gestore della memoria. Queste funzioni sono implementate come *server*, ovvero semplici processi che non terminano mai. I server e i processi utente operano al di sopra del microkernel, il quale effettua semplicemente la gestione degli interrupt e fornisce la comunicazione tra i server e i processi utente.



**Figura 4.7** Struttura dei sistemi operativi basati su microkernel.

La dimensione ridotta e l'espandibilità dei microkernel sono proprietà preziose per l'ambiente dei sistemi embedded, poiché i sistemi operativi hanno bisogno di essere sia piccoli che messi bene a punto per soddisfare i requisiti di un'applicazione embedded. L'espandibilità di un microkernel inoltre sconsiglia la visione di utilizzare lo stesso microkernel per un ampio spettro di computer, dai sistemi palmari a grandi sistemi paralleli e distribuiti. Questa visione in qualche misura è stata realizzata. Il microkernel Mach è stato usato per implementare diverse versioni di Unix. Il sistema distribuito Amoeba usa un microkernel identico su tutti i computer in un sistema distribuito che va dalle workstation a grandi sistemi multiprocessore.

Cosa sia il "nucleo essenziale del codice del SO" è stato un argomento di dibattito e attualmente esistono molte varianti nei servizi inclusi in un microkernel. Per esempio, l'implementazione IBM del microkernel Mach lascia fuori dal kernel la politica di schedulazione dei processi e i driver di dispositivo: queste funzioni vengono eseguite come server. Il microkernel QNX include le routine di gestione dell'interrupt, la schedulazione dei processi, la comunicazione tra i processi e i servizi di rete essenziali. Il microkernel L4 include la gestione della memoria e supporta solo sette system call. Sia QNX che L4 hanno una dimensione di soli 32 KB. Nonostante una tale variabilità, si può sostenere che certi servizi *devono* essere messi a disposizione dal microkernel. Questi includono il supporto per la gestione della memoria, la comunicazione tra processi e la gestione degli interrupt. La gestione della memoria e la comunicazione tra processi sarebbero richiamate dai moduli di più alto livello al di fuori del microkernel. La routine di servizio dell'interrupt accetterebbe gli interrupt e li passerebbe ai moduli di livello più alto per l'elaborazione.

I sistemi operativi che utilizzavano la prima generazione di microkernel erano caratterizzati fino al 50% di perdita nel throughput se confrontati con i sistemi operativi che non usavano i microkernel. Questo problema ha la sua origine nel fatto che alcune

funzionalità di un kernel convenzionale sono suddivise tra un microkernel e un SO implementato utilizzando il microkernel: si presenta ancora il solito problema della stratificazione. Per esempio, un kernel include completamente la funzione di gestione dei processi, che esegue la creazione, la schedulazione e il dispatching dei processi, invece un microkernel potrebbe includere solo la creazione dei processi e il dispatching, mentre la schedulazione dei processi potrebbe essere eseguita come un server al di fuori del microkernel. La comunicazione tra le due parti richiederebbe l'uso della funzionalità di comunicazione tra processi (interprocess communication, IPC). I ricercatori scoprirono che fino al 73% della perdita di prestazioni era dovuta all'IPC. Il microkernel L4, che rappresenta un microkernel di seconda generazione, implementò l'IPC in maniera più efficiente eliminando di default i controlli di validità e dei permessi e mettendo a punto il microkernel per l'hardware utilizzato. Queste azioni resero l'IPC 20 volte più veloce rispetto a quello utilizzato nei microkernel di prima generazione. Inoltre, le attività di paginazione relative alla gestione della memoria furono spostate fuori dal microkernel e nel sistema operativo costruito usando il microkernel. Dopo questi miglioramenti, si constatò che i sistemi operativi basati su microkernel presentavano un throughput peggiore solo del 5% rispetto ai sistemi operativi che non usavano un microkernel.

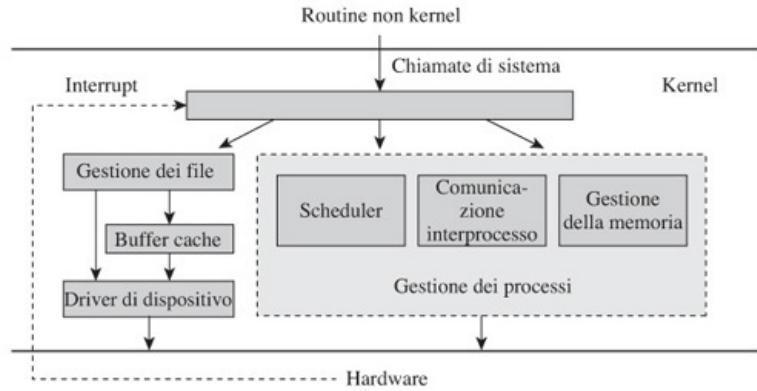
L'*exokernel* presenta una filosofia di struttura del SO radicalmente differente per ridurre il degrado delle prestazioni: la gestione delle risorse non ha bisogno di essere centralizzata e può essere effettuata dalle stesse applicazioni in maniera distribuita. Di conseguenza, un exokernel fornisce in maniera semplice un multiplexing efficiente delle risorse hardware, ma non fornisce nessuna astrazione. In questo modo un processo vede una risorsa nel computer nella sua forma grezza. Questo approccio risulta in operazioni primitive estremamente veloci, 10/100 volte più veloci rispetto a un kernel Unix monolitico. Per esempio, i dati letti da un dispositivo di I/O passano direttamente al processo che li ha richiesti; non passano quindi attraverso l'exokernel, mentre invece sarebbero passati attraverso il kernel Unix. Poiché le funzionalità tradizionali di un SO sono implementate al livello di applicazione, un'applicazione può selezionare e usare un SO da una libreria di sistemi operativi. Il SO viene eseguito come un *processo* in modalità non kernel e utilizza le caratteristiche dell'exokernel.

## 4.8 Casi di studio

Nei paragrafi precedenti è stata affrontata la struttura di un sistema operativo, ovvero, l'organizzazione delle sue parti e le proprietà delle diverse organizzazioni. In questo paragrafo, discuteremo sia della struttura che dell'*architettura* di alcuni sistemi operativi moderni, dove per architettura si intende la struttura del sistema operativo nonché le funzionalità delle sue componenti e le relazioni tra di esse. Le caratteristiche di progettazione e implementazione di alcune componenti di SO specifici sono descritte in capitoli dedicati delle Parti II-IV del libro.

### 4.8.1 Architettura di Unix

Unix è un sistema operativo basato su kernel. La [Figura 4.8](#) mostra un diagramma del kernel Unix. Si compone di componenti principali, gestione dei processi e gestione dei file. La componente di gestione dei processi si compone di un modulo per la comunicazione tra processi, la quale implementa la comunicazione e la sincronizzazione tra processi; inoltre fanno parte di questa parte del kernel i moduli di gestione della memoria e di schedulazione. La componente relativa alla gestione dei file effettua l'I/O attraverso i driver di dispositivo. Ogni driver di dispositivo gestisce una classe specifica di dispositivi di I/O e utilizza tecniche come la schedulazione del disco per assicurare un buon throughput di un dispositivo di I/O. Un buffer della cache viene utilizzato per ridurre sia il tempo richiesto per implementare un trasferimento dati tra un processo e un dispositivo di I/O, sia il numero di operazioni effettuate sui dispositivi come i dischi (Paragrafo 1.4.4).



**Figura 4.8** Kernel del sistema operativo Unix.

Le componenti del kernel relative alla gestione dei processi e dei file sono attivate attraverso la generazione in hardware di interrupt, attraverso le chiamate di sistema effettuate dai processi e dalle routine non kernel del SO. L'interfaccia utente del SO è costituita da un interprete dei comandi, denominato *shell*, in esecuzione come processo utente. Il kernel Unix non può essere interrotto arbitrariamente; può essere interrotto esclusivamente quando termina un processo che sta eseguendo codice kernel, o quando la sua esecuzione raggiunge un punto in cui può essere interrotto senza creare problemi. Questa caratteristica assicura che le strutture dati del kernel non siano in uno stato inconsistente quando si verifica un interrupt e un altro processo comincia a eseguire il codice del kernel, cosa che semplifica drasticamente la codifica del kernel (Paragrafo 2.3.2).

Il kernel Unix ha una lunga storia di oltre quattro decenni. Il kernel originale era piccolo e semplice. Forniva un piccolo insieme di astrazioni, caratteristiche semplici ma potenti come il meccanismo delle pipe, che consentivano agli utenti di eseguire diversi programmi concorrentemente e un piccolo file system che supportava solo un tipo di organizzazione dei file chiamata *byte stream*. Tutti i dispositivi erano rappresentati come file, cosa che unificava la gestione dei dispositivi di I/O e dei file. Il kernel era scritto in linguaggio C e aveva un dimensione inferiore a 100 KB, risultando facilmente portabile.

Tuttavia, il kernel Unix era monolitico e non molto espandibile. Per questo motivo è stato necessario modificarlo con l'avvento di nuovi ambienti di elaborazione, come l'ambiente client-server. La comunicazione tra processi e thread fu aggiunta per supportare l'elaborazione client-server. Analogamente, il supporto di rete richiese la modifica del kernel.

Un punto forte di Unix era l'uso di standard aperti. Questa caratteristica consentì a un gran numero di organizzazioni che spaziavano dal mondo accademico a quello industriale, a partecipare al suo sviluppo, cosa che portò a un uso diffuso di Unix, ma portò anche allo sviluppo di un gran numero di varianti a causa dell'evoluzione concorrente e non coordinata. Il kernel divenne ingombrante, crescendo fino a raggiungere la dimensione di pochi milioni di byte, cosa che influenzò la sua portabilità. In tale periodo, fu aggiunta una funzionalità per caricare dinamicamente i moduli del kernel quando necessario. Questa caratteristica riduceva i requisiti di memoria del kernel, ma non la dimensione del codice. Dunque non migliorò la portabilità.

Sono stati fatti molti sforzi nel tempo per riprogettare il kernel Unix e renderlo modulare ed espandibile. Il kernel Mach, che pone un'enfasi particolare sui sistemi multiprocessore, è un esempio di questo andamento. Successivamente Mach si è evoluto in un sistema operativo basato su microkernel.

### 4.8.2 Il Kernel di Linux

Il sistema operativo Linux fornisce le funzionalità dell'Unix System V e dell'Unix BSD; inoltre è conforme allo standard Posix. Inizialmente fu implementato su un Intel 80386 e da allora è stato implementato su processori Intel successivi e su diverse altre architetture.

Linux possiede un kernel monolitico. Il kernel è progettato per comprendere un

insieme di moduli caricabili individualmente. Ogni modulo ha un'interfaccia ben definita che indica come le sue funzionalità possono essere invocate e come altri moduli possono accedere alle sue strutture. Inoltre, l'interfaccia specifica anche le funzioni e i dati di altri moduli utilizzati da questo modulo. Ogni modulo può essere caricato individualmente in memoria, o rimosso da essa, in base al fatto che verosimilmente sarà utilizzato nel prossimo futuro. In linea di principio, ogni componente del kernel può essere strutturata come modulo caricabile, ma tipicamente i driver di dispositivo sono implementati come moduli.

Un numero ristretto di moduli del kernel sono caricati quando il sistema viene avviato. Un nuovo modulo del kernel è caricato dinamicamente quando necessario; tuttavia, deve essere integrato con i moduli del kernel che già si trovano in memoria in modo tale che i moduli in memoria possano funzionare come un kernel monolitico. Questa integrazione viene effettuata come segue: il kernel mantiene una tabella in cui memorizza gli indirizzi delle funzioni e dei dati definiti nei moduli caricati in memoria. Nel caricare un nuovo modulo, il kernel ne analizza l'interfaccia e cerca quali funzioni e dati di altri moduli utilizza, recupera gli indirizzi relativi nella tabella e li inserisce nelle istruzioni appropriate del nuovo modulo. Alla fine di questa fase, il kernel aggiorna la sua tabella aggiungendo gli indirizzi delle funzioni e dei dati definiti nel nuovo modulo.

L'uso dei moduli del kernel con un'interfaccia ben definita fornisce diversi vantaggi. L'esistenza dell'interfaccia del modulo semplifica la fase di test e la manutenzione del kernel. Un singolo modulo può essere modificato per fornire nuove funzionalità o migliorare quelle esistenti. Questa caratteristica consente di superare il problema della scarsa espandibilità tipicamente associata con i kernel monolitici. L'uso dei moduli caricabili limita inoltre le richieste di memoria del kernel, poiché alcuni moduli possono non essere caricati durante il funzionamento del sistema. Al fine di aumentare questo vantaggio, il kernel ha una funzionalità per rimuovere automaticamente dalla memoria i moduli non utilizzati; periodicamente il kernel genera un interrupt e controlla quale dei suoi moduli in memoria non è stato utilizzato dall'ultimo interrupt. Questi moduli sono rimosso dal kernel e dalla memoria. In alternativa, i moduli possono essere caricati individualmente e rimosso dalla memoria mediante chiamate di sistema.

Il kernel Linux 2.6, rilasciato nel 2003, ha eliminato molte delle limitazioni del kernel 2.5 e ha inoltre migliorato le sue funzionalità in molti modi. Due dei miglioramenti più importanti ebbero come obiettivo un sistema maggiormente reattivo e in grado di supportare sistemi embedded. I kernel fino alla versione 2.5 erano nonpreemptive, per cui se il kernel fosse stato coinvolto nell'esecuzione di un task a bassa priorità, i task a priorità più alta del kernel sarebbero stati ritardati. Con la versione 2.6 di Linux, il kernel è stato reso prelazionabile, cosa che lo ha reso più reattivo ai programmi e alle applicazioni degli utenti. Tuttavia, il kernel non dovrebbe essere prelazionato quando è difficile salvare il suo stato, o quando sta eseguendo operazioni sensibili alle interruzioni. Per questo motivo il kernel può disabilitare e abilitare la propria prelazionabilità mediante l'uso di speciali funzioni. Il kernel Linux 2.6 può anche supportare architetture che non possiedono la memory management unit (MMU), cosa che lo rende adatto all'uso nei sistemi embedded. In questo modo, lo stesso kernel può ora essere utilizzato nei sistemi embedded, nei pc desktop e nei server. L'altra caratteristica degna di nota nel kernel Linux 2.6 è una migliore scalabilità ottenuta attraverso un miglioramento del modello a thread, uno scheduler migliorato e una veloce sincronizzazione tra i processi; queste caratteristiche sono descritte nei capitoli successivi.

### 4.8.3 Il Kernel di Solaris

I primi sistemi operativi per i computer Sun erano basati sull'Unix BSD; tuttavia, lo sviluppo successivo fu basato sull'Unix SVR4. Le versioni precedenti alla SVR4 del SO sono chiamate SunOs, mentre quelle basate su SVR4 e quelle successive sono chiamate Solaris. Sin dagli anni '80, la Sun si è focalizzata sul supporto di rete e sul calcolo distribuito; molte delle caratteristiche di rete e dell'elaborazione distribuita dei sistemi operativi della Sun sono diventate standard industriali, quali, per esempio, le remote procedure calls (RPC) e un file system per ambienti distribuiti (NFS). Successivamente, Sun si è anche focalizzata sui sistemi multiprocessore, enfatizzando il kernel multithreading, rendendolo prelazionabile (Paragrafo 2.3.2) e adottando nel kernel tecniche di sincronizzazione veloci.

Il kernel di Solaris adotta un livello di macchina astratta che supporta un insieme molto ampio di architetture di processori SPARC e della famiglia dei processori Intel 80x86, includendo le architetture multiprocessore. Il kernel è interamente prelazionabile

e mette a disposizione funzionalità real-time. Solaris 7 adotta la metodologia di progettazione dei moduli del kernel caricabili dinamicamente (Paragrafo 4.6.1). Il nucleo del kernel è esso stesso un modulo che viene sempre caricato; contiene le routine di servizio degli interrupt, le chiamate di sistema, la gestione dei processi e della memoria e un framework di un file system virtuale che può supportare diversi file system in maniera concorrente. Gli altri moduli del kernel sono caricati e scaricati dinamicamente. Ogni modulo contiene le informazioni riguardanti gli altri moduli da cui dipende e altri moduli che dipendono da lui. Il kernel mantiene una tabella dei simboli contenente le informazioni relative ai simboli definiti nei moduli kernel attualmente caricati. Nuove informazioni sono aggiunte alla tabella dei simboli dopo che un modulo è stato caricato, mentre alcune informazioni sono cancellate dopo che un modulo è stato scaricato.

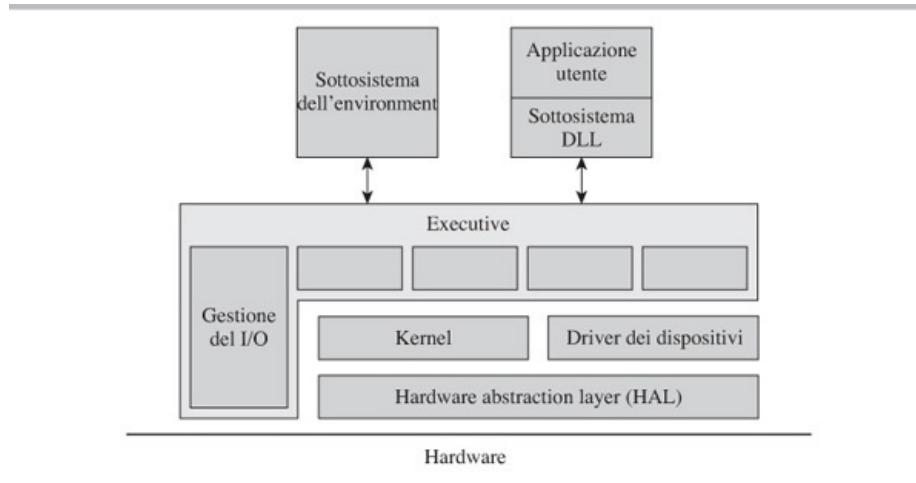
Il kernel di Solaris supporta sette tipi di moduli caricabili:

- classi di scheduler;
- file system;
- system call;
- loader per diversi tipi di file eseguibili;
- moduli di stream;
- controller dei bus e driver di dispositivi;
- altri moduli.

L'uso dei moduli del kernel caricabili fornisce una facile espandibilità. In questo modo, possono essere aggiunti facilmente nuovi file system, nuovi formati di file eseguibili, nuove chiamate di sistema e nuovi tipi di bus e dispositivi. Una caratteristica interessante nel kernel è che quando deve essere aggiunto un nuovo modulo, il kernel crea un nuovo thread per il caricamento, il linking (o collegamento) e l'inizializzazione del nuovo modulo. Questa organizzazione permette il caricamento dei moduli in modo concorrente con il normale funzionamento del kernel. Inoltre, consente di effettuare il caricamento di più moduli concorrentemente.

#### 4.8.4 Architettura di Windows

La [Figura 4.9](#) mostra l'architettura del sistema operativo Windows. L'*hardware abstraction layer* (HAL) si interfaccia con l'hardware del computer e mette a disposizione le astrazioni delle interfacce di I/O, dei controller degli interrupt e i meccanismi di comunicazione tra processori in un sistema multiprocessore. Il kernel utilizza le astrazioni fornite dall'HAL: mettere a disposizione i servizi di base come l'elaborazione degli interrupt e la sincronizzazione multiprocessore. In questo modo, il kernel è indipendente dalle peculiarità di una specifica architettura, cosa che migliora la sua portabilità. L'HAL e il kernel sono insieme equivalenti a un kernel convenzionale ([Figura 4.6](#)). Un *driver di dispositivo*, inoltre, utilizza le astrazioni fornite dall'-HAL per gestire le operazioni di I/O su una classe di dispositivi.



**Figura 4.9** Architettura di Windows.

Il kernel esegue le funzioni di sincronizzazione dei processi e di scheduling. L'*executive*

comprende le routine non kernel del SO; il suo codice utilizza le funzionalità nel kernel per fornire servizi come la creazione e la terminazione dei processi, la gestione della memoria virtuale, una funzionalità per il passaggio di messaggi tra processi per la comunicazione client/server chiamato *local procedure call* (LPC), gestione della memoria e un *cache file* per fornire un I/O efficiente dei file e un *security reference monitor* che effettua la validazione degli accessi ai file.

Il *gestore dell'I/O* utilizza i driver di dispositivo, caricati dinamicamente quando necessario. Molte funzioni dell'executive funzionano in modalità kernel, evitando in questo modo context switch frequenti quando l'executive interagisce con il kernel; questa organizzazione ha ovvi benefici sulle prestazioni.

I sottosistemi dell'ambiente forniscono supporto per l'esecuzione dei programmi sviluppati per altri sistemi operativi come MS-DOS, Win32 e SO/2. Operativamente, un sottosistema dell'ambiente è analogo a un sistema operativo ospite all'interno di un SO a macchina virtuale (Paragrafo 4.5). Esso funziona come un processo che tiene traccia dello stato delle applicazioni degli utenti che utilizzano i suoi servizi. Per implementare l'interfaccia di un SO ospite, ogni sottosistema di ambiente fornisce una *libreria collegata dinamicamente* (dynamic linked library - DLL) e si suppone che le applicazioni degli utenti invochino la DLL quando necessitano di uno specifico servizio di sistema. La DLL implementa il servizio richiesto, oppure passa la richiesta del servizio all'executive oppure, ancora, invia un messaggio al sottosistema dell'ambiente affinché fornisca il servizio.

## Riepilogo

La *portabilità* di un sistema operativo si riferisce alla facilità con cui il SO può essere implementato su un computer che ha una differente architettura. L'*espandibilità* di un sistema operativo si riferisce alla facilità con cui le sue funzionalità possono essere modificate o migliorate per adattarle a un nuovo ambiente di elaborazione. La portabilità e l'espandibilità sono diventate requisiti fondamentali a causa del lungo ciclo di vita dei moderni sistemi operativi. In questo capitolo abbiamo discusso modi differenti di strutturare i sistemi operativi per soddisfare questi requisiti.

Una funzionalità di un SO tipicamente contiene una *politica*, che specifica il principio che deve essere usato per svolgere la funzionalità e alcuni *meccanismi* che effettuano le azioni per implementare la funzionalità. I meccanismi come il dispatching e il salvataggio del contesto interagiscono strettamente con il computer, quindi il loro codice è intrinsecamente dipendente dall'architettura. Dunque la portabilità e l'espandibilità di un SO dipendono da come è strutturato il codice, dalle sue politiche e meccanismi.

I primi sistemi operativi avevano una struttura *monolitica*. Questi sistemi operativi avevano poca portabilità poiché il codice dipendente dall'architettura era presente in gran parte nel SO. Inoltre, soffrivano a causa dell'alta complessità del progetto. La *progettazione a livelli* dei sistemi operativi utilizzava il principio dell'astrazione per controllare la complessità della progettazione del SO. Questo tipo di progettazione vedeva il SO come una gerarchia di livelli, in cui ogni livello forniva un insieme di servizi al livello superiore ed esso stesso usava i servizi messi a disposizione dal livello inferiore. Le dipendenze dall'architettura erano spesso ristrette ai livelli più bassi nella gerarchia; tuttavia, non era garantita dalla metodologia di progettazione.

I *sistemi operativi basati su macchina virtuale* (SO VM) supportavano il funzionamento di diversi sistemi operativi su un computer simultaneamente, creando una *virtual machine* per ogni utente e permettendo all'utente di eseguire un SO di sua scelta nella virtual machine. Il SO VM alternava l'esecuzione delle macchine virtuali degli utenti sul computer host attraverso una procedura analoga allo scheduling. Quando una virtual machine veniva schedulata, il suo SO organizzava l'esecuzione delle applicazioni degli utenti in esso attive.

In un progetto di sistema operativo *basato su kernel*, il *kernel* è il nucleo del sistema operativo, che invoca le *routine non kernel* per implementare le operazioni sui processi e sulle risorse. Il codice dipendente dall'architettura in un SO generalmente risiede nel kernel; questa caratteristica aumenta la portabilità di un sistema operativo.

Un *microkernel* è il nucleo essenziale del codice di un SO. È di dimensione ridotta, contiene pochi meccanismi e non contiene nessuna politica. I moduli contenenti le

politiche sono implementati come processi *server*; possono essere cambiati o sostituiti senza coinvolgere il microkernel, fornendo in tal modo elevata espansibilità al SO.

## Domande

- 4.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. I meccanismi di un SO sono tipicamente indipendenti dall'architettura.
  - b. L'organizzazione di un SO a livelli riduce il gap semantico tra i livello più alto del SO e l'hardware del sistema.
  - c. In un SO basato su virtual machine, ogni utente può eseguire un SO di sua scelta.
  - d. La struttura di un SO basato sul kernel fornisce espansibilità.
  - e. In un SO basato su microkernel, il processo scheduler può essere eseguito come processo utente.
- 4.2. Classificare ognuna delle seguenti funzioni effettuate da un SO come politica o meccanismo (fare riferimento ai paragrafi rilevanti dei Capitoli 1 e 3).
  - a. Prelazionare un programma.
  - b. Scheduling a priorità usati nei sistemi multiprogrammati.
  - c. Caricare in memoria un programma scaricato su disco.
  - d. Verificare se un programma utente può avere accesso a un file.
- 4.3. Quale dei seguenti sistemi operativi ha la portabilità più elevata?
  - a. Un SO con struttura monolitica.
  - b. Un SO con struttura a livelli.
  - c. Un SO basato su virtual machine.
  - d. Un SO basato su kernel.

## Problemi

- 4.1. Il meccanismo di scheduling “manipolare le liste di scheduling” (Tabella 4.3) è invocato per modificare le liste di scheduling in risposta agli eventi nel sistema e alle azioni dello scheduler. Descrivere le operazioni che questo meccanismo dovrebbe effettuare per (a) uno scheduling round-robin e (b) uno scheduling basato sulle priorità (come quello usato in un SO multiprogrammato).
- 4.2. Giustificare la seguente affermazione: “Il funzionamento sicuro di un sistema operativo basato su virtual machine non richiede una completa virtualizzazione; tuttavia, può degradare l’efficienza del funzionamento di un SO ospite.”
- 4.3. Quali sono le conseguenze dell'unione tra le routine non kernel con (a) l'interfaccia utente, (b) il kernel? (Sugg.: Si faccia riferimento al Paragrafo 1.1.)
- 4.4. Elencare le differenze tra un kernel che utilizza il caricamento automatico dei moduli e (a) un kernel monolitico e (b) un microkernel.

## Note bibliografiche

Dijkstra (1968) descrive la struttura del sistema multiprogrammato THE. Il sistema operativo basato su virtual machine VM/370 è basato su CP/67 ed è descritto in Creasy (1981). XEN e la virtual machine VMware sono descritti rispettivamente in Barham et al. (2003) e Sugaram et al. (2001). Il volume di maggio 2005 dell'IEEE *Computer* è dedicato alle tecnologie di virtualizzazione. Rosenblum e Garfinkel (2005) inquadrano l'andamento nella progettazione dei monitor delle virtual machine.

Warhol (1994) commenta i passi fatti dai microkernel nei primi anni '90 mentre Liedtke (1996) descrive i principi di progettazione del microkernel. Harting et al. (1997) descrivono il porting e le prestazioni del SO Linux sul microkernel L4. Engler et al. (1995) descrivono la progettazione di un Exokernel. Bach (1986), Vahalia (1996) e McKusick et al. (1996) descrivono il kernel Unix. Beck et al. (2002), Bovet e Cesati (2005) e Love (2005) descrivono il kernel di Linux, mentre Mauro e McDougall (2006) descrivono il kernel di Solaris. Tanenbaum (2001) descrive i microkernel dei sistemi operativi Amoeba e Mach. Russinovich e Solomon (2005) descrivono l'architettura di

Windows.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
2. Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003): "XEN and the art of virtualization," *ACM Symposium on Operating System Principles*, 164-177.
3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
5. Creasy, R.J. (1981): "The origin of the VM/370 time-sharing system," *IBM Journal of Research and Development*, **25** (5), 483-490.
6. Dijkstra, E.W. (1968): "The structure of THE multiprogramming system," *Communications of the ACM*, 11, 341-346.
7. Engler D.R., M.F. Kaashoek, and J. O'Toole (1995): "Exokernel: An operating system architecture for application-level resource management," *Symposium on OS Principles*, 251-266.
8. Hartig, H., M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter (1997): "The performance of microkernel-based systems," *16th ACM Symposium on Operating System Principles*.
9. Liedtke J. (1996): "Towards real microkernels," *Communications of the ACM*, **39** (9), 70-77.
10. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
11. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
12. McKusick, M.K., K. Bostic, M.J. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley, Reading, Mass.
13. Meyer, J., and L.H. Seawright (1970): "A virtual machine time-sharing system," *IBM Systems Journal*, **9** (3), 199-218.
14. Rosenblum, M., and T. Garfinkel (2005): "Virtual machine monitors: current technology and future trends," *IEEE Computer*, **38** (5), 39-47.
15. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
16. Sugarman, J., G. Venkitachalam, and B.H. Lim (2001): "Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor," *2001 USENIX Annual Technical Conference*.
17. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
18. Vahalia, U. (1996): *UNIX Internals—the New Frontiers*, Prentice-Hall, Englewood Cliffs, N.J.
19. Warhol, P.D. (1994): "Small kernels hit it big," *Byte*, January 1994, 119-128.

## PARTE 2

# Gestione dei processi

Un *processo* è un programma in esecuzione. Un'applicazione può essere progettata per avere molti processi che operano concorrentemente e interagiscono tra di loro per ottenere un risultato comune. In questo modo, l'applicazione può essere in grado di fornire una risposta più veloce all'utente.

Un SO mantiene in esecuzione un gran numero di processi per ogni istante di tempo. Gestire i processi significa: crearli, soddisfare le richieste di risorse, *schedularli* per l'uso della CPU, *sincronizzarli* per controllare la loro interazione, evitare *deadlock* in modo che non siano in attesa l'un l'altro indefinitamente e terminarli quando hanno concluso l'esecuzione. Il modo in cui un SO *schedula* i processi per l'utilizzo della CPU determina i tempi di risposta dei processi, l'uso efficiente delle risorse e le prestazioni del sistema.

Un *thread* usa le risorse di un processo ma somiglia a un processo in tutti gli altri aspetti. La gestione dei thread genera meno overhead rispetto alla gestione dei processi. Utilizzeremo il termine *processo* per identificare sia i processi che i thread.

Linee guida per la Parte 2



Diagramma schematico dell'ordine con cui i capitoli di questa parte dovrebbero essere affrontati in un corso.

### Capitolo 5 - Processi e thread

Questo capitolo inizia descrivendo come un'applicazione crea i processi mediante chiamate di sistema e come la presenza di molti processi realizza la concorrenza e il parallelismo all'interno dell'applicazione. Successivamente, descrive come il sistema operativo gestisce un processo, come utilizza la nozione di *stato del processo* per tenere traccia di cosa sta facendo un processo e come gestisce l'effetto di un evento sugli stati dei processi coinvolti. Il capitolo inoltre introduce la nozione di *thread*, descrive i loro benefici e illustra le loro caratteristiche.

### Capitolo 6 - Sincronizzazione dei processi: memoria condivisa

I processi di un'applicazione cooperano per il raggiungimento di un obiettivo comune condividendo dati e coordinandosi tra di loro. I concetti chiave della sincronizzazione dei processi riguardano l'uso della *mutua esclusione* per salvaguardare la

consistenza dei dati condivisi e l'uso delle *operazioni atomiche* per il coordinamento delle attività dei processi. Questo capitolo illustra alcuni problemi classici di sincronizzazione dei processi e come possono essere risolti utilizzando tecniche come i *semafori* e i *monitor* messe a disposizione dai linguaggi di programmazione e dai sistemi operativi.

## **Capitolo 7 - Scheduling**

Lo scheduling è l'azione di selezione del prossimo processo che verrà eseguito dalla CPU. Questo capitolo illustra come lo scheduler utilizza le tecniche fondamentali dello *scheduling a priorità*, del *riordinamento* delle richieste e la modifica della *time slice* per ottenere un'appropriata combinazione di servizio per l'utente, uso efficiente delle risorse e prestazioni del sistema. Inoltre, descrive le diverse politiche di schedulazione e le loro proprietà.

## **Capitolo 8 - Deadlock**

Un deadlock è una situazione in cui ogni processo attivo nel sistema attende indefinitamente che uno o più processi rilascino una risorsa condivisa o si sincronizzino. Questo capitolo discute di come possono verificarsi i deadlock e di come un SO effettua la *gestione dei deadlock* sia mediante tecniche di *rilevamento* e *risoluzione* delle situazioni di deadlock sia attraverso politiche di allocazione delle risorse che *prevengono* o *evitano* il verificarsi di situazioni di deadlock.

## **Capitolo 9 - Sincronizzazione dei processi: message passing**

I processi scambiano informazioni attraverso la *comunicazione interprocesso*. Questo capitolo illustra la semantica dello scambio di messaggi e il ruolo del SO nel bufferizzare e inoltrare i messaggi. Inoltre, discute di come la tecnica del message passing viene utilizzata nei protocolli di alto livello per fornire il servizio di posta elettronica e di comunicazione tra task nei programmi paralleli e distribuiti.

## **Capitolo 10 - Sincronizzazione e scheduling nei sistemi operativi multiprocessore**

La presenza di più CPU nei sistemi multiprocessore consente che le applicazioni ottengano elevati throughput e tempi di risposta ridotti. Questo capitolo illustra i differenti tipi di sistemi multiprocessore e descrive le tecniche di progettazione utilizzate nei SO per ottenere i benefici elencati e per rendere possibile l'esecuzione parallela del codice del kernel, e la sincronizzazione e schedulazione dei processi.

---

# CAPITOLO 5

## Processi e thread

---

### Obiettivi di apprendimento

- Definizione di processo
- Processi e programmi
- Implementazione dei processi: stati, contesto, eventi
- Condivisione, comunicazione e sincronizzazione tra processi: i segnali
- Thread
- Processi e thread nei sistemi operativi: Unix, Linux, Solaris, Windows

Il concetto di *processo* ci aiuta a capire come i programmi vengono eseguiti in un sistema operativo. Un processo è *un programma in esecuzione* che utilizza un insieme di risorse. Abbiamo specificato “un” perché nel sistema operativo possono essere presenti diverse istanze dello stesso programma allo stesso tempo; queste esecuzioni costituiscono differenti processi. Questa situazione si verifica quando gli utenti avviano indipendentemente le esecuzioni dei programmi, ognuno con i propri dati, oppure quando viene eseguito un programma codificato con tecniche di programmazione concorrente. Il kernel alloca le risorse ai processi e li schedula per l'utilizzo della CPU. In questo modo, si realizza l'esecuzione uniforme di programmi sequenziali e concorrenti.

Anche un *thread* è un programma in esecuzione ma viene eseguito nell'ambito di un processo, ovvero utilizza il codice, i dati e le risorse di un processo. È possibile che molti thread vengano eseguiti nell'ambito dello stesso processo; in questo caso condividono il codice, i dati e le risorse del processo. Il context switch dei thread genera meno overhead rispetto al context switch dei processi.

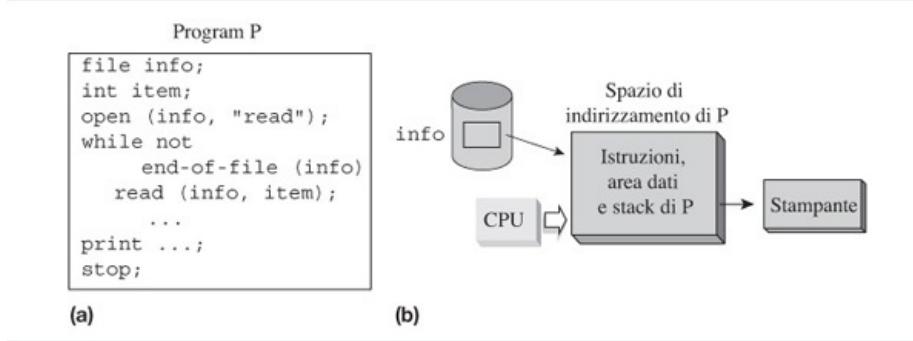
In questo capitolo, spiegheremo come il kernel controlla processi e thread, come tiene traccia dei loro *stati* e come utilizza le informazioni di stato per organizzare la loro esecuzione. Discuteremo inoltre di come un programma può creare processi o thread concorrenti e di come questi possono interagire per raggiungere un obiettivo comune.

### 5.1 Processi e programmi

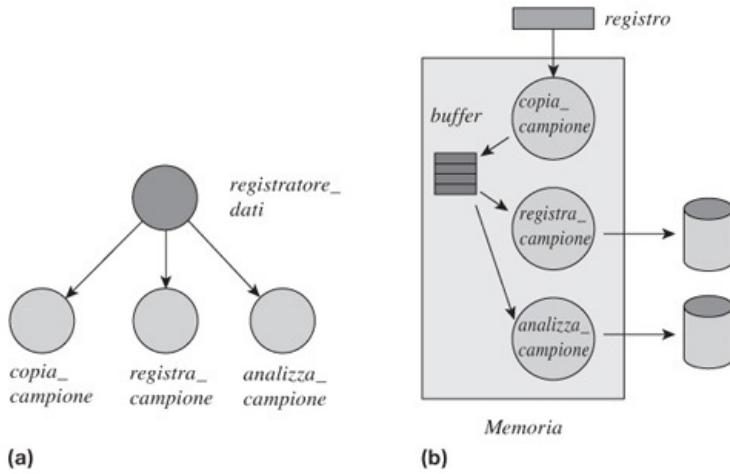
Un programma è un'entità passiva che non effettua nessuna operazione da solo; per effettuare le azioni per cui è stato creato, deve essere eseguito. Un *processo* è un programma in esecuzione. Di fatto esegue le azioni specificate all'interno del programma. Un sistema operativo rende possibile l'esecuzione dei programmi degli utenti schedulando i processi per l'utilizzo della CPU.

#### 5.1.1 Cosa è un processo?

Per comprendere cosa è un processo, bisogna prima capire come il SO esegue i programmi. Il programma P mostrato in [Figura 5.1\(a\)](#) contiene le dichiarazioni di un file `info` e di un variabile `item` e le istruzioni per leggere dei valori da `info`, usarli per effettuare alcune elaborazioni e stampare un risultato prima di terminare l'esecuzione. Durante l'esecuzione, le istruzioni di questo programma usano i valori contenuti nelle aree dati e nello stack per eseguire le elaborazioni. La [Figura 5.2\(b\)](#) mostra una visione astratta dell'esecuzione del programma P. Le istruzioni, i dati e lo stack del programma costituiscono il suo *spazio di indirizzamento*. Per realizzare l'esecuzione del programma P, il SO alloca la memoria per lo spazio di indirizzamento di P, alloca una stampante per stampare i risultati, imposta il meccanismo attraverso cui P accede al file `info` e schedula P per l'esecuzione. La CPU è mostrata come un rettangolo leggermente ombreggiato poiché non esegue sempre le istruzioni di P, il SO condivide la CPU tra l'esecuzione di P e l'esecuzione di altri programmi.



**Figura 5.1** Un programma e una visione astratta della sua esecuzione.



**Figura 5.2** Applicazione real-time del Paragrafo 3.7: (a) albero dei processi; (b) processi.

In base a quanto detto, è possibile definire un processo come segue:

**Definizione 5.1 Processo** Un programma in esecuzione che utilizza le risorse a esso allocate.

Quando un utente avvia l'esecuzione di un programma, il SO crea un nuovo processo e gli assegna un identificativo univoco. Successivamente alloca alcune risorse al processo, una quantità di memoria sufficiente per contenere lo spazio di indirizzamento del programma e alcuni dispositivi come la tastiera e il monitor in modo da consentire l'interazione con l'utente. Il processo può effettuare chiamate di sistema durante l'esecuzione per richiedere ulteriori risorse, per esempio un file. Nel seguito ci riferiremo allo spazio di indirizzamento del programma e allo spazio di indirizzamento a esso allocato, rispettivamente, come spazio di indirizzamento e risorse del processo. Un processo è pertanto caratterizzato da una parte statica (il programma) e una parte dinamica (le risorse che usa).

Nello specifico, un processo comprende sei componenti:

$$(id, codice, dati, stack, risorse, stato della CPU) \quad (5.1)$$

dove *id* è l'identificativo univoco assegnato dal SO

il *codice* è il codice del programma (anche chiamato *testo* del programma)

i *dati* sono i dati usati durante l'esecuzione del programma, inclusi i dati contenuti nei file

lo *stack* contiene i parametri delle funzioni e delle procedure chiamate durante l'esecuzione del programma e i loro indirizzi di ritorno

*risorse* è l'insieme di risorse allocate dal SO

*stato della CPU* è composto dal contenuto del PSW e dai registri general purpose (GPR) della CPU (assumiamo che il puntatore allo stack venga conservato in uno dei GPR)

Lo stato della CPU (Paragrafo 2.2.1) contiene l'indicazione della prossima istruzione da eseguire e altre informazioni, come per esempio il contenuto del campo del *codice di condizione* (anche chiamato il campo *flag*) del PSW, che possono influenzare l'esecuzione del programma. Lo stato della CPU cambia man mano che l'esecuzione del programma progredisce. Utilizziamo il termine *esecuzione di un processo* per intendere l'esecuzione di un programma. Per cui un processo viene eseguito quando è schedulato.

### 5.1.2 Relazioni tra processi e programmi

Un programma consiste di un insieme di funzioni e procedure. Durante l'esecuzione, vengono eseguite le funzioni e le procedure secondo la logica del programma.

L'esecuzione di una funzione o di una procedura è un processo? Questo dubbio porta alla ovvia domanda: qual è la relazione tra processi e programmi?

Il SO non è a conoscenza della natura di un programma, incluse le funzioni e le procedure utilizzate nel codice. È solo a conoscenza delle system call richiamate. Il resto dell'esecuzione è sotto il controllo del programma. In questo modo le funzioni di un programma possono essere processi separati o possono costituire una parte di codice di un singolo processo. Nel seguito verranno mostrati degli esempi di queste situazioni.

La **Tabella 5.1** mostra due tipi di relazioni che possono esistere tra processi e programmi. Una relazione uno a uno esiste quando una singola istanza di un programma sequenziale è in corso, per esempio, l'esecuzione del programma P in **Figura 5.1**. Una relazione uno a molti esiste tra molti processi e un programma in due casi: molte istanze di un programma possono essere in esecuzione allo stesso tempo; i processi che rappresentano queste istanze presentano una relazione molti a uno con il programma. Durante l'esecuzione, un programma può effettuare una system call per richiedere che una parte specifica del suo codice venga eseguita in maniera concorrente, ovvero come una attività separata eseguita simultaneamente ad altre attività. Il kernel imposta l'esecuzione della parte specificata del codice e la considera come un processo separato. Il nuovo processo e il processo che rappresenta l'esecuzione del programma hanno una relazione molti a uno con il programma. Un simile programma viene chiamato *programma concorrente*.

| Relazione   | Esempi   |
|-------------|--|
| Uno a uno   | Una singola esecuzione di un programma sequenziale.                                  |
| Molti a uno | Molte esecuzioni simultanee di un programma, esecuzione di un programma concorrente. |

**Tabella 5.1** Relazioni tra processi e programmi.

I processi che coesistono nel sistema allo stesso momento sono detti *processi concorrenti*. I processi concorrenti possono condividere il codice, i dati e le risorse con altri processi; essi hanno l'opportunità di interagire l'uno con l'altro durante la loro esecuzione.

### 5.1.3 Processi figli

Il kernel avvia l'esecuzione di un programma creando un processo. A causa della mancanza di un termine tecnico, lo chiameremo *processo primario* per l'esecuzione del programma. Il processo primario può effettuare chiamate di sistema, come descritto nel paragrafo precedente, per creare altri processi. Questi processi diventano i suoi *processi figli* e il processo primario diventa il loro *genitore*. Un processo figlio può a sua volta creare altri processi e così via. La relazione genitore-figlio tra questi processi può essere rappresentata come un *albero di processi*, la cui radice è il processo primario. Un processo figlio può ereditare alcune risorse dal suo genitore; potrebbe ottenere ulteriori risorse durante il suo funzionamento utilizzando delle chiamate di sistema.

Solitamente, un processo crea uno o più processi figli e delega a ognuno di loro una parte del suo lavoro. Questa tecnica è chiamata *multitasking* all'interno di

un'applicazione e presenta i tre benefici illustrati nella [Tabella 5.2](#). La creazione di processi figli presenta gli stessi benefici dell'uso della multiprogrammazione in un SO; il kernel può essere in grado di alternare l'esecuzione dei processi I/O-bound e CPU-bound dell'applicazione, cosa che può portare a una riduzione della durata, ovvero del tempo di esecuzione, dell'applicazione.

| Beneficio                                    | Spiegazione  |
|--|--|
| Speedup dell'elaborazione                    | Azioni che il processo primario di un'applicazione avrebbe dovuto effettuare sequenzialmente se non avesse creato i processi figli, sarebbero effettuate concorrentemente quando crea i processi figli. Questo può ridurre la durata dell'applicazione, ovvero il tempo di esecuzione. |
| Priorità per le funzioni critiche            | Un processo figlio che effettua una funzione critica può ottenere una priorità alta; questo può aiutare a soddisfare i requisiti real-time di un'applicazione.   |
| Proteggere un processo genitore dagli errori | Il kernel termina un processo figlio se si verifica un errore durante la sua esecuzione. Il processo genitore non risente dell'errore e può essere in grado di eseguire un'azione di recupero.   |

**Tabella 5.2** Benefici dei processi figli.

Questa caratteristica viene chiamata *speedup dell'elaborazione*. Molti sistemi operativi consentono a un processo genitore di assegnare priorità ai processi figli. Un'applicazione real-time può assegnare una priorità alta a un processo figlio che esegue un funzione critica per assicurare che i suoi requisiti di risposta vengano soddisfatti. Elaboreremo questo aspetto nell'Esempio 5.1.

Il terzo beneficio, ovvero, proteggere un processo genitore dagli errori in un processo figlio, si verifica come segue: si consideri un processo che debba richiamare un codice non verificato. Se questo codice fosse incluso nel codice del processo, un errore nella codifica costringerebbe il kernel a terminare il processo; tuttavia, se il processo creasse un processo figlio per eseguire il codice non verificato, lo stesso errore porterebbe alla terminazione del processo figlio e il processo genitore non subirebbe nessun danno. L'interprete dei comandi del SO è un processo e utilizza questa tecnica quando deve eseguire un comando. In questo modo la sua esecuzione non può essere interrotta dai malfunzionamenti che si verificano nei programmi degli utenti.

L'Esempio 5.1 illustra come il sistema di log dei dati del Paragrafo 3.7 ottenga benefici dall'uso di processi figli.

### Esempio 5.1 Processi figli in un'applicazione real-time

L'applicazione real-time per il log di dati del Paragrafo 3.7 riceve i campioni dei dati da un satellite a un tasso di 500 campioni al secondo e li memorizza in un file. Assumiamo che ogni campione che arriva dal satellite venga memorizzato in un registro speciale del computer. Il processo primario dell'applicazione, che chiameremo processo *registratore\_dati*, deve eseguire le seguenti tre funzioni.

1. Copiare il campione dallo speciale registro in memoria.
2. Copiare il campione dalla memoria in un file.
3. Effettuare alcune analisi del campione e memorizzare il risultato in un altro file usato per elaborazioni future.

Per eseguire queste funzioni, crea tre processi figli chiamati *copia\_campione*, *registra\_campione* e *analizza\_campione*, che costituiscono l'albero dei processi mostrato in [Figura 5.2\(a\)](#). Va notato che un processo è rappresentato da un cerchio e una relazione genitore-figlio è rappresentata da una freccia. Come mostrato in [Figura 5.2\(b\)](#), *copia\_campione* copia il campione dal registro in un'area di memoria chiamata *buffer* che può contenere, per esempio, 50 campioni. *registra\_campione* scrive un campione da *buffer* in un file. *analizza\_campione* analizza un campione da *buffer* e memorizza il risultato in un altro file. L'arrivo di un nuovo campione genera un interrupt e una routine di servizio dell'interrupt definita dal programmatore viene associata a questo interrupt. Il kernel esegue questa routine ogni volta che arriva un nuovo campione, attivando la funzione *copia\_campione*.

L'esecuzione dei tre processi può sovrapporsi come segue: *copia\_campione* può copiare un campione in *buffer*, *registra\_campione* può scrivere nel file un campione precedente, mentre *analizza\_campione* può analizzarlo e scrivere il risultato in un altro file. Questa organizzazione consente di avere, nel caso peggiore, un minore tempo di risposta dell'applicazione rispetto al caso in cui queste funzioni fossero eseguite sequenzialmente. Finché *buffer* ha spazio libero, solo *copia\_campione* deve essere completata prima che arrivi il prossimo campione. Gli altri processi possono essere eseguiti successivamente. Questa possibilità viene sfruttata assegnando la priorità più alta a *copia\_campione*.

Per facilitare l'uso dei processi figli, il kernel fornisce funzioni per:

1. creare un processo figlio e assegnargli una priorità;
2. terminare un processo figlio;
3. determinare lo stato di un processo figlio;
4. condivisione, comunicazione e sincronizzazione dei processi.

Il loro utilizzo può essere descritto come segue: nell'Esempio 5.1, il processo *registratore\_dati* crea tre processi figli. I processi *copia\_campione* e *registra\_campione* condividono *buffer\_area*. È necessario che sincronizzino le rispettive operazioni in modo che il processo *registra\_campione* possa copiare un campione da *buffer\_area* solo dopo che il processo *copia\_campione* ne abbia scritto uno. Il processo *registratore\_dati* potrebbe essere programmato sia per terminare i suoi figli prima che esso stesso termini, sia per concludere la propria esecuzione dopo aver accertato che tutti i suoi processi figli abbiano terminato l'esecuzione.

### 5.1.4 Concorrenza e parallelismo

Il *parallelismo* si riferisce alla caratteristica di verificarsi allo stesso tempo. Due eventi sono paralleli se si verificano allo stesso tempo e due operazioni sono parallele se sono effettuate allo stesso tempo. La *concorrenza* è un'illusione di parallelismo. In questo modo, due task sono concorrenti se c'è l'illusione di eseguirli in parallelo, mentre, in realtà, solo uno può essere eseguito a ogni istante di tempo.

In un SO, la concorrenza è ottenuta alternando l'esecuzione dei processi da parte della CPU, creando l'illusione che questi processi siano eseguiti allo stesso tempo. Il parallelismo è ottenuto usando più CPU, come in un sistema multiprocessore, per eseguire diversi processi contemporaneamente. Come può la semplice concorrenza portare benefici? Abbiamo visto diversi esempi nel [Capitolo 3](#). Nel Paragrafo 3.5 abbiamo discusso di come il throughput di un SO multiprogrammato aumenta alternando l'esecuzione dei processi, poiché un'operazione di I/O in un processo si sovrappone con l'elaborazione di un altro processo. Nel Paragrafo 3.6, abbiamo visto come l'esecuzione alternata di processi creati da differenti utenti in un sistema time-sharing consente a ogni utente di pensare di avere un computer dedicato, sebbene sia più lento del vero computer utilizzato. Nel Paragrafo 5.1.2 e nell'Esempio 5.1, abbiamo visto che l'alternanza dei processi può portare a una elaborazione più veloce.

Il parallelismo può fornire un miglior throughput in modo ovvio poiché i processi possono essere eseguiti su CPU. Inoltre può consentire di velocizzare l'elaborazione; tuttavia, lo speedup dell'elaborazione ottenuto mediante il parallelismo è qualitativamente diverso da quello ottenuto attraverso la concorrenza; quando viene utilizzata la concorrenza, lo speedup è ottenuto sovrapponendo le operazioni di I/O di un processo con le elaborazioni della CPU di altri processi, mentre quando viene utilizzato il parallelismo, le operazioni della CPU e di I/O in un processo si possono sovrapporre con le operazioni della CPU e di I/O di altri processi.

Lo speedup dell'elaborazione di un'applicazione mediante concorrenza e parallelismo dipende da diversi fattori.

- *Parallelismo insito nell'applicazione*: l'applicazione effettua operazioni che possono progredire indipendentemente l'una dall'altra?
- *Overhead dovuto alla concorrenza e al parallelismo*: l'overhead dovuto all'impostazione e alla gestione della concorrenza non dovrebbe predominare sui benefici dovuti all'esecuzione concorrente delle operazioni. Per esempio, se le parti di codice da eseguire concorrentemente sono troppo piccole, l'overhead prodotto dalla concorrenza può portare pochi benefici allo speedup dell'elaborazione.

- *Il modello di concorrenza e parallelismo supportati dal SO:* quanto overhead genera il modello e quanto del parallelismo insito nell'applicazione può essere sfruttato mediante il suo uso.

Finora abbiamo affrontato un modello di concorrenza e parallelismo, ovvero il modello basato sui *processi*. Nel Paragrafo 5.3, introduciamo un modello alternativo detto modello basato sui *thread* e analizziamo le sue proprietà.

## 5.2 Implementazione dei processi

Dal punto di vista del sistema operativo, un processo è una unità di elaborazione. Dunque il compito principale del kernel è di controllare l'esecuzione dei processi per fornire l'utilizzo effettivo del sistema. Di conseguenza, il kernel alloca le risorse a un processo, protegge il processo e le sue risorse dalle interferenze degli altri processi e assicura che il processo ottenga l'uso della CPU finché non completa la sua esecuzione.

Il kernel viene attivato quando un *evento*, ovvero una situazione che richiede l'attenzione del kernel, porta alla generazione di un interrupt hardware oppure a una chiamata di sistema (Paragrafo 2.3). Il kernel a questo punto effettua quattro funzioni fondamentali per controllare il funzionamento dei processi ([Figura 5.3](#)).



**Figura 5.3** Funzioni fondamentali del kernel per il controllo dei processi.

1. *Salvataggio del contesto:* salvare lo stato della CPU e le informazioni riguardanti le risorse del processo la cui esecuzione è stata interrotta.
2. *Gestione dell'evento:* analizzare la condizione che ha portato a un interrupt, o la richiesta da parte di un processo che ha portato a una system call ed effettuare le azioni appropriate.
3. *Scheduling:* selezionare il prossimo processo da eseguire sulla CPU.
4. *Dispatching:* impostare l'accesso alle risorse per il processo schedulato e caricare il suo stato salvato della CPU per iniziare o proseguire l'esecuzione.

Il kernel esegue la funzione di salvataggio del contesto per salvare le informazioni riguardanti il processo interrotto. Successivamente viene eseguita un'appropriata routine di gestione dell'evento, la quale può ritardare ulteriormente l'esecuzione del processo interrotto, per esempio, se l'interrupt fosse causato dal completamento di un'operazione di I/O. Il kernel a questo punto effettua la funzione di schedulazione per selezionare un processo e la funzione di dispatching per iniziare o riprenderne l'esecuzione.

Come discusso precedentemente nei Paragrafi 3.5.1 e 3.6, un sistema operativo per effettuare lo scheduling deve sempre conoscere quali processi richiedono la CPU. Dunque la chiave per controllare l'esecuzione dei processi è di monitorare tutti i processi e conoscere cosa ogni processo sta facendo a ogni istante di tempo, cioè se sta utilizzando la CPU, se è in attesa di ottenere l'uso della CPU, se è in attesa del completamento di un'operazione di I/O o è in attesa per essere riportato in memoria. Il sistema operativo monitora lo *stato del processo* tenendo traccia di cosa ogni processo sta facendo ogni momento.

Nel Paragrafo 5.2 vedremo cosa si intende per stato di un processo, analizzeremo i

vari stati in cui si può trovare un processo e l'organizzazione utilizzata dal sistema operativo per conservare le informazioni relative allo stato dei processi. Lo scheduling verrà affrontato nel [Capitolo 7](#).

### 5.2.1 Stati del processo e transizioni di stato

Un sistema operativo utilizza la nozione di *stato di un processo* per tenere traccia dell'attività di un processo.

**Definizione 5.2 Stato del processo** L'indicatore che descrive la natura dell'attività corrente di un processo.

Il kernel utilizza gli stati del processo per semplificare il suo funzionamento, pertanto il numero degli stati di un processo e i loro nomi possono variare tra i SO. Tuttavia, molti SO usano i quattro stati fondamentali descritti nella [Tabella 5.3](#). Il kernel considera un processo nello stato *bloccato* (blocked) se il processo ha effettuato la richiesta di un risorsa che non è stata ancora soddisfatta, o se è in attesa del verificarsi di un evento. Una CPU non dovrebbe essere allocata a un processo in questo stato finché non termina l'attesa. Il kernel impone lo stato del processo a *pronto* (ready) quando la richiesta è soddisfatta o si è verificato l'evento del quale il processo era in attesa. Questo processo può essere preso in considerazione per la schedulazione. Il kernel impone lo stato del processo a *in esecuzione* (running) quando ne è stato effettuato il dispatch. Lo stato è impostato a *terminato* (terminated) nel momento in cui viene portata a termine l'esecuzione del processo o se, per qualche ragione, il processo viene terminato dal kernel.

| Stato             | Descrizione  |
|-------------------|--|
| <i>Running</i>    | Una CPU sta eseguendo le istruzioni di un processo.  |
| <i>Blocked</i>    | Il processo deve aspettare finché non viene soddisfatta una sua richiesta per una risorsa o finché non si verifica uno specifico evento.     |
| <i>Ready</i>      | Il processo richiede l'uso della CPU per continuare la sua esecuzione; tuttavia, non è stato ancora eseguito il dispatch.                    |
| <i>Terminated</i> | L'esecuzione del processo, ovvero, l'istanza del programma che rappresenta, è stata completata correttamente o è stata terminata dal kernel. |

**Tabella 5.3** Stati fondamentali di un processo.

Un generico computer contiene solo una CPU e di conseguenza al più un processo si può trovare nello stato *running*. Un qualsiasi numero di processi può invece trovarsi negli stati *blocked*, *ready* e *terminated*. Un SO può definire più stati per un processo per semplificare il suo funzionamento o per supportare funzionalità addizionali come lo swapping. Questo aspetto verrà affrontato nel Paragrafo 5.2.1.1.

#### Transizioni di stato dei processi

Una *transizione di stato* per un processo  $P_i$  consiste nel cambiamento di stato. Una transizione di stato è causata dall'occorrenza di qualche evento come l'inizio o la fine di un'operazione di I/O. Quando si verifica l'evento, il kernel determina il suo effetto sulle attività dei processi e modifica di conseguenza lo stato dei processi interessati.

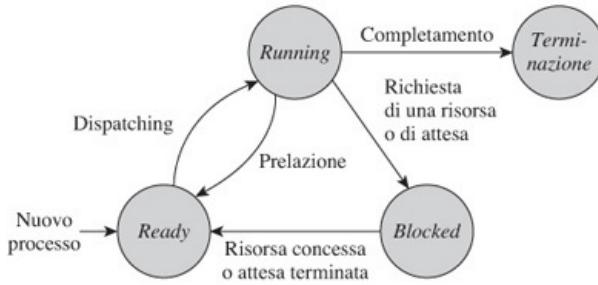
Quando un processo  $P_i$  che si trova nello stato *running* effettua una richiesta per un'operazione di I/O, il suo stato deve essere impostato a *blocked* finché l'operazione non viene completata. Al termine dell'operazione di I/O, lo stato di  $P_i$  viene portato da *blocked* a *ready* poiché ora richiede l'uso della CPU. Simili modifiche allo stato si verificano quando un processo effettua qualche richiesta che non può essere soddisfatta immediatamente dal SO. Lo stato del processo viene impostato a *blocked* al momento della richiesta, ovvero quando si verifica l'evento della richiesta ed è portato a *ready* quando la richiesta è stata soddisfatta. Lo stato di un processo *ready* viene modificato a *running* quando si procede al dispatching. Lo stato di un processo *running* è portato a *ready* quando il processo è prelazionato oppure se un processo a più alta priorità passa

nello stato *ready* oppure se il processo ha utilizzato tutto il tempo di CPU messogli a disposizione dal SO (Paragrafi 3.5.1 e 3.6). La [Tabella 5.4](#) riassume le cause delle transizioni di stato.

| Transizione di stato               | Descrizione  |
|------------------------------------|--|
| <i>ready</i> → <i>running</i>      | Il processo è sottoposto a dispatching. La CPU comincia e continua l'esecuzione delle sue istruzioni.  |
| <i>blocked</i> → <i>ready</i>      | Una richiesta fatta da un processo è soddisfatta o si verifica un evento di cui il processo era in attesa.   |
| <i>running</i> → <i>ready</i>      | Il processo viene prelazionato poiché il kernel decide di schedulare un altro processo. Questa transizione si verifica o perché un processo a più alta priorità passa nello stato <i>ready</i> , o perché la time-slice del processo è terminata.  |
| <i>running</i> → <i>blocked</i>    | Il processo in esecuzione effettua una chiamata di sistema per ottenere l'uso di una risorsa o è in attesa che si verifichi un evento. Le cinque maggiori cause di blocco di un processo sono: <ul style="list-style-type: none"> <li>• il processo richiede un'operazione di I/O;</li> <li>• il processo richiede una risorsa;</li> <li>• il processo deve essere interrotto per un determinato intervallo di tempo;</li> <li>• il processo è in attesa di un messaggio da un altro processo;</li> <li>• il processo è in attesa di qualche azione da parte di altri processi.</li> </ul>   |
| <i>running</i> → <i>terminated</i> | L'esecuzione del programma è completata. Le cinque ragioni principali per la terminazione di un processo sono: <ul style="list-style-type: none"> <li>• <i>auto-terminazione</i>: il processo in esecuzione completa il suo compito o realizza che non può continuare in maniera consistente ed effettua una system call per essere terminato. La seconda situazione si presenta a causa di dati incorretti o inconsistenti o a causa dell'impossibilità di accedere ai dati nel modo richiesto, per esempio perché il processo non ha i permessi necessari;</li> <li>• <i>terminazione richiesta dal processo genitore</i>: un processo effettua una chiamata di sistema per terminare un processo figlio <math>P_i</math>, quando rileva che l'esecuzione del processo figlio non è più necessaria o consistente;</li> <li>• <i>ccesso di utilizzo di una risorsa</i>: un SO può limitare le risorse a disposizione di un processo. Un processo che eccede un limite imposto a un risorsa viene terminato dal kernel;</li> <li>• <i>condizioni anomale durante l'esecuzione</i>: il kernel termina un processo se l'esecuzione dell'istruzione corrente causa una condizione anomala, quale per esempio l'esecuzione di un'istruzione non valida, l'esecuzione di un'istruzione privilegiata, condizioni aritmetiche come l'overflow o la violazione della protezione della memoria;</li> <li>• <i>interazione non corretta con altri processi</i>: il kernel può terminare un processo coinvolto in un deadlock.</li> </ul> |

**Tabella 5.4** Cause delle fondamentali transizioni di stato per un processo.

La [Figura 5.4](#) mostra le fondamentali transizioni di stato per un processo. Un nuovo processo è portato nello stato *ready* dopo che le risorse richieste sono state allocate. Può poi entrare negli stati *running*, *blocked* e *ready* un certo numero di volte come risultato degli eventi descritti nella [Tabella 5.4](#). A un certo punto entra nello stato *terminated*.



**Figura 5.4** Transizioni di stato fondamentali per un processo.

### Esempio 5.2 Transizioni di stato del processo

Si consideri il sistema time-sharing dell'Esempio 3.2, che utilizza una time slice di 10 ms. Nel sistema sono attivi due processi  $P_1$  e  $P_2$ .  $P_1$  ha un CPU burst di 15 ms seguito da un'operazione di I/O che dura 100 ms, mentre  $P_2$  ha un CPU burst di 30 ms seguito da un'operazione di I/O che dura 60 ms. L'esecuzione di  $P_1$  e  $P_2$  è stata descritta nella [Figura 3.7](#). La [Tabella 5.5](#) illustra le transizioni di stato durante il funzionamento del sistema. L'esecuzione dei programmi procede come segue: al tempo 0 entrambi i processi sono nello stato *ready*. Lo scheduler seleziona il processo  $P_1$  per l'esecuzione e impone il suo stato a *running*. A 10 ms,  $P_1$  viene prelazionato e  $P_2$  è sottoposto a dispatch. Dunque lo stato di  $P_1$  è impostato a *ready* e lo stato di  $P_2$  è impostato a *running*. A 20 ms,  $P_2$  viene prelazionato e  $P_1$  è mandato in esecuzione. A 25 ms,  $P_1$  entra nello stato *blocked* a causa dell'operazione di I/O. Viene effettuato il dispatch di  $P_2$  poiché si trova nello stato *ready*. A 35 ms,  $P_2$  viene prelazionato poiché la time slice è terminata; tuttavia viene effettuato nuovamente il dispatch di  $P_2$  poiché non ci sono altri processi nello stato *ready*.  $P_2$  avvia un'operazione di I/O a 45 ms. Ora entrambi i processi sono nello stato *blocked*.

| Temo | Evento               | Considerazioni     | Nuovo stato        |         |
|------|----------------------|--------------------|--------------------|---------|
|      |                      |                    | $P_1$              | $P_2$   |
| 0    |                      | $P_1$ è schedulato | running            | ready   |
| 10   | $P_1$ è prelazionato | $P_2$ è schedulato | ready              | running |
| 20   | $P_2$ è prelazionato | $P_1$ è schedulato | running            | ready   |
| 25   | $P_1$ avvia I/O      | $P_2$ è schedulato | blocked            | running |
| 35   | $P_2$ è prelazionato | –                  | blocked            | ready   |
|      |                      |                    | $P_2$ è schedulato | blocked |
| 45   | $P_2$ avvia I/O      | –                  | blocked            | blocked |

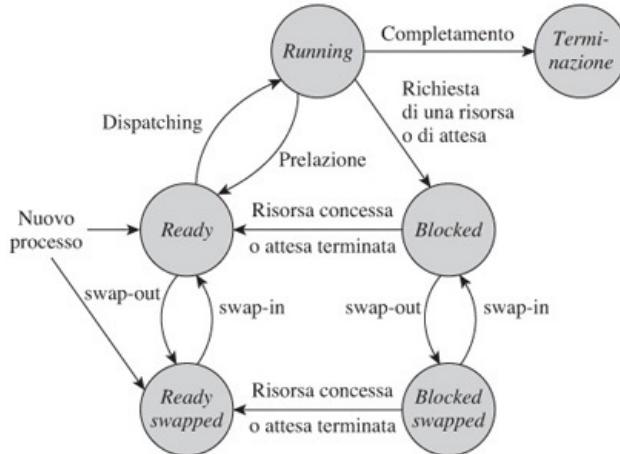
**Tabella 5.5** Transizioni di stato dei processi in un sistema time-sharing.

### Processi sospesi

Un kernel deve prevedere stati addizionali per descrivere la natura dell'attività di un processo che non si trova in uno dei quattro stati fondamentali descritti precedentemente. Si consideri un processo che si trovava nello stato *ready* o nello stato *blocked* prima di essere swappato sul disco. Il processo deve essere riportato in memoria prima che possa riprendere l'esecuzione, per cui il kernel deve definire un nuovo stato. Indicheremo questo processo come *sospeso*. Se un utente specifica che il suo processo non deve essere considerato schedulabile per un certo periodo di tempo, anche in questo caso si parla di processo sospeso. Quando un processo sospeso deve riprendere l'attività interrotta, deve essere riportato allo stato in cui si trovava nel momento in cui era stato sospeso. Per facilitare questa transizione di stato, il kernel può definire più stati *sospesi* e impostare il processo nello stato sospeso appropriato.

Restriniamo la discussione dei processi sospesi ai processi swappati e usiamo per la sospensione due stati chiamati *ready swapped* e *blocked swapped*. La [Figura 5.5](#) mostra gli stati del processo e le transizioni di stato conformi a questa nuova organizzazione. La

transizione *ready* → *ready swapped* o *blocked* → *blocked swapped* è causata da un'azione di swap-out. La transizione di stato inversa si verifica quando il processo è riportato in memoria. La transizione *blocked swapped* → *ready swapped* si verifica se viene soddisfatta la richiesta per cui il processo è in attesa, anche nel caso in cui il processo sia sospeso, per esempio, se viene concesso l'uso di una risorsa. Tuttavia, il processo continua a rimanere sul disco. Quando viene riportato in memoria, il suo stato è impostato a *ready* e compete per l'utilizzo della CPU con gli altri processi *ready*. Un nuovo processo è impostato nello stato *ready* o nello stato *ready swapped* in base alla disponibilità di memoria.



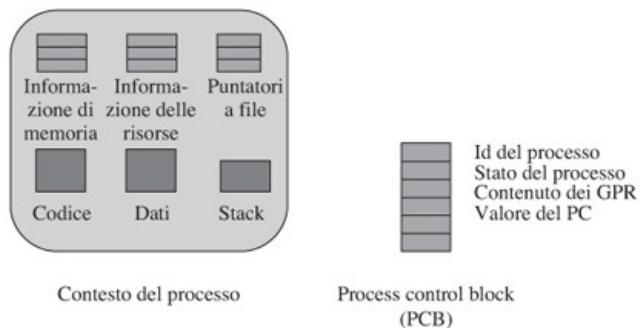
**Figura 5.5** Stati del processo e transizioni di stato utilizzando due stati per lo swap.

### 5.2.2 Il contesto del processo e il process control block

Il kernel alloca le risorse al processo e lo schedula per l'utilizzo della CPU. Di conseguenza, il kernel vede il processo composto di due parti:

- Codice, dati e stack del processo e le informazioni che riguardano la memoria e le altre risorse a esso allocate, come per esempio i file.
- Le informazioni riguardanti l'esecuzione del programma, come lo stato del processo, lo stato della CPU, incluso il puntatore allo stack, e altre informazioni che verranno descritte successivamente in questo paragrafo.

Queste due parti sono contenute, rispettivamente, nel *contesto del processo* e nel *process control block* (PCB) (Figura 5.6). Questa organizzazione consente ai diversi moduli del kernel di accedere, in modo conveniente ed efficiente, alle informazioni rilevanti relative al processo.



**Figura 5.6** Visione di un processo da parte del kernel.

#### Contesto del processo

Il contesto del processo si compone delle seguenti parti.

1. *Spazio di indirizzamento del processo*: codice, dati e stack del processo (Definizione 5.1).
2. *Informazione di allocazione della memoria*: informazione relativa all'area di memoria allocata al processo. Questa informazione viene utilizzata dall'unità per la gestione della memoria (memory management unit - MMU) durante l'esecuzione del processo (Paragrafo 2.2.2).
3. *Stato delle attività di elaborazione del file*: informazione relativa ai file usati, come le posizioni correnti nei file.
4. *Informazione relativa all'interazione col processo*: informazione necessaria per controllare l'interazione del processo con gli altri processi, per esempio l'id del processo genitore e dei processi figli e i messaggi inviati al processo che non sono ancora stati consegnati.
5. *Informazione relativa alle risorse*: informazione riguardante le risorse allocate al processo.
6. *Informazioni varie*: varie informazioni necessarie al funzionamento di un processo.

Il SO crea un contesto allocando memoria al processo, caricando il codice del processo nella memoria allocata e impostando il suo spazio dati. L'informazione riguardante le risorse allocate al processo e la sua interazione con altri processi è mantenuta nel contesto del processo durante il suo ciclo di vita. Questa informazione cambia in base all'esito delle operazioni eseguite, come per esempio l'apertura e la chiusura dei file o la creazione e la cancellazione dei dati effettuati dal processo durante l'esecuzione.

### **Process Control Block (PCB)**

Il *Process Control Block* (PCB) di un processo contiene tre tipi di informazione riguardanti il processo: l'informazione relativa all'identificazione, come per esempio l'id del processo, l'id del processo genitore e l'id dell'utente che lo ha creato; l'informazione circa lo stato del processo, ovvero il suo stato e il contenuto del PSW e dei registri general purpose (GPR); l'informazione necessaria per controllare la sua esecuzione, cioè la sua priorità e l'interazione con gli altri processi. Inoltre contiene un campo puntatore usato dal kernel per creare la lista dei PCB necessaria per la schedulazione (la lista dei processi *ready*). La [Tabella 5.6](#) descrive i campi della struttura dati del PCB.

| <b>Campo del PCB</b>        | <b>Contenuto</b>  |
|-----------------------------|---|
| Id del processo             | L'id univoco assegnato al processo al momento della creazione.  |
| Id del genitore e dei figli | Questi id sono usati per la sincronizzazione dei processi, tipicamente per consentire a un processo di verificare se un processo figlio ha terminato la sua esecuzione.   |
| Priorità                    | La priorità è generalmente un valore numerico. Al momento della creazione, al processo viene assegnata una priorità. Il kernel può cambiare la priorità dinamicamente in base alla natura del processo (CPU-bound o I/O-bound), al tempo di attività e alle risorse utilizzate (solitamente il tempo di CPU). |
| Stato del processo          | Lo stato corrente del processo.   |
| PSW                         | Questa è un'istantanea, ovvero un'immagine del PSW eseguita l'ultima volta che il processo è stato bloccato o prelazionato. Il caricamento di questa immagine nel PSW ripristina l'esecuzione del processo (vedi <a href="#">Figura 2.2</a> per i campi del PSW).   |
| GPR                         | Il contenuto dei registri general purpose salvati l'ultima volta che il processo è stato bloccato o prelazionato.   |
| Informazione sugli eventi   | Per un processo nello stato <i>blocked</i> , questo campo contiene l'informazione relativa all'evento per il quale il processo è in attesa.   |
| Informazioni sui segnali    | Informazione relativa ai gestori dei segnali (vedi Paragrafo 5.2.6).  |

Puntatore al PCB

Questo campo è utilizzato per creare una lista di PCB, necessaria per la schedulazione.

**Tabella 5.6** Campi del PCB.

Le informazioni relative alla priorità e allo stato vengono utilizzate dallo scheduler, che passa l'id del processo selezionato al dispatcher. Per un processo che non si trova nello stato *running*, il PSW e i campi del GPR insieme contengono lo *stato della CPU* del processo, salvato l'ultima volta che il processo è stato bloccato o prelazionato (Paragrafo 2.2.1). L'esecuzione del processo può essere ripristinata semplicemente caricando queste informazioni dal PCB nella CPU. Questa azione viene eseguita quando il processo è sottoposto a dispatching.

Quando un processo passa allo stato *blocked*, è importante memorizzarne la causa. Ciò è possibile annotando la causa del blocco, come per esempio la richiesta di una risorsa o un'operazione di I/O, nel campo relativo alle *informazioni degli eventi* nel PCB. Si consideri un processo  $P_i$ , che è bloccato per un'operazione di I/O.

Il campo relativo alle *informazioni degli eventi* del PCB indica che il processo è in attesa del completamento dell'operazione di I/O sul dispositivo  $d$ . Quando l'operazione di I/O sul dispositivo  $d$  viene completata, il kernel utilizza questa informazione per effettuare la transizione  $blocked \rightarrow ready$  per il processo  $P_i$ .

### 5.2.3 Salvataggio del contesto, scheduling e dispatching

Al verificarsi di un evento, la funzione di salvataggio del contesto ne effettua la gestione. Salva lo stato della CPU del processo interrotto nel PCB e salva l'informazione relativa al contesto (Paragrafo 5.2.2). Va ricordato che il processo interrotto si trovava nello stato *running* prima che si verificasse l'evento. La funzione di salvataggio del contesto imposta lo stato a *ready*. Il gestore dell'evento può successivamente cambiare lo stato del processo a *blocked*, per esempio se l'evento corrente era una richiesta di un'operazione di I/O effettuata dallo stesso processo interrotto.

La funzione di scheduling utilizza le informazioni di stato del processo salvate nel PCB per selezionare un processo *ready* per l'esecuzione e passa il suo id alla funzione di dispatching. La funzione di dispatching imposta il contesto del processo selezionato, modifica il suo stato a *running* e carica lo stato della CPU salvato dal PCB nella CPU. Per evitare problemi di protezione, ripulisce i buffer per la traduzione degli indirizzi utilizzati dalla memory management unit (MMU). L'Esempio 5.3 illustra le funzioni di salvataggio del contesto, di scheduling e dispatching in un SO che utilizza uno scheduler basato sulle priorità.

#### Esempio 5.3 Salvataggio del contesto, scheduling e dispatching

In un SO sono attivi due processi  $P_1$  e  $P_2$ , con  $P_2$  che ha una priorità più alta di  $P_1$ . Sia  $P_2$  *bloccato* su un'operazione di I/O e sia  $P_1$  nello stato *running*. Le seguenti azioni avvengono quando si verifica l'evento relativo al completamento dell'operazione di I/O del processo  $P_2$ .

1. La funzione di salvataggio del contesto viene effettuata per  $P_1$  e il suo stato è impostato a *ready*.
2. Usando il campo del PCB relativo alle *informazioni dell'evento*, il gestore dell'evento rileva che l'operazione di I/O era stata avviata da  $P_2$ , per cui modifica lo stato di  $P_2$  da *blocked* a *ready*.
3. Viene effettuato lo scheduling.  $P_2$  è selezionato poiché è il processo *ready* a più alta priorità.
4. Lo stato di  $P_2$  viene impostato a *running* e sottoposto a dispatching.

#### Commutazione tra processi e di contesto (context switch)

Le funzioni 1, 3 e 4 dell'Esempio 5.3 collettivamente eseguono la commutazione tra i processi  $P_1$  e  $P_2$ . La commutazione tra i processi si verifica anche quando un processo nello stato *running* passa nello stato *blocked* in seguito a una richiesta o nel caso in cui venga prelazionato al termine della time slice. Un evento non conduce alla commutazione tra processi se l'occorrenza dell'evento o (1) causa una transizione di stato solo in un

processo la cui priorità è inferiore a quella del processo la cui esecuzione è stata interrotta dall'evento oppure (2) non causa nessuna transizione di stato, per esempio se l'evento è causato da una richiesta che è immediatamente soddisfatta. Nel primo caso, la funzione di scheduling seleziona il processo interrotto per il dispatching. Nel secondo caso, lo scheduling non viene effettuato affatto; la funzione di dispatching potrebbe semplicemente cambiare lo stato del processo interrotto nuovamente a *running* ed effettuarne il dispatch.

La commutazione tra processi necessita di più azioni rispetto al salvataggio dello stato della CPU di un processo e il caricamento dello stato della CPU di un altro processo.

Anche il contesto del processo ha bisogno di essere commutato. Si utilizza il termine *informazione di stato di un processo* per indicare tutte le informazioni che è necessario salvare e ripristinare durante la commutazione tra processi. L'overhead dovuto alla commutazione tra processi dipende dalla dimensione dell'informazione di stato del processo. Alcuni sistemi mettono a disposizione speciali istruzioni per ridurre l'overhead dovuto alla commutazione, per esempio istruzioni che salvano o caricano il PSW e tutti i registri general purpose, o che ripuliscono i buffer per la traduzione degli indirizzi utilizzati della memory management unit (MMU).

La commutazione tra processi ha anche dell'overhead indiretto. Il nuovo processo schedulato può non avere nessuna parte del suo spazio di indirizzamento nella cache, per cui le sue prestazioni sarebbero scarse finché non riesce a portare in cache informazioni sufficienti (Paragrafo 2.2.3). Anche il funzionamento della memoria virtuale inizialmente non è ottimale poiché i buffer per la traduzione degli indirizzi nella MMU non contengono nessuna informazione rilevante per il nuovo processo schedulato.

## 5.2.4 Gestione degli eventi

I seguenti eventi si verificano durante il funzionamento di un SO.

1. *Evento per la crezione di un processo*: viene creato un nuovo processo.
2. *Evento per la terminazione di un processo*: un processo termina la sua esecuzione.
3. *Evento timer*: si verifica un interrupt del timer.
4. *Evento per la richiesta di una risorsa*: un processo effettua la richiesta per una risorsa.
5. *Evento per il rilascio di una risorsa*: un processo rilascia una risorsa.
6. *Evento per la richiesta di avvio di I/O*: un processo richiede l'avvio di un'operazione di I/O.
7. *Evento per il completamento di I/O*: un'operazione di I/O è completata.
8. *Evento per l'invio di un messaggio*: un processo invia un messaggio a un altro processo.
9. *Evento per la ricezione di un messaggio*: un processo riceve un messaggio.
10. *Evento per l'invio di un segnale*: un processo invia un segnale a un altro processo.
11. *Evento per la ricezione di un segnale*: un processo riceve un segnale.
12. *Un interrupt da programma*: l'istruzione corrente nel processo *running* non è stata eseguita correttamente.
13. *Evento per il malfunzionamento dell'hardware*: una unità hardware del computer ha causato un malfunzionamento.

Gli eventi relativi al timer, al completamento del I/O e al malfunzionamento dell'hardware sono causati da situazioni esterne al processo *running*. Tutti gli altri eventi sono causati da azioni nell'ambito del processo. Per analizzare le azioni dei gestori degli eventi divideremo gli eventi 1-9 in due grandi classi, mentre gli eventi 10 e 11 sono affrontati nel Paragrafo 2.5.6. L'azione di default effettuata dal kernel, quando si verificano gli eventi 12 e 13, è la terminazione del processo *running*.

### Eventi relativi alla creazione, terminazione e prelazione del processo

Quando un utente lancia un comando per eseguire un programma, l'interprete dei comandi effettua la chiamata di sistema *create\_process*, passando come argomento il nome del programma. Quando un processo vuole creare un figlio per eseguire un programma, effettua la chiamata di sistema *create\_process* passando come parametro il nome del programma.

La routine di gestione dell'evento collegata alla system call *create\_process* crea un PCB per il nuovo processo, gli assegna un id univoco, una priorità e inserisce queste

informazioni e l'id del processo genitore negli opportuni campi del PCB. Successivamente determina la quantità di memoria necessaria per lo spazio di indirizzamento del processo, ovvero il codice e i dati del programma da eseguire e il suo stack e alloca questa quantità di memoria al processo (le tecniche di allocazione della memoria sono discusse nei Capitoli 11 e 12). In molti sistemi operativi, ad ogni processo sono assegnate delle risorse standard, per esempio, una tastiera, un file per lo standard input e un file per lo standard output; il kernel alloca queste risorse al processo durante la fase di creazione. La fase successiva consiste nell'inserire, nel contesto del nuovo processo, le informazioni relative alla memoria allocata e alle risorse. Infine, il kernel imposta lo stato del processo a *ready* e inserisce il processo nella lista appropriata di PCB.

Quando un processo effettua una system call per terminare la propria esecuzione o quella di un processo figlio, il kernel ritarda questo evento finché non vengono completate le operazioni di I/O avviate dal processo. A questo punto rilascia la memoria e le risorse allocate.

Questa funzione è svolta utilizzando le informazioni contenute nel contesto del processo. Infine imposta lo stato del processo a *terminated*. Il processo genitore può richiedere il controllo dello stato di uscita del processo figlio. Per questo motivo il PCB non viene distrutto immediatamente ma solo dopo che il genitore ne ha recuperato lo stato o dopo che esso stesso ha terminato la sua esecuzione. Se il genitore del processo è in attesa della terminazione del figlio, il kernel deve attivare il processo genitore. A tal fine, il kernel legge l'id del genitore nel PCB del processo terminato e verifica nel PCB del processo genitore se quest'ultimo è in attesa della terminazione del processo figlio (Paragrafo 5.2.2).

Il processo nello stato *running* dovrebbe essere prelazionato nel caso in cui la time slice fosse esaurita. La funzione di salvataggio del contesto avrebbe già impostato lo stato del processo in esecuzione a *ready* prima di invocare il gestore dell'evento per gli interrupt generati da timer, così che il gestore dell'evento potrebbe semplicemente inserire il PCB del processo nell'appropriata lista di PCB. La prelazione dovrebbe essere anche attuata quando un processo a più alta priorità passa allo stato *ready*, ma ciò si realizza implicitamente quando il processo ad alta priorità viene schedulato. In questo modo il gestore dell'evento non deve eseguire un'azione esplicita.

### **Eventi relativi all'utilizzo delle risorse**

Quando un processo richiede un risorsa attraverso una system call, il kernel può essere in grado di allocare la risorsa immediatamente, nel qual caso il gestore dell'evento non causa nessuna transizione di stato e il kernel può evitare la schedulazione e invocare direttamente il dispatcher per ripristinare l'esecuzione del processo interrotto. Se la risorsa non può essere allocata, il gestore dell'evento modifica lo stato del processo interrotto a *blocked* e annota l'id della risorsa richiesta nel campo delle *informazioni relative agli eventi* del PCB. Quando un processo rilascia un risorsa attraverso una chiamata di sistema, il gestore dell'evento non deve cambiare lo stato del processo che ha effettuato la system call. Tuttavia, dovrebbe controllare se qualche altro processo era bloccato in attesa della stessa risorsa e, nel caso, dovrebbe allocare la risorsa a uno dei processi bloccati, impostando il suo stato a *ready*. Questa operazione richiede una speciale organizzazione che discuteremo brevemente.

Le chiamate di sistema per richiedere di avviare un'operazione di I/O e per segnalare mediante interrupt la fine di un'operazione di I/O comportano un'analogia sequenza di azioni per la gestione dell'evento. Quando si avvia l'operazione di I/O, lo stato del processo viene impostato a *blocked* e la causa del blocco viene annotata nel campo delle *informazioni relative agli eventi* del suo PCB; lo stato del processo viene riportato a *ready* quando l'operazione di I/O è completata. Una richiesta di ricezione di un messaggio da un altro processo e una richiesta di invio di un messaggio a un altro processo portano ad azioni analoghe.

### **Event Control Block (ECB)**

Quando si verifica un evento, il kernel deve trovare il processo il cui stato è influenzato dall'evento. Per esempio, quando si verifica un interrupt in seguito al completamento di un'operazione di I/O, il kernel deve identificare il processo in attesa dell'evento di completamento.

L'operazione è effettuata mediante un ricerca nei campi delle *informazioni relative agli eventi* dei PCB dei processi. Questa ricerca è costosa in termini di tempo, per cui i sistemi operativi utilizzano vari schemi per velocizzarla. Uno di questi consiste nell'utilizzo dell'*Event Control Block* (ECB).

Come mostrato in [Figura 5.7](#), un ECB contiene tre campi. Il campo *descrizione dell'evento* descrive un evento e il campo *id del processo* contiene l'id del processo in attesa dell'evento. Quando un processo  $P_i$  si blocca a causa del verificarsi di un evento  $e_i$ , il kernel crea un ECB e vi inserisce le informazioni rilevanti riguardanti  $e_i$  e  $P_i$ . Il kernel può mantenere una lista separata degli ECB per ogni classe di evento, come i messaggi interprocesso o le operazioni di I/O, in modo tale che il campo *puntatore a ECB* venga utilizzato per inserire l'ECB appena creato nella lista appropriata degli ECB.

---

|                         |
|-------------------------|
| Descrizione dell'evento |
| Id del processo         |
| Puntatore all'ECB       |

---

**Figura 5.7** Event Control Block (ECB).

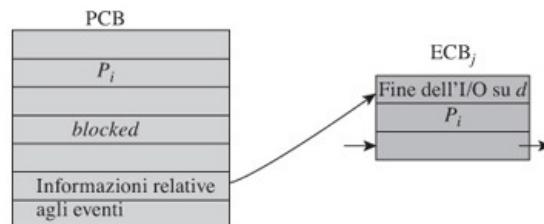
Quando si verifica un evento, il kernel esamina la lista degli ECB appropriata per trovare un ECB con una descrizione dell'evento corrispondente. Il campo *id del processo* dell'ECB indica quale processo è in attesa di un evento. Lo stato di questo processo viene modificato come conseguenza dell'occorrenza dell'evento. L'esempio seguente illustra l'uso degli ECB per la gestione di un evento relativo al completamento di un'operazione di I/O; il loro utilizzo per la gestione dei messaggi interprocesso è descritto nel [Paragrafo 9.2.2](#).

Il campo delle *informazioni relative agli eventi* del PCB a questo punto è ridondante; tuttavia, lo si conserva perché il kernel potrebbe avere bisogno di sapere su quale evento un processo è bloccato, per esempio, nel caso in cui dovesse terminare il processo.

#### Esempio 5.4 Utilizzo dell'ECB per la gestione del completamento dell'I/O

Di seguito sono riportate le operazioni eseguite dal kernel quando un processo  $P_i$  richiede un'operazione di I/O su qualche dispositivo  $d$  e quando l'operazione di I/O viene completata.

1. Il kernel crea un ECB e lo inizializza come segue.
  - a. Descrizione dell'evento := fine dell'I/O sul dispositivo  $d$ .
  - b. Processo in attesa dell'evento :=  $P_i$ .
2. L'ECB appena creato (per esempio  $ECB_j$ ) viene aggiunto alla lista degli ECB.
3. Lo stato di  $P_i$  viene impostato a *blocked* e l'indirizzo dell' $ECB_j$  è copiato nel campo "Informazioni relative agli eventi" del PCB relativo a  $P_i$  (vedi [Figura 5.8](#)).

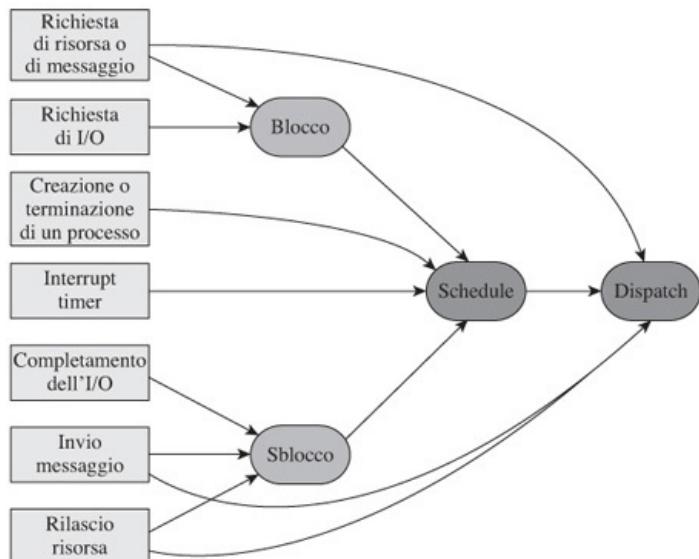


**Figura 5.8** Relazione PCB-ECB.

4. Quando viene generato l'interrupt 'Fine dell'I/O sul dispositivo  $d$ ',  $ECB_j$  viene ritrovato cercando un ECB con un campo di descrizione dell'evento corrispondente.
5. L'id del processo coinvolto, per esempio  $P_i$  viene estratto dall' $ECB_j$ . Il PCB del processo  $P_i$  viene ritrovato e il suo stato impostato a *ready*.

### Riassunto della gestione dell'evento

La Figura 5.9 illustra le azioni del kernel per la gestione dell'evento descritte precedentemente. L'azione *block* modifica sempre lo stato del processo chiamante da *ready* a *blocked*. L'azione *unblock* cerca un processo la cui richiesta può essere soddisfatta e cambia il suo stato da *blocked* a *ready*. Una chiamata di sistema per richiedere una risorsa implica un'azione di *block* se la risorsa non può essere allocata al processo che la richiede. Questa azione viene seguita dallo scheduling e dal dispatching poiché deve essere selezionato un altro processo per utilizzare la CPU. L'azione di blocco non viene effettuata se la risorsa può essere allocata direttamente. In questo caso, il processo interrotto è semplicemente sottoposto a nuovo dispatching. Quando un processo rilascia una risorsa, nel caso in cui un altro processo sia in attesa della risorsa rilasciata, viene eseguita un'azione di *unblock* seguita da scheduling e dispatching, poiché il processo sbloccato può avere una priorità maggiore del processo che ha rilasciato la risorsa. Anche in questo caso, lo scheduling non viene effettuato se in conseguenza dell'evento nessun processo è stato sbloccato.



**Figura 5.9** Azioni del kernel per la gestione dell'evento.

### 5.2.5 Condivisione, comunicazione e sincronizzazione tra processi

I processi di un'applicazione hanno bisogno di interagire l'uno con l'altro per un obiettivo comune. La Tabella 5.7 descrive quattro tipi di interazione tra processi. Riassumiamo di seguito le loro caratteristiche più importanti.

| <b>Tipo di interazione</b> | <b>Descrizione</b>   |
|----------------------------|--|
| Condivisione dei dati      | I dati condivisi possono diventare inconsistenti se diversi processi li modificano allo stesso tempo. Per questo motivo i processi devono interagire per decidere quando è possibile per un processo modificare o usare i dati condivisi in modo sicuro. |
| Scambio di messaggi        | I processi scambiano informazioni inviando messaggi l'uno all'altro.   |
| Sincronizzazione           | Per ottenere un obiettivo comune, i processi devono coordinare le proprie attività.  |
| Segnali                    | Un segnale è usato per comunicare a un processo l'occorrenza di una situazione di eccezione.   |

**Tabella 5.7** Quattro tipi di interazione tra processi.

### **Condivisione dei dati**

Una variabile condivisa può contenere valori inconsistenti se molti processi la aggiornano in maniera concorrente. Se due processi eseguono l'istruzione  $a:=a+1$ , dove  $a$  rappresenta una variabile condivisa, il risultato può dipendere da come il kernel alterna la loro esecuzione, per esempio, il valore di  $a$  può essere incrementato solo di 1 (discuteremo questo problema successivamente nel Paragrafo 6.2). Per evitare questo problema, solo un processo alla volta dovrebbe avere accesso alla variabile condivisa a ogni istante di tempo, per cui l'accesso al dato da parte di un processo deve essere ritardato se un altro processo sta accedendo allo stesso dato. Questa tecnica è chiamata *mutua esclusione*. In questo modo, la condivisione dei dati da parte di processi concorrenti genera un overhead dovuto alla mutua esclusione.

### **Scambio di messaggi**

Un processo può inviare alcune informazioni a un altro processo nella forma di un messaggio. L'altro processo può copiare le informazioni nelle sue strutture dati e utilizzarle. Sia il processo mittente che quello destinatario devono anticipare lo scambio di informazioni, ovvero un processo deve sapere quando è previsto che invii o riceva un messaggio, per cui lo scambio di informazioni diventa una parte della convenzione o protocollo tra i processi.

### **Sincronizzazione**

Un programma può richiedere che un'azione  $a_i$  debba essere eseguita solo dopo l'esecuzione di un'altra azione  $a_j$ . La sincronizzazione tra processi è richiesta se queste azioni vengono eseguite in processi differenti - il processo che vuole eseguire l'azione  $a_i$  viene messo in attesa finché un altro processo non esegue l'azione  $a_j$ .

### **Segnali**

Un segnale viene utilizzato per segnalare una situazione di eccezione a un processo in modo che possa gestire la situazione mediante azioni appropriate. Il codice che un processo esegue alla ricezione di un segnale è chiamato *gestore del segnale* (signal handler). Il meccanismo dei segnali è costruito sul modello degli interrupt. In questo modo, quando un segnale è inviato a un processo, il kernel interrompe l'esecuzione del processo ed esegue un signal handler, se ne è stato specificato uno dal processo; altrimenti, lo può gestire con un'azione di default. I sistemi operativi differiscono nel modo in cui ripristinano un processo dopo l'esecuzione di un signal handler.

L'Esempio 5.5 illustra la condivisione, la comunicazione e la sincronizzazione tra processi nell'applicazione real-time dell'Esempio 5.1. L'implementazione dei segnali è descritta nel Paragrafo 5.2.6.

### **Esempio 5.5 Interazione tra processi in un'applicazione real-time di registrazione di dati**

Nell'applicazione real-time di registrazione di dati dell'Esempio 5.1, buffer è condiviso dai processi *copia\_campione* e *registra\_campione*. Se si utilizza una variabile *num\_di\_campioni\_in\_buffer* per indicare quanti campioni sono attualmente presenti nel buffer, entrambi questi processi avrebbero bisogno di aggiornare *num\_di\_campioni\_in\_buffer*, per cui la sua consistenza dovrebbe essere salvaguardata ritardando un processo che volesse aggiornarla se un altro processo sta accedendo a essa. Inoltre questi processi devono sincronizzare le loro attività in modo tale che un nuovo campione viene inserito in una locazione di *buffer* solo dopo che il campione precedentemente contenuto nella locazione è scritto nel file, e il contenuto di una locazione è scritto nel file solo dopo che un nuovo campione vi è stato inserito.

Questi processi, inoltre, hanno bisogno di conoscere la dimensione del buffer, ovvero quanti campioni può contenere. Allo stesso modo di *num\_di\_campioni\_in\_buffer*, un variabile *size* potrebbe essere utilizzata come dato condiviso. Tuttavia, l'uso come dato condiviso genera un overhead causato dalla mutua esclusione, che non è giustificato poiché la dimensione del buffer non è aggiornata regolarmente; essa cambia solo in situazioni di eccezione. Dunque questi processi potrebbero essere progettati per usare la dimensione del buffer come una variabile *locale buf\_size*. Il suo valore sarebbe inviato ai due processi dal processo *registratore\_dati* mediante l'uso di messaggi. Il processo *registratore\_dati* dovrebbe inviare i segnali a questi processi se la dimensione del buffer dovesse cambiare.

## 5.2.6 Segnali

Un segnale è utilizzato per notificare una situazione di eccezione a un processo e consentirgli di gestirlo immediatamente. Un SO definisce una lista di situazioni di eccezione e i nomi o numeri dei segnali a esse associati. Alcuni esempi sono le condizioni di overflow della CPU e le condizioni relative ai processi figli, all'utilizzo delle risorse o a comunicazioni urgenti da un utente a un processo. Il kernel invia un segnale a un processo quando si verifica la corrispondente situazione di eccezione. Alcuni tipi di segnali possono anche essere inviati dai processi utente. Un segnale inviato a un processo a causa di una condizione nel suo funzionamento, come una condizione di overflow nella CPU, è definito segnale *sincrono*, mentre quello inviato a causa di qualche altra condizione è definito segnale *asincrono*.

Per utilizzare i segnali, un processo effettua una chiamata di sistema *register\_handler* con la quale specifica una routine da eseguire alla ricezione di uno specifico segnale; questa routine è chiamata *signal handler*. Se un processo non specifica un signal handler per un segnale, il kernel esegue un *default handler* che effettua alcune azioni standard come il dump dello spazio di indirizzamento e la terminazione del processo.

Un processo  $P_i$  che desidera inviare un segnale a un altro processo  $P_j$  richiama la funzione di libreria *signal* con due parametri: l'id del processo destinatario, ovvero  $P_j$  e il numero del segnale. Questa funzione utilizza l'istruzione per generare un interrupt software *<SI\_instr> <codice\_interrupt>* per richiamare una system call chiamata *signal*. La routine di gestione dell'evento per la chiamata *signal* estrae i parametri per recuperare il numero del segnale. A questo punto, la routine di gestione dell'evento effettua il passaggio del segnale al processo  $P_j$  e termina, non effettuando nessun cambiamento nello stato del processo mittente, cioè di  $P_i$ .

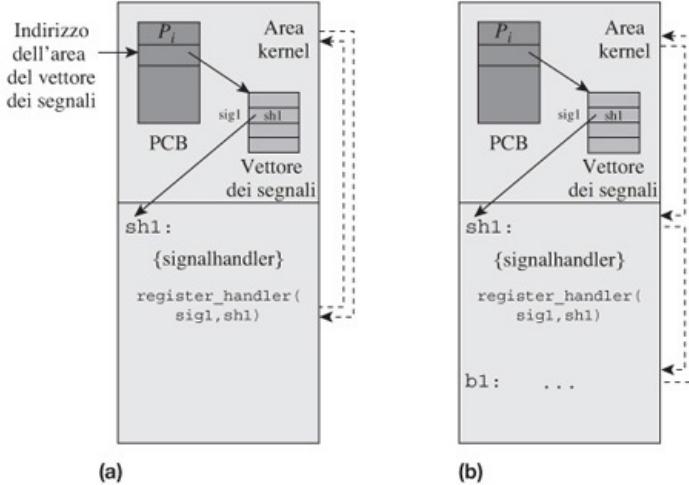
In un SO la gestione del segnale nel processo è implementata con le stesse modalità della gestione degli interrupt. Nel Paragrafo 2.2 abbiamo descritto come l'hardware degli interrupt utilizza un vettore di interrupt per ogni classe di interrupt, il quale contiene l'indirizzo di una routine che gestisce gli interrupt per una determinata classe. Una simile organizzazione può essere usata in ogni processo. L'area per la memorizzazione dei vettori dei segnali conterrebbe un vettore di segnali per ogni tipo di segnale, che a sua volta conterrebbe l'indirizzo di un signal handler. Quando un segnale è inviato a un processo, il kernel accede all'area di memorizzazione dei vettori dei segnali per verificare se è stato specificato un signal handler per quel segnale. In caso affermativo, passa il controllo all'handler; altrimenti, esegue il default handler per quel segnale.

La gestione dei segnali diventa complicata se il processo al quale si invia un segnale si trova nello stato *blocked*. Il kernel dovrebbe cambiare temporaneamente lo stato del processo a *ready* in modo che possa eseguire un signal handler, dopodiché dovrebbe riportare lo stato a *blocked*. Alcuni sistemi operativi preferiscono un approccio più semplice: annotare l'arrivo di un segnale se il processo destinatario si trova nello stato *blocked* e organizzare l'esecuzione del signal handler quando il processo diventa *ready* e viene schedulato.

L'Esempio 5.6 illustra come un processo gestisce un segnale.

### Esempio 5.6 Gestione del segnale

La [Figura 5.10](#) illustra l'organizzazione usata per la gestione dei segnali. Il codice del processo  $P_i$  contiene una funzione chiamata *sh1*, la cui ultima istruzione è un'istruzione di "ritorno dalla funzione", che estrae un indirizzo dallo stack e passa il controllo all'istruzione che si trova a questo indirizzo. Il processo  $P_i$  effettua una chiamata di libreria *register\_handler(sig1, sh1)* per registrare *sh1* come signal handler per il segnale *sig1*. La routine di libreria *register\_handler* effettua la system call *register\_handler*. Durante la gestione di questa chiamata, il kernel accede al PCB di  $P_i$ , ottiene l'indirizzo base dell'area di memorizzazione dei vettori dei segnali e inserisce l'indirizzo *sh1* nel vettore dei segnali relativo al segnale *sig1*. Il controllo, a questo punto, ritorna a  $P_i$ . Le frecce nella [Figura 5.10\(a\)](#) indicano gli indirizzi nelle strutture dati del kernel, mentre le frecce tratteggiate indicano come la CPU è commutata al kernel quando viene effettuata la chiamata di sistema e come viene commutata nuovamente al processo  $P_i$ .



**Figura 5.10** Gestione del segnale effettuata dal processo  $P_i$ : (a) registrazione di un signal handler; (b) invocazione del signal handler.

Supponiamo che il processo  $P_i$  sia prelazionato prima di eseguire l'istruzione con indirizzo  $b1$ . Alcuni istanti dopo, un processo  $P_i$  effettua la chiamata di sistema `signal` ( $P_i$ ,  $sig1$ ). Il kernel ritrova il PCB di  $P_i$ , ottiene l'indirizzo dell'area di memorizzazione dei vettori dei segnali e recupera il vettore dei segnali per  $sig1$ . A questo punto organizza l'esecuzione del signal handler per il processo  $P_i$  a partire dall'indirizzo  $sh1$  prima di riprendere la normale esecuzione come segue: ottiene l'indirizzo contenuto nel campo *program counter* (PC) dello stato salvato di  $P_i$  ovvero l'indirizzo  $b1$  che contiene la prossima istruzione che  $P_i$  deve eseguire. A questo punto il kernel inserisce questo indirizzo nello stack di  $P_i$  e copia l'indirizzo  $sh1$  nel campo *program counter* dello stato salvato di  $P_i$ . In questo modo, quando il processo  $P_i$  viene schedulato, esegue la funzione del signal handler con l'indirizzo base  $sh1$ . L'ultima istruzione di  $sh1$  estrae l'indirizzo  $b1$  dallo stack e passa il controllo all'istruzione con indirizzo  $b1$ , che ripristina il normale funzionamento del processo  $P_i$ . In effetti, come mostrato dalla freccia tratteggiata in [Figura 5.10\(b\)](#), l'esecuzione di  $P_i$  è dirottata al signal handler con indirizzo base  $sh1$  ed è ripristinata dopo l'esecuzione dello stesso.

### 5.3 Thread

Le applicazioni usano processi concorrenti per velocizzare la loro esecuzione. Tuttavia, la commutazione tra i processi all'interno di un'applicazione genera un elevato overhead dovuto alla quantità di informazione da salvare e ripristinare a ogni commutazione (Paragrafo 5.2.3). Per questo motivo i progettisti dei sistemi operativi hanno sviluppato un modello alternativo di esecuzione, chiamato *thread*, che favorisce la concorrenza all'interno di un'applicazione con un ridotto overhead.

Per comprendere la nozione di thread, analizziamo l'overhead dovuto alla commutazione tra processi per determinare un possibile risparmio di tempo. L'overhead dovuto alla commutazione ha due componenti.

- *Overhead relativo all'esecuzione*: lo stato della CPU del processo in esecuzione deve essere salvato e lo stato della CPU del nuovo processo deve essere caricato nei registri della CPU. Questo overhead è inevitabile.
- *Overhead dovuto all'uso delle risorse*: anche il contesto del processo deve essere commutato. Questo coinvolge la commutazione delle informazioni relative alle risorse allocate al processo, come memoria e file, e all'interazione del processo con altri processi. La maggior parte di queste informazioni aggiunge overhead alla commutazione di processo.

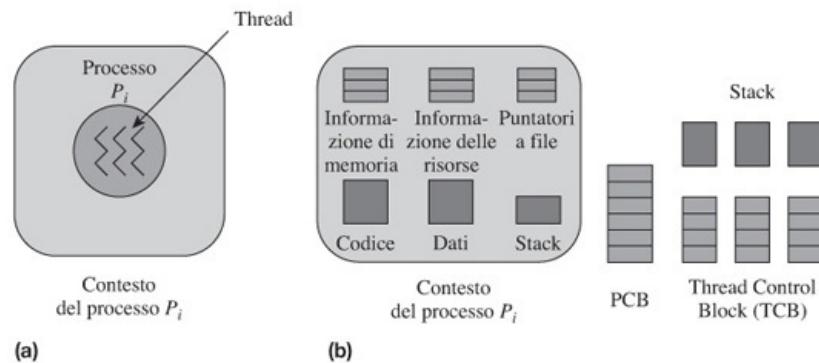
Si considerino i processi figli  $P_i$  e  $P_j$  del processo primario di un'applicazione. Questi processi ereditano il contesto del loro processo genitore. Se nessuno dei due ha allocata

alcuna risorsa, il loro contesto è identico; le loro informazioni di stato differiscono solo nello stato della CPU e nel contenuto dei rispettivi stack. Di conseguenza, durante la commutazione tra  $P_i$  e  $P_j$ , molte delle operazioni di salvataggio e caricamento sono ridondanti. I thread sfruttano questa caratteristica per ridurre l'overhead dovuto alla commutazione.

**Definizione 5.3 Thread** Esecuzione di un programma che usa le risorse di un processo.

Un processo crea un thread mediante una system call. Il thread non possiede risorse proprie, per cui non ha un contesto; viene eseguito utilizzando il contesto del processo e in tal modo accede alle risorse del processo. Utilizziamo la frase “thread di un processo” e “processo genitore di un thread” per descrivere la relazione tra un thread e il processo di cui utilizza il contesto. Bisogna notare che i thread non sostituiscono i processi figli; un'applicazione crea processi figli per eseguire differenti parti del suo codice e ogni processo figlio potrebbe creare dei thread per ottenere la concorrenza.

La [Figura 5.11](#) illustra la relazione tra thread e processi. Nella visione astratta di [Figura 5.11\(a\)](#), il processo  $P_i$  ha tre thread, che sono rappresentati come linee ondulate all'interno del cerchio che rappresentano il processo  $P_i$ . La [Figura 5.11\(b\)](#) mostra un'implementazione di quanto detto. Il processo  $P_i$  ha un contesto e un PCB. Ogni thread di  $P_i$  è un'istanza del programma, per cui ha un proprio stack e un *Thread Control Block* (TCB), che è analogo al PCB e memorizza le seguenti informazioni:



**Figura 5.11** Thread nel processo  $P_i$ : (a) concetto; (b) implementazione.

1. informazioni per la schedulazione del thread: thread id, priorità e stato;
2. stato della CPU, ovvero il contenuto della PSW e dei GPR;
3. puntatore al PCB del processo genitore;
4. puntatore al TCB, usato per creare le liste di TCB per la schedulazione.

L'uso dei thread di fatto divide lo stato del processo in due parti: lo stato delle risorse è associato al processo mentre uno stato dell'esecuzione, cioè lo stato della CPU, è associato a ogni thread. Il costo della concorrenza all'interno del contesto di un processo è ora semplicemente una replica dello stato di esecuzione per ogni thread. Gli stati di esecuzione devono essere scambiati durante la commutazione tra thread. Lo stato delle risorse non è né replicato né modificato durante la commutazione dei thread del processo.

#### ***Stati del thread e transizioni di stato***

Fatta eccezione per il fatto che i thread non hanno allocate risorse proprie, i thread e i processi sono analoghi. Dunque gli stati dei thread e le transizioni di stato dei thread sono analoghi agli stati dei processi e alle transizioni di stato dei processi. Lo stato di un processo appena creato è impostato a *ready* poiché il processo genitore già ha allocato le risorse necessarie. Entra nello stato *running* quando è sottoposto a dispatching. Non entra nello stato *blocked* a causa di una richiesta di risorsa, poiché non effettua nessuna

richiesta di risorsa; tuttavia, può entrare nello stato *blocked* a causa dei requisiti di sincronizzazione del processo. Per esempio, se i thread fossero utilizzati nell'applicazione real-time per il logging dei dati dell'Esempio 5.1, il thread *registro\_campione* entrerebbe nello stato *blocked* se non ci fossero dati campione in *buffer*.

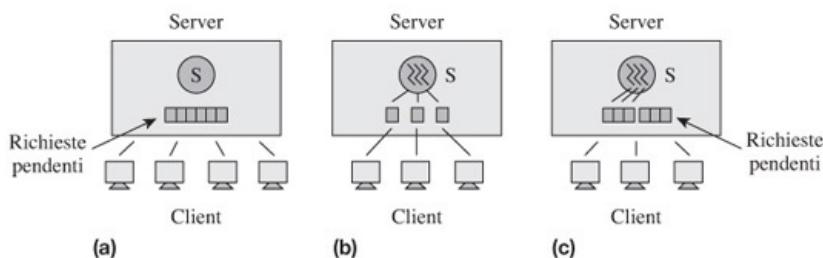
### Vantaggi dei thread rispetto ai processi

La [Tabella 5.8](#) riassume i vantaggi dei thread rispetto ai processi, tra i quali abbiamo già visto il vantaggio di un overhead inferiore relativamente alla creazione e alla commutazione. Diversamente dai processi figli, i thread condividono lo spazio di indirizzamento del processo genitore, per cui possono comunicare attraverso l'uso di dati condivisi anziché mediante messaggi, con il risultato di eliminare l'overhead delle chiamate di sistema.

| Vantaggio   | Motivazione   |
|---|---|
| Minore overhead dovuto alla creazione e alla commutazione | Lo stato del thread consiste solo dello stato dell'elaborazione. Lo stato dell'allocazione delle risorse e lo stato delle comunicazioni non sono parte dello stato del thread, per cui la creazione e la commutazione dei thread produce un overhead inferiore. |
| Comunicazione più efficiente                              | I thread di un processo possono comunicare tra loro attraverso dati condivisi, evitando in questo modo l'overhead di comunicazione dovuto alle chiamate di sistema.   |
| Progettazione semplificata                                | L'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente.  |

**Tabella 5.8** Vantaggi dei thread rispetto ai processi.

Le applicazioni che servono le richieste ricevute dagli utenti, come i sistemi di prenotazione dei voli o i sistemi bancari, sono chiamati *server*, i loro utenti sono chiamati *client* (l'elaborazione client-server è discussa nel [Paragrafo 16.5.1](#)). Le prestazioni dei server possono essere migliorate utilizzando la concorrenza e il parallelismo ([Paragrafo 5.1.4](#)), ovvero sia alternando le richieste che coinvolgono le operazioni di I/O sia utilizzando molte CPU per servire differenti richieste. L'uso dei thread semplifica la loro progettazione; ne discutiamo con l'aiuto della [Figura 5.12](#).



**Figura 5.12** Uso dei thread nella struttura di un server: (a) server che utilizza codice sequenziale; (b) server multithread; (c) server che utilizza un gruppo di thread.

La [Figura 5.12\(a\)](#) rappresenta un sistema di prenotazione di una linea area. Il server inserisce le richieste effettuate dai client in una coda e le evade una dopo l'altra. Se più richieste devono essere servite concorrentemente, il server dovrebbe adottare tecniche avanzate di I/O come l'I/O asincrono, e una logica complessa per commutare tra le elaborazioni delle richieste. Di contro, un server multithread potrebbe creare un nuovo thread per servire ogni nuova richiesta. Questo server non dovrebbe adottare nessuna tecnica speciale per la concorrenza poiché essa è implicita nella creazione dei thread. La [Figura 5.12\(b\)](#) mostra un server multithread, che ha creato tre thread poiché ha ricevuto tre richieste.

La creazione e la terminazione dei thread è più efficiente rispetto alla creazione e alla terminazione dei processi; tuttavia, il suo overhead può causare un decadimento delle prestazioni del server nel caso in cui i client effettuassero richieste a un tasso molto

elevato. Un'organizzazione denominata *thread pool*, impiegata per evitare questo overhead, consiste nel riutilizzare i thread invece di distruggerli dopo aver soddisfatto le richieste. Il gruppo di thread si compone di un thread principale che effettua la gestione e alcuni thread, utilizzati ripetutamente, che effettuano il lavoro. Il thread principale mantiene una lista di richieste pendenti e una lista di thread lavoratori idle. Quando viene sottoposta una nuova richiesta, il thread principale la assegna a un thread lavoratore libero, se ne esiste uno; altrimenti, inserisce la richiesta nella lista delle richieste pendenti. Quando un thread lavoratore completa il servizio di una richiesta, il thread principale gli assegna una nuova richiesta o lo inserisce nella lista dei thread idle. La [Figura 5.12\(c\)](#) illustra un server che utilizza un pool di thread, che contiene tre thread impegnati nel servizio di tre richieste, mentre altre tre richieste sono in attesa. Se fosse implementata la tecnica del pool di thread, il SO costituirebbe il thread principale del pool, cosa che semplificherebbe la codifica del server poiché non dovrebbe gestire esplicitamente la concorrenza. Il SO potrebbe anche modificare dinamicamente il numero di thread lavoratori per fornire un'adeguata concorrenza nell'applicazione e ridurre l'impiego delle risorse dovuto ai thread lavoratori idle.

### **Codifica per l'utilizzo dei thread**

I thread assicurano la correttezza dei dati condivisi e la sincronizzazione (Paragrafo 5.2.5). Il Paragrafo 5.3.1 descrive le caratteristiche dello standard POSIX, relativo ai thread, che può essere utilizzato per questo scopo. La correttezza dei dati condivisi ha anche un altro aspetto. Le funzioni o le subroutine che usano dati statici o globali per conservare dei valori durante le attivazioni successive, possono produrre risultati non corretti quando richiamate in maniera concorrente, poiché le chiamate di fatto condividono questi dati senza l'utilizzo della mutua esclusione. Queste routine sono dette *thread unsafe*. Un'applicazione che utilizza i thread deve essere codificata in modo *thread safe* e deve richiamare le routine solo da una libreria thread safe.

La gestione dei segnali richiede un'attenzione particolare in un'applicazione multithread. Bisogna ricordare che il kernel consente a un processo di specificare un signal handler (Paragrafo 5.2.6). Quando più thread sono creati all'interno di un processo, quale thread dovrebbe gestire un segnale? Ci sono diverse possibilità. Il kernel può selezionare uno dei thread per la gestione del segnale. Questa scelta può essere sia statica, per esempio il primo o l'ultimo thread creato nel processo, sia dinamica, per esempio il thread a più alta priorità. In alternativa, il kernel può consentire a un'applicazione di specificare quale thread debba gestire i segnali in ogni istante.

Un segnale sincrono viene generato come risultato dell'attività di un thread, per cui la scelta migliore è lasciare che lo gestisca il thread stesso. Idealmente, ogni thread dovrebbe essere in grado di specificare quali segnali sincroni è interessato a gestire. Tuttavia, per implementare questa funzione, il kernel dovrebbe replicare, per ogni thread, l'organizzazione per la gestione dei segnali di [Figura 5.6](#), per cui pochi sistemi operativi la mettono a disposizione. Un segnale asincrono può essere gestito da un thread nel processo. Per garantire una risposta immediata alla condizione che ha causato il segnale, il thread a più alta priorità dovrebbe gestire questi segnali.

### **5.3.1 Thread POSIX**

La Portable Operating System Interface (POSIX) dello standard ANSI/IEEE, definisce l'API pthread utilizzabile nei programmi scritti in linguaggio C. Diffusamente chiamata thread POSIX, questa interfaccia fornisce 60 routine che effettuano le seguenti operazioni.

- *Gestione dei thread*: i thread sono gestiti mediante chiamate alle routine della libreria dei thread per la creazione, il recupero dello stato, la terminazione normale o anormale, l'attesa della terminazione, l'impostazione degli attributi di schedulazione e il dimensionamento dello stack.
- *Supporto per la condivisione dei dati*: i dati condivisi dai thread possono contenere valori non corretti se due o più thread li aggiornano concorrentemente. Una funzionalità chiamata mutex è fornita per assicurare la mutua esclusione tra thread nell'accesso ai dati condivisi, ovvero per assicurare che solo un thread abbia accesso ai dati condivisi a ogni istante di tempo. Sono inoltre fornite routine per accedere all'uso dei dati condivisi e per segnalare la fine dell'uso degli stessi. Se nell'Esempio 5.5 si adottasse il modello a thread, i thread *copia\_campione* e *registra\_campione* utilizzerebbero un mutex per assicurare la mutua esclusione nell'accesso e

nell'aggiornamento della variabile *num\_campioni\_in\_buffer*.

- *Supporto per la sincronizzazione*: le variabili di condizione sono messe a disposizione per semplificare il coordinamento tra thread in modo che le loro azioni possono essere eseguite nell'ordine desiderato. Se nell'Esempio 5.5 si adottasse il modello a thread, le variabili di condizione verrebbero impiegate per assicurare che il thread *copia\_campione* copi un campione in *buffer* prima che *registra\_campione* lo scriva nel file.

La [Figura 5.13](#) illustra l'uso dei pthread nell'applicazione real-time di raccolta dei dati dell'Esempio 5.1. Un pthread è creato mediante la chiamata

```
#include <pthread.h>
#include <stdio.h>
int size, buffer[100], no_of_samples_in_buffer;
int main()
{
    pthread_t id1, id2, id3;
    pthread_mutex_t buf_mutex, condition_mutex;
    pthread_cond_t buf_full, buf_empty;
    pthread_create(&id1, NULL, move_to_buffer, NULL);
    pthread_create(&id2, NULL, write_into_file, NULL);
    pthread_create(&id3, NULL, analysis, NULL);
    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
    pthread_join(id3, NULL);
    pthread_exit(0);
}

void *move_to_buffer()
{
    /* Ripeti finché non sono stati ricevuti tutti i campioni */
    /* Se non c'è spazio nel buffer, esegui una wait su buf_full */
    /* Usa buf_mutex per accedere al buffer, incrementa il numero di
       elementi del buffer */
    /* Esegui una signal su buf_empty */
    pthread_exit(0);
}

void *write_into_file()
{
    /* Ripeti finché tutti i campioni non sono stati scritti nel file */
    /* Se non ci sono dati nel buffer, esegui una wait su buf_empty */
    /* Usa buf_mutex per accedere al buffer, decrementa il numero di
       elementi del buffer */
    /* Esegui una signal su buf_full */
    pthread_exit(0);
}

void *analysis()
{
    /* Ripeti finché tutti i campioni non sono stati analizzati */
    /* Leggi un campione dal buffer ed analizzarlo */
    pthread_exit(0);
}
```

[Figura 5.13](#) Profilo dell'applicazione di raccolta dei dati utilizzando i thread PSOIX.

```
pthread_create (<struttura dati>, <attributi>,
               <routine di avvio>, <argomenti>)
```

dove la struttura dati del thread è di fatto l'id del thread, mentre gli attributi indicano la priorità per la schedulazione e le opzioni di sincronizzazione. Un thread termina la sua esecuzione con la chiamata alla routine *pthread\_exit* che prende come parametro lo stato del thread. La sincronizzazione tra il thread genitore e un thread figlio è effettuata mediante la chiamata *pthread\_join*, che prende come parametri un id del thread e altri valori. Nell'effettuare questa chiamata, il thread genitore viene bloccato finché il thread indicato nella chiamata non ha terminato la sua esecuzione; viene generato un errore se lo stato di terminazione del thread non coincide con gli attributi indicati nella chiamata a

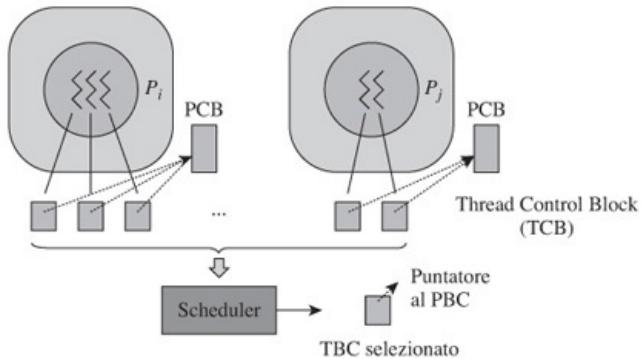
`pthread_join`. Alcune implementazioni richiedono che un thread sia creato con l'attributo "joinable" per indicare che deve essere sincronizzato. Il codice in [Figura 5.13](#) crea tre thread per eseguire le funzioni effettuate dai processi nell'Esempio 5.1. Come detto sopra e indicato nei commenti in [Figura 5.13](#), i thread utilizzerebbero il mutex `buf_mutex` per assicurare l'accesso esclusivo al buffer e utilizzerebbero le variabili di condizione `buf_full` e `buf_empty` per assicurare che depositino i campioni nel buffer e li prendano dal buffer nel corretto ordine. I dettagli dei mutex e delle variabili di condizione saranno discussi successivamente nel [Capitolo 6](#).

### 5.3.2 Thread di livello kernel, di livello utente e ibridi

Questi tre modelli di thread differiscono nel ruolo del processo e del kernel nella creazione e nella gestione dei thread. Questa differenza ha un impatto significativo sull'overhead dovuto alla commutazione del thread, sulla concorrenza e sul parallelismo interno al processo.

#### Thread di livello kernel

Un thread di livello kernel è implementato dal kernel. Dunque la creazione, la terminazione dei thread di livello kernel e il recupero dello stato sono effettuati mediante system call. La [Figura 5.14](#) mostra uno schema di come il kernel gestisce i thread di livello kernel. Quando un processo effettua una system call `create_thread`, il kernel crea un thread, gli assegna un id e alloca un thread control block (TCB). Il TCB contiene un puntatore al PCB del processo genitore del thread.



**Figura 5.14** Schedulazione dei thread di livello kernel.

Quando si verifica un evento, il kernel salva lo stato della CPU del thread interrotto nel suo TCB. Dopo la gestione dell'evento, lo scheduler esamina i TCB di tutti i thread e ne seleziona uno *ready*; il dispatcher utilizza il puntatore al PCB presente nel TCB per controllare se il thread selezionato appartiene a un processo diverso rispetto a quello del thread interrotto. In caso affermativo, salva il contesto del processo cui appartiene il thread interrotto e sottopone il thread a dispatch. Tuttavia, le azioni per salvare e caricare il contesto del processo non vengono effettuate se entrambi i thread appartengono allo stesso processo. Questa caratteristica riduce l'overhead dovuto alla commutazione, che nel caso di thread di livello kernel, risulta più veloce di un ordine di grandezza, ovvero 10 volte più veloce, rispetto alla commutazione tra processi.

#### Vantaggi e svantaggi dei thread di livello kernel

Un thread di livello kernel è come un processo eccetto che ha una quantità inferiore di informazioni di stato. Questa similarità è conveniente per i programmati, poiché la programmazione dei thread non è differente dalla programmazione dei processi. In un sistema multiprocessore, i thread di livello kernel forniscono parallelismo (vedi Paragrafo 5.1.4), dal momento che molti processi appartenenti a un processo possono essere schedulati simultaneamente, cosa che non è possibile con i thread di livello utente descritti nel prossimo paragrafo; pertanto forniscono un miglior speedup dell'elaborazione rispetto ai thread di livello utente.

Tuttavia, gestire i thread come i processi ha anche i suoi svantaggi. La commutazione

dei thread è effettuata dal kernel come risultato della gestione di un evento, per cui si genera overhead anche se il thread interrotto e quello selezionato appartengono allo stesso processo. Questa caratteristica limita il risparmio nell'overhead dovuto alla commutazione dei thread.

### Thread di Livello Utente

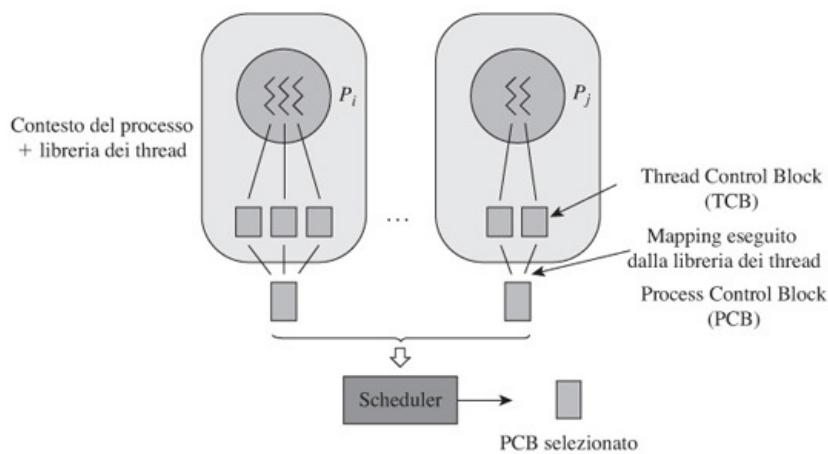
I thread di livello utente sono implementati da una *libreria di thread*, che viene linkata al codice del processo. La libreria imposta l'organizzazione dell'implementazione dei thread mostrata in [Figura 5.11\(b\)](#) senza coinvolgere il kernel e gestisce l'alternanza dell'esecuzione dei thread nel processo. In questo modo, il kernel non è a conoscenza della presenza dei thread di livello utente in un processo; il kernel vede solo il processo. Molti SO implementano l'API `pthread` introdotta nello standard POSIX dell'IEEE (Paragrafo 5.3.1) in questo modo.

Vediamo una descrizione della creazione e del funzionamento dei thread. Un processo invoca la funzione di libreria `create_thread` per creare un nuovo thread, pertanto la funzione di libreria crea un TCB per il nuovo thread e inizia a considerare il nuovo thread per la "schedulazione". Quando il thread nello stato *running* invoca una funzione di libreria per effettuare la sincronizzazione, per esempio per attendere finché non si verifica un evento, la funzione di libreria richiama lo "scheduler" e seleziona un altro thread del processo per l'esecuzione. In questo modo, il kernel non ha bisogno di commutare tra i thread ritenendo che il *processo* sia in continua esecuzione. Se la libreria di thread non trova nel processo un thread pronto per l'esecuzione, effettua un chiamata di sistema per sospendere il processo.

Il kernel a questo punto blocca il processo, che sarà sbloccato quando qualche evento attiva uno dei suoi thread e di conseguenza ripristina la funzione di libreria, che a sua volta effettuerà la schedulazione e la commutazione del thread appena attivato.

### Schedulazione dei thread di livello utente

La [Figura 5.15](#) rappresenta uno schema della schedulazione dei thread di livello utente. Il codice della libreria di thread è parte di ogni processo. Effettua la "schedulazione" per selezionare un thread e organizza la sua esecuzione. Possiamo vedere questa operazione come una "mappatura" del TCB del thread selezionato sul PCB del processo.



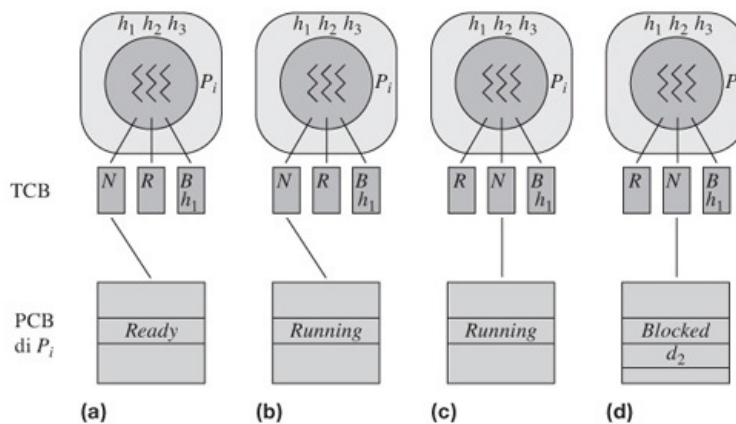
**Figura 5.15** Scheduling dei thread di livello utente.

La libreria di thread utilizza le informazioni nel TCB per decidere quale thread debba essere eseguito ogni volta. Per effettuare il "dispatch" del thread, lo stato della CPU del thread dovrebbe diventare lo stato della CPU del processo e lo stack pointer del processo dovrebbe puntare allo stack del processo. Poiché la libreria di thread è parte del processo, la CPU si trova in modalità utente. Dunque un thread non può essere sottoposto a dispatch caricando nuove informazioni nel PSW; la libreria di thread deve usare istruzioni non privilegiate per modificare il contenuto del PSW. Di conseguenza, carica l'indirizzo dello stack del thread nel registro contenente l'indirizzo dello stack, ottiene l'indirizzo contenuto nel campo *program counter* (PC) dello stato della CPU del

thread presente nel TCB ed esegue un’istruzione di salto per trasferire il controllo all’istruzione che si trova a questo indirizzo. L’esempio seguente illustra delle situazioni interessanti che si verificano durante la schedulazione dei thread di livello utente.

### Esempio 5.7 Scheduling dei thread di livello utente

La Figura 5.16 illustra come la libreria di thread gestisce tre thread in un processo  $P_i$ . I codici  $N$ ,  $R$  e  $B$  nei TCB rappresentano, rispettivamente, gli stati *running*, *ready* e *blocked*. Il processo  $P_i$  è nello stato *running* e la libreria di thread è in esecuzione. Il thread  $h_1$  è sottoposto a dispatching, per cui il suo stato è mostrato come  $N$ , ovvero *running*. Successivamente il processo  $P_i$  viene prelazionato dal kernel. La Figura 5.16(a) illustra gli stati dei thread e del processo  $P_i$ . Il thread  $h_i$  è nello stato *running* e il processo  $P_i$  è nello stato *ready*. Il thread  $h_1$  riprende la sua esecuzione quando il processo  $P_i$  viene schedulato di nuovo. La linea dal TCB di  $h_1$  al PCB di  $P_i$  indica che il TCB di  $h_1$  attualmente è mappato sul PCB di  $P_i$ . Questo è importante per il dispatching e il salvataggio del contesto effettuati dalla libreria di thread.



**Figura 5.16** Azioni della libreria di thread ( $N$ ,  $R$ ,  $B$  indicano *running*, *ready* e *blocked*).

Il thread  $h_2$  si trova nello stato *ready* in Figura 5.16(a), per cui il suo TCB contiene il codice  $R$ . Il thread  $h_3$  attende un’azione di sincronizzazione da  $h_1$ , per cui si trova nello stato *blocked*. Il suo TCB contiene il codice  $B$  e  $h_1$  per indicare che è in attesa di un evento causato da un’azione di sincronizzazione effettuata da  $h_1$ . La Figura 5.16(b) mostra la situazione quando il kernel effettua il dispatch di  $P_i$  e modifica il suo stato a *running*.

La libreria di thread sovrappone l’esecuzione dei thread utilizzando il timer. Durante la “schedulazione” di  $h_1$ , la libreria avrebbe richiesto un interrupt dopo un piccolo intervallo di tempo. Quando si verifica l’interrupt timer, prende il controllo mediante la routine di gestione dell’evento del kernel per gli interrupt timer e decide di prelazionare  $h_1$ . Quindi salva lo stato della CPU nel TCB di  $h_1$  e “schedula”  $h_2$ . Di conseguenza i codici di stato nei TCB di  $h_1$  e  $h_2$  cambiano, rispettivamente, a  $R$  e  $N$  (Figura 5.16(c)). È bene ricordare che lo scheduling dei thread effettuato dalla libreria non è visibile al kernel. Durante tutti questi eventi, il kernel vede il processo  $P_i$  nello stato *running*.

Un thread utente non dovrebbe effettuare system call bloccanti; tuttavia, vediamo cosa succederebbe se  $h_2$  effettuasse una system call per avviare un’operazione di I/O sul dispositivo  $d_2$ , che risulta in una chiamata di sistema bloccante. Il kernel imposterebbe lo stato del processo  $P_i$  a *blocked* e annoterebbe che è bloccato a causa di un’operazione di I/O sul dispositivo  $d_2$  (Figura 5.16(d)). Ad un certo punto dopo il completamento dell’operazione di I/O, il kernel schedulerebbe il processo  $P_i$  e l’esecuzione di  $h_2$  verrebbe ripristinata. Bisogna notare che lo stato nel TCB di  $h_2$  rimane a  $N$ , ovvero *running*, durante tutta la sua esecuzione!

### **Vantaggi e svantaggi dei thread di livello utente**

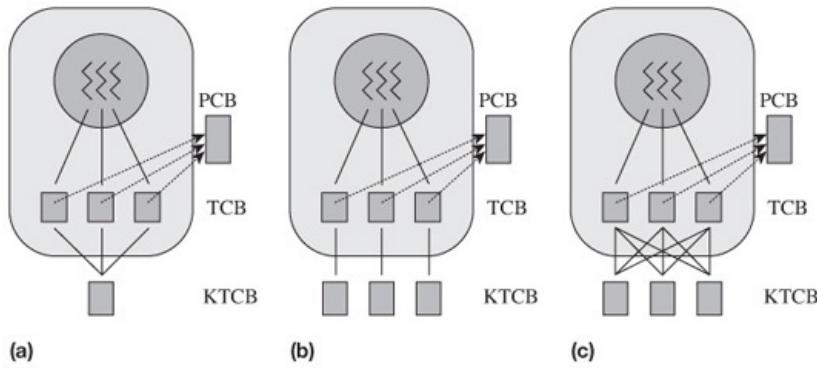
La sincronizzazione e la schedulazione dei thread sono implementate dalla libreria. Questa organizzazione evita l'overhead della system call per la sincronizzazione dei thread, per cui l'overhead dovuto alla commutazione dei thread potrebbe essere di un ordine di grandezza inferiore rispetto ai thread di livello kernel. Inoltre questa organizzazione consente a ogni processo di usare una politica di scheduling che meglio si adatta alla sua natura. Un processo che implementa un'applicazione real-time può utilizzare uno scheduling a priorità per soddisfare i requisiti di risposta, mentre un processo che implementa un server multithread può usare uno scheduling roundrobin. Tuttavia, le prestazioni di un'applicazione dipenderebbero dal fatto che la schedulazione dei thread di livello utente a opera della libreria sia compatibile con la schedulazione dei processi effettuata dal kernel. Per esempio, lo scheduling roundrobin nella libreria di thread sarebbe compatibile sia con lo scheduling round-robin che con quello a priorità nel kernel, mentre lo scheduling a priorità sarebbe compatibile solo con quello a priorità nel kernel.

La gestione dei thread senza il coinvolgimento del kernel ha anche alcuni svantaggi. Primo fra tutti, il kernel non conosce la differenza tra un thread e un processo, per cui se un thread si bloccasse su una system call, il kernel bloccherebbe il processo genitore. In effetti, *tutti* i thread del processo sarebbero bloccati finché non sarebbe rimossa la causa del blocco. Nella [Figura 5.16\(d\)](#) dell'Esempio 5.7, il thread  $h_1$  non può essere schedulato sebbene si trovi nello stato *ready* poiché il thread  $h_2$  ha effettuato una chiamata di sistema bloccante. Dunque i thread non devono effettuare chiamate di sistema che possono portare a un blocco. Per facilitare questa situazione, un SO dovrebbe rendere disponibile un versione non bloccante di ogni chiamata di sistema che altrimenti porterebbe al blocco di un processo. Inoltre, dal momento che il kernel schedula un processo e la libreria schedula un thread all'interno del processo, al più un thread di un processo può essere in esecuzione in ogni istante di tempo. Di conseguenza, i thread di livello utente non possono implementare il parallelismo (Paragrafo 5.1.4) e la concorrenza da essi fornita viene seriamente compromessa se un thread effettua una chiamata di sistema che porta a un blocco.

### **Modello dei thread ibrido**

Un modello di thread ibrido implementa sia i thread di livello utente sia i thread di livello kernel e anche un metodo per associare i thread di livello utente ai thread di livello kernel. Metodi differenti di implementazione di questa associazione producono differenti combinazioni caratterizzate da ridotto overhead di commutazione dei thread di livello utente ed elevata concorrenza e parallelismo dei thread di livello kernel.

La [Figura 5.17](#) illustra tre metodi per associare i thread di livello utente ai thread di livello kernel. La libreria di thread crea i thread di livello utente in un processo e associa un *thread control block* (TCB) a ogni thread utente. Il kernel crea i thread di livello kernel in un processo e associa un *kernel thread control block* (KTCB) a ogni thread kernel. Nel metodo di associazione molti a uno, il kernel crea un singolo thread kernel nel processo e tutti i thread utente creati nel processo dalla libreria sono associati con l'unico thread kernel. Questo metodo crea un effetto simile ai semplici thread di livello utente; i thread utente possono essere concorrenti senza essere paralleli, la commutazione dei thread genera un overhead ridotto e il blocco di un thread porta al blocco di tutti i thread nel processo.



**Figura 5.17** Associazioni dei thread ibridi: (a) molti a uno; (b) uno a uno; (c) molti a molti.

Nel modello di associazione uno a uno, ogni thread di livello utente è mappato permanentemente su un thread di livello kernel. Questa associazione crea un effetto simile ai semplici thread di livello kernel: i thread possono essere eseguiti in parallelo su diverse CPU di un sistema multiprocessore; tuttavia, la commutazione dei thread è effettuata a livello kernel e provoca un elevato overhead. Il blocco di un thread utente non blocca gli altri thread del processo poiché sono mappati su differenti thread kernel.

Il metodo di associazione molti a molti permette a un thread utente di essere mappato su differenti thread kernel in momenti diversi (Figura 5.17(c)). Questo modello consente di sfruttare il parallelismo tra i thread di livello utente che sono mappati, allo stesso tempo, su differenti thread di livello kernel e provoca un basso overhead di commutazione dei thread utente che sono schedulati sullo stesso thread kernel dalla libreria di thread. Tuttavia, il metodo di associazione molti a molti richiede un'implementazione complessa. I dettagli di questo modello verranno discussi nel Paragrafo 5.4.3 dove si parlerà del modello di thread ibrido utilizzato nel sistema operativo Sun Solaris a partire da Solaris 8.

## 5.4 Casi di studio: processi e thread

### 5.4.1 Processi in Unix

#### Strutture dati

Unix utilizza due strutture dati per memorizzare i dati di controllo relativi ai processi:

- *struttura proc*: contiene l'id del processo, lo stato del processo, la priorità, le informazioni relative alle relazioni con altri processi, un descrittore dell'evento per il quale un processo bloccato è in attesa, la maschera per la gestione dei segnali e le informazioni relative alla gestione della memoria.
- *u-area* ("user area"): contiene un process control block, che conserva lo stato della CPU per un processo bloccato, il puntatore alla struttura *proc*, gli id dell'utente e del gruppo e le seguenti informazioni: gestori dei segnali, file aperti e la directory corrente, il terminale collegato al processo e l'utilizzo della CPU del processo.

Queste strutture dati insieme mantengono l'informazione analoga alla struttura dati PCB discussa nel Paragrafo 5.2. La struttura *proc* principalmente memorizza i dati relativi alla schedulazione mentre la *u-area* contiene i dati relativi all'allocazione delle risorse e alla gestione dei segnali. La struttura *proc* di un processo è sempre mantenuta in memoria. La *u-area* ha bisogno di essere in memoria solo quando il processo è in esecuzione.

#### Tipi di processi

In Unix esistono due tipi di processi: processi utente e processi kernel. Un *processo utente* esegue un'elaborazione per l'utente ed è associato al terminale di controllo dell'utente. Quando un utente avvia un programma, il kernel crea il processo primario, che a sua volta può creare processi figli (Paragrafo 5.1.2).

Un *processo daemon* è un processo che non dipende dal terminale di controllo

dell'utente. Viene eseguito in background e generalmente esegue funzioni di sistema, per esempio lo spooling di stampa e la gestione della rete. Una volta creati, i processi daemon possono rimanere in esecuzione durante tutto il funzionamento del SO. I *processi kernel* eseguono il codice del kernel e riguardano le attività di background del kernel come lo swapping. Sono creati automaticamente all'avvio del sistema e possono richiamare le funzioni del kernel o far riferimento alle strutture dati del kernel senza dover effettuare una system call.

### **Creazione e terminazione di un processo**

La chiamata di sistema *fork* crea un processo figlio e imposta il suo contesto (chiamato, nella letteratura Unix, *contesto utente*). Alloca una struttura *proc* per il processo appena creato, imposta il suo stato a *ready* e infine alloca la *u-area* per il processo. Il kernel tiene traccia della relazione genitore-figlio utilizzando la struttura *proc*. La *fork* restituisce l'id del processo figlio.

Il contesto utente del processo figlio è una copia del contesto utente del genitore. Di conseguenza il figlio esegue lo stesso codice del genitore. Alla creazione, il program counter del processo figlio contiene l'indirizzo dell'istruzione successiva alla *fork*. La chiamata a *fork* restituisce uno 0 nel processo figlio, che rappresenta l'unica differenza tra il processo genitore e figlio. Un processo figlio può eseguire lo stesso programma del genitore oppure può usare una chiamata di sistema della famiglia delle system call *exec* per caricare un altro programma da eseguire. Sebbene questa organizzazione sia macchinosa, essa consente al processo figlio di scegliere se eseguire il codice del genitore nel contesto del genitore oppure un altro programma. La prima alternativa era usata nei sistemi Unix più vecchi per impostare dei server che potessero servire molte richieste in maniera concorrente.

La visione d'insieme della creazione e della terminazione di un processo in Unix è la seguente: dopo la fase di boot, il sistema crea un processo *init*. Questo processo crea un processo figlio per ogni terminale connesso al sistema. Dopo una sequenza di chiamate *exec*, ogni processo figlio avvia il programma di login. Quando un utente specifica il nome di un file dalla linea di comando, la shell crea un nuovo processo che esegue una chiamata *exec* per il file specificato, che di fatto diventa il processo principale del programma. In questo modo il processo principale è un figlio del processo che esegue la shell, il quale esegue una system call *wait* (descritta successivamente in questo paragrafo) per attendere la fine del processo principale del programma. Di conseguenza si blocca finché il programma non completa la sua esecuzione, e ritorna a essere attivo per accettare un nuovo comando. Se un processo di shell esegue una chiamata *exit* per concludere la propria esecuzione, *init* crea un nuovo processo per il terminale, che esegue il programma di login.

Un processo  $P_i$  può terminare se stesso mediante la system call *exit (status\_code)*, dove *status\_code* è il codice che indica lo stato di terminazione del processo. Alla ricezione della chiamata *exit* il kernel salva il codice di stato nella struttura *proc* di  $P_i$ , chiude tutti i file aperti, rilascia la memoria allocata al processo e distrugge la sua *u-area*. Tuttavia, la struttura *proc* è conservata finché il genitore di  $P_i$  non la distrugge. In questo modo il genitore di  $P_i$  può richiedere lo stato di terminazione in ogni momento. In pratica, il processo è terminato ma esiste, per cui si chiama processo *zombie*.

La chiamata *exit* inoltre invia un segnale al genitore di  $P_i$ . I processi figli di  $P_i$  diventano figli del processo *init* del kernel. In questo modo *init* riceve un segnale quando un figlio di  $P_i$ , per esempio  $P_c$ , termina e dunque può rilasciare la struttura *proc* di  $P_c$ .

### **Attesa della terminazione di un processo**

Un processo  $P_i$  può attendere la terminazione di un processo figlio utilizzando la chiamata di sistema *wait (addr(...))*, dove *(addr(...))* è l'indirizzo di una variabile, per esempio *xyz*, all'interno dello spazio di indirizzamento di  $P_i$ . Se il processo  $P_i$  possiede processi figli e almeno uno di loro ha terminato la sua esecuzione, la chiamata *wait* memorizza lo stato di terminazione del processo terminato in *xyz* e restituisce immediatamente l'id del processo figlio. Se più di un processo ha terminato la propria esecuzione, i rispettivi stati di terminazione saranno disponibili al processo  $P_i$  solo con ripetute chiamate alla *wait*. Lo stato del processo  $P_i$  viene impostato a *blocked* se possiede figli ma nessuno ha terminato l'esecuzione. Sarà sbloccato quando uno dei processi figli termina. La chiamata *wait* restituisce "-1" se  $P_i$  non ha figli. Il seguente

esempio mostra i benefici di questa semantica della chiamata *wait*.

### Esempio 5.8 Processi figli in Unix

La [Figura 5.18](#) mostra il codice C di un processo che crea tre processi figli nel ciclo for e attende la fine della loro esecuzione. Questo codice può essere usato per impostare i processi del sistema di registrazione dei dati real-time dell'Esempio 5.1. Bisogna notare che la chiamata *fork* restituisce al processo chiamante l'id del processo figlio appena creato mentre restituisce uno 0 al processo figlio. A causa di questa peculiarità, i processi figli vengono eseguiti nell'istruzione *if* mentre il processo genitore non esegue l'istruzione *if* ed esegue l'istruzione *wait*. L'attesa si conclude appena un processo figlio termina con l'istruzione *exit*. Tuttavia, il processo genitore vuole attendere finché non termina l'ultimo processo, per cui esegue un'altra *wait* se il valore restituito è diverso da -1. La quarta chiamata alla *wait* restituisce -1, il che provoca l'uscita del processo genitore dal ciclo. Il codice del processo genitore non contiene una chiamata esplicita a *exit()*. Il compilatore la aggiunge automaticamente alla fine del *main()*.

```
main()
{
    int saved_status;
    for (i=0; i<3; i++)
    {
        if (fork()==0)
        { /* codice del processo figlio */
            ...
            exit();
        }
    }
    while (wait(&saved_status) !=-1);
        /* cicla finché tutti i processi figli
           non sono terminati */
}
```

**Figura 5.18** Creazione e terminazione di un processo in Unix.

### Attesa dell'occorrenza degli eventi

Si dice che un processo bloccato su un evento è in *sleep* sull'evento; per esempio, un processo che avvia un'operazione di I/O sarebbe in *sleep* sull'evento relativo al completamento. Unix utilizza un'interessante organizzazione per attivare i processi in *sleep* su un evento. Unix non utilizza gli event control block (ECB) descritti precedentemente nel [Paragrafo 5.2.4](#) e utilizza invece gli *indirizzi dell'evento*. Un insieme di indirizzi è riservato nel kernel e ogni evento viene mappato in uno di questi indirizzi. Quando un processo deve attendere un evento, viene calcolato l'indirizzo dell'evento, lo stato del processo è modificato a *blocked* e l'indirizzo dell'evento è memorizzato nella struttura del processo. Questo indirizzo viene utilizzato come descrizione dell'evento atteso dal processo. Quando si verifica l'evento, il kernel calcola l'indirizzo dell'evento e attiva tutti i processi in attesa.

In alcune situazioni, questa organizzazione genera un overhead inutile. Per esempio, si considerino diversi processi in attesa dello stesso evento in conseguenza di una sincronizzazione per l'accesso ad alcuni dati. Quando si verifica l'evento, tutti questi processi sono attivati ma solo uno ottiene l'accesso ai dati mentre gli altri ritornano in attesa. Questa situazione è analoga al *busy wait*, che verrà discusso nel prossimo capitolo. A questo si aggiunge il problema di mappare gli eventi su indirizzi. Viene utilizzato uno schema di hashing per la mappatura, per cui due o più eventi possono essere mappati sullo stesso indirizzo dell'evento. In questo modo l'occorrenza di uno qualsiasi di questi eventi attiverà tutti i processi in attesa su *tutti* questi eventi. Ogni processo attivato dovrebbe controllare se l'evento che attendeva si è davvero verificato e ritornare in attesa nel caso non si sia verificato.

### Gestione dell'interrupt

Unix blocca gli interrupt durante l'esecuzione delle operazioni sensibili di livello kernel, assegnando a ogni interrupt un *livello di priorità dell'interrupt (ipl)*. In base al programma eseguito dalla CPU, un livello di priorità dell'interrupt viene anche associato

alla CPU. Quando viene generato un interrupt a un livello di priorità  $l$ , esso viene gestito solo se  $l$  è maggiore del livello di priorità dell'interrut della CPU; in caso contrario, viene sospeso finché il livello di priorità dell'interrupt della CPU diventa  $< l$ . Il kernel utilizza questa caratteristica per prevenire inconsistenze delle strutture dati del kernel aumentando l'ipl della CPU a un valore alto prima di iniziare ad aggiornare le sue strutture dati e diminuendolo una volta completato l'aggiornamento.

### **System call**

Quando viene effettuata una system call, il gestore delle system call utilizza il numero della chiamata per determinare quale funzione del sistema è stata invocata. Dalla sua tabella interna, recupera l'indirizzo del gestore per quella funzione. Inoltre conosce il numero di parametri necessari per effettuare la chiamata. Tuttavia, questi parametri sono presenti sullo stack utente, il quale è parte del contesto del processo che ha effettuato la chiamata. Per cui questi parametri vengono copiati dallo stack del processo all'interno nell'*u-area* del processo prima che il controllo sia passato al gestore della specifica chiamata di sistema. Questa azione semplifica il funzionamento dei singoli gestori degli eventi.

### **Segnali**

Un segnale può essere inviato a un processo o a un gruppo di processi. Questa azione è effettuata dalla system call *kill* ( $<pid>$ ,  $<signum>$ ), dove  $<pid>$  è un valore intero positivo, nullo o negativo. Un valore positivo indica l'id del processo al quale il segnale deve essere inviato. Uno 0 indica che il segnale deve essere inviato ad alcuni processi appartenenti allo stesso albero del mittente, ovvero ad alcuni processi che condividono un antenato con il processo mittente. Questa caratteristica è implementata come segue: al momento della chiamata *fork*, al processo appena creato viene assegnato un id del gruppo che è lo stesso assegnato al suo genitore. Un processo può modificare il suo id di gruppo utilizzando la system call *setgrp*. Quando  $<pid> = 0$ , il segnale è inviato a tutti i processi con id del gruppo uguale a quello del mittente. Un valore negativo di  $<pid>$  viene utilizzato per raggiungere i processi al di fuori dell'albero dei processi del mittente. In questa sede non approfondiremo questa caratteristica.

Un processo specifica un gestore del segnale eseguendo l'istruzione

```
oldfunction = signal (<signum>, <function>)
```

dove *signal* è una funzione della libreria C che effettua una chiamata alla system call *signal*,  $<signum>$  è un intero e  $<function>$  è una funzione nello spazio di indirizzamento del processo. Questa chiamata specifica che la funzione  $<function>$  dovrà essere eseguita al verificarsi del segnale  $<signum>$ . La chiamata *signal* ritorna l'azione precedentemente specificata per il segnale  $<signum>$ . Un utente può specificare *SIG\_DFL* come  $<function>$  per indicare che, all'occorrenza del segnale, deve essere eseguita l'azione di default specificata nel kernel, per esempio la generazione del core dump e la terminazione del processo, oppure può specificare *SIG\_IGN* come  $<function>$  per indicare che l'occorrenza del segnale deve essere ignorata.

Il kernel utilizza la *u-area* di un processo per annotare le azioni specificate per la gestione dei segnali e un insieme di bit nella struttura *proc* per registrare l'occorrenza dei segnali. Ogni volta che un segnale è inviato al processo, il bit corrispondente al segnale è impostato a 1 nella struttura *proc* del processo destinazione. Il kernel a questo punto determina se il segnale è stato ignorato dal processo destinatario. In caso negativo, inoltra il segnale al processo. Se un segnale è ignorato, rimane pendente e viene consegnato quando il processo manifesta il suo interesse alla ricezione del segnale (o specificando un'azione o specificando che l'azione di default dovrà essere adoperata per la sua gestione). Un segnale rimane pendente se il processo destinatario si trova nello stato *blocked*. Il segnale è consegnato quando il processo esce dallo stato *blocked*. In generale, il kernel controlla se ci sono segnali pendenti quando un processo riprende l'esecuzione in seguito a una chiamata di sistema o a un interrupt, quando un processo viene sbloccato e prima che un processo si blocchi su un evento.

L'esecuzione dell'azione per la gestione del segnale è implementata come descritto precedentemente nel Paragrafo 5.2.6. Alcune anomalie esistono nel modo in cui sono gestiti i segnali. Se un segnale si verifica ripetutamente, il kernel annota semplicemente che si è verificato, ma non conta il numero di occorrenze. Per cui il gestore del segnale può essere eseguito una o più volte, in base a quando il processo viene schedulato per eseguire il gestore del segnale. Un'altra anomalia riguarda un segnale inviato a un

processo che è bloccato a causa di una system call. Dopo l'esecuzione del signal handler, questo processo non riprende l'esecuzione della system call. Invece, ritorna dalla system call. Se necessario, può dover ripetere la chiamata. La [Tabella 5.9](#) elenca alcuni segnali interessanti di UNIX.

| Segnale | Descrizione   |
|---------|---|
| SIGCHLD | Un processo figlio ha concluso l'esecuzione o è sospeso |
| SIGFPE  | Errore aritmetico                                       |
| SIGILL  | Istruzione non valida (illegal instruction)             |
| SIGINT  | Tty interrupt (Control-C)                               |
| SIGKILL | Kill del processo                                       |
| SIGSEGV | Segmentation fault                                      |
| SIGSYS  | System call non valida                                  |
| SIGXCPU | Superato il limite del tempo di CPU                     |
| SIGXFSZ | Superato il limite della dimensione del file            |

**Tabella 5.9** Segnali interessanti in Unix.

### ***Stati del processo e transizioni di stato***

Esiste un'importante differenza tra il modello di processo descritto nel Paragrafo 5.2.1 e quello utilizzato in Unix. Nel modello del Paragrafo 5.2.1, un processo nello stato *running* viene impostato a *ready* nel momento in cui viene interrotta la sua esecuzione. Un processo di sistema gestisce l'evento che ha causato l'interrupt. Nel caso in cui il processo in esecuzione ha esso stesso causato un'interruzione software eseguendo una *<SI\_instrn>*, il suo stato può ulteriormente cambiare a *blocked* se la richiesta non può essere immediatamente soddisfatta. In questo modello un processo utente esegue solo codice utente, per cui non necessita di privilegi speciali. Un processo di sistema può dover usare istruzioni privilegiate come l'avvio dell'I/O e l'impostazione delle informazioni relative alla protezione della memoria, per cui il processo di sistema viene eseguito con la CPU in modalità kernel.

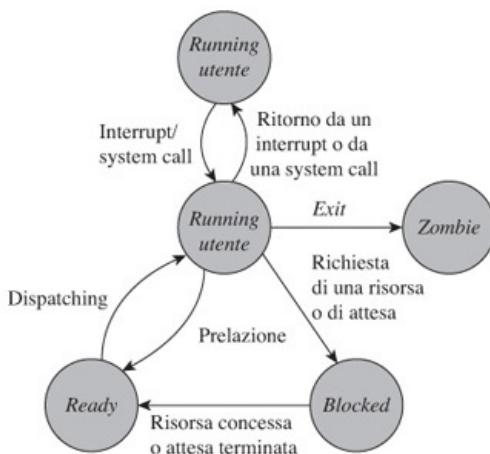
I processi hanno un comportamento diverso nel modello Unix. Quando un processo effettua una system call, il processo stesso esegue il codice kernel per la gestione della system call. Per assicurare che abbia i privilegi necessari, deve essere eseguito con la CPU in modalità kernel. Per cui è necessario effettuare un cambio di modalità ogni volta che viene effettuata una chiamata di sistema. Il cambio di modalità opposto è necessario al termine dell'esecuzione della chiamata. Si attua un procedimento analogo è richiesto quando, in conseguenza di un interrupt, un processo avvia la sua gestione nel kernel e quando ritorna al termine della gestione dell'interrupt.

Il codice del kernel di Unix è rientrante, in modo che più processi possano eseguirlo concorrentemente. Questa caratteristica gestisce la situazione per cui un processo viene bloccato mentre esegue il codice kernel, per esempio quando esegue una system call per avviare un'operazione di I/O o se effettua una richiesta che non può essere soddisfatta immediatamente. Per garantire che il codice sia rientrante, ogni processo che esegue il codice del kernel deve utilizzare un proprio stack kernel. Questo stack contiene la storia delle chiamate di funzione dal momento in cui il processo ha iniziato l'esecuzione del codice kernel. Se anche un altro processo inizia a eseguire il codice del kernel, la storia delle sue chiamate di funzione sarà conservata nel suo stack kernel. In questo modo, le rispettive operazioni non interferiranno l'una con l'altra. In linea di principio, lo stack kernel di un processo non ha bisogno di essere distinto dallo stack utente; tuttavia, in pratica, vengono utilizzati stack differenti poiché molte architetture utilizzano stack differenti quando la CPU è in modalità kernel e in modalità utente.

Unix utilizza due distinti stati *running*. Questi stati sono chiamati stato *running utente* e stato *running kernel*. Un processo utente esegue codice utente quando è nello stato *running utente* ed esegue codice kernel quando è nello stato *running kernel*. Una transizione da *running utente* a *running kernel* si verifica quando il processo effettua una system call o quando viene generato un interrupt. Il processo può essere bloccato mentre si trova nello stato *running kernel* a causa di un'operazione di I/O o della non disponibilità di una risorsa. Quando l'operazione di I/O termina o viene concesso l'uso della risorsa, il processo ritorna allo stato *running kernel* e completa l'esecuzione del

codice kernel che stava eseguendo. A questo punto lascia la modalità kernel e ritorna alla modalità utente. Di conseguenza, il suo stato viene modificato da *running kernel* a *running utente*.

A causa di questa organizzazione, un processo non può bloccarsi o essere prelazionato quando si trova nello stato *running utente*; prima deve effettuare una transizione allo stato *running kernel* e successivamente può bloccarsi o essere prelazionato. Infatti, *user running* → *kernel running* è l'unica transizione per uscire dallo stato *running utente*. La Figura 5.19 illustra gli stati fondamentali in cui può trovarsi un processo e le transizioni di stato in Unix. Come mostrato, anche la terminazione di un processo si verifica quando un processo è nello stato *running kernel*. Questo succede perché il processo esegue la chiamata di sistema *exit* mentre si trova nello stato *running utente*. Questa chiamata modifica il suo stato a *running kernel*. Il processo termina e diventa un processo *zombie* come conseguenza di questa chiamata.



**Figura 5.19** Transizioni di stato di un processo in Unix.

## 5.4.2 Processi e thread in Linux

### Strutture Dati

La versione 2.6 del kernel Linux supporta il modello di thread 1 : 1, ovvero i thread di livello kernel. Utilizza un *descrittore di processo*, che è una struttura dati del tipo *task\_struct*, per memorizzare tutte le informazioni riguardanti un processo o un thread. Per un processo, questa struttura dati contiene lo stato del processo, le informazioni riguardanti il suo processo genitore e i suoi processi figli, il terminale usato dal processo, la directory corrente, i file aperti, la memoria a esso allocata, i segnali e i gestori dei segnali. Il kernel crea sottostruuture per memorizzare le informazioni riguardanti il terminale, la directory, i file, la memoria e i segnali e memorizza i puntatori a queste strutture nel descrittore del processo. Questa organizzazione consente sia di risparmiare memoria sia di ridurre l'overhead quando viene creato un thread.

### Creazione e terminazione di processi e thread

Sia i processi che i thread sono creati utilizzando la system call *fork* e *vfork*, le cui funzionalità sono identiche alle corrispondenti chiamate Unix. Queste funzionalità sono di fatto implementate dalla system call *clone*, che però è trasparente ai programmi. La chiamata di sistema *clone* prende quattro parametri: l'indirizzo base del processo o del thread, i parametri da passare al processo o al thread da creare, le flag e la specifica di uno stack figlio. Alcune flag importanti sono:

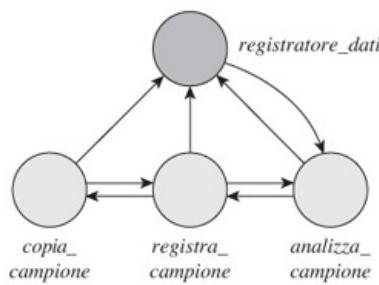
|          |  |
|----------|--|
| CLONE_VM | Condivide le informazioni sulla gestione della memoria usate dalla MMU |
| CLONE_FS | Condivide le informazioni sulla directory root e la directory corrente |

|               |  |
|---------------|--|
| CLONE_FILES   | Condivide le informazioni sui file aperti                |
| CLONE_SIGHAND | Condivide le informazioni sui segnali e i signal handler |

L'organizzazione della struttura `task_struct` facilita la condivisione selettiva di queste informazioni dal momento che contiene solo puntatori alle sottostrutture dove sono contenute le informazioni. Nel momento in cui viene effettuata un chiamata, il kernel effettua una copia della struttura `task_struct` nella quale vengono copiati alcuni di questi puntatori e altri sono modificati. Un thread è creato invocando la `clone` con tutte le flag settate, in modo che il nuovo thread condivide con il suo genitore lo spazio di indirizzamento, i file e i signal handler. Un processo è creato invocando `clone` con tutte le flag non settate, così che il nuovo processo non condivide nessuna di queste componenti. La versione 2.6 del kernel Linux inoltre include il supporto per la Native POSIX Threading Library (NPTL), la quale mette a disposizione alcuni miglioramenti che portano notevoli benefici alle applicazioni che utilizzano i thread. Può supportare fino a due miliardi di thread, mentre la versione 2.4 del kernel Linux poteva supportare solo fino a 8192 thread per ogni CPU. È stata introdotta una nuova system call `exit_group()` per terminare un processo e tutti i suoi thread, capace di terminare un processo con centomila thread in circa 2 secondi, contro i circa 15 minuti necessari nella versione 2.4 del kernel Linux. La gestione dei segnali viene effettuata in kernel space; un segnale è consegnato a uno dei thread disponibili in un processo. I segnali di blocco e sblocco coinvolgono un intero processo, mentre i segnali di errore lo terminano. Queste caratteristiche semplificano la gestione dei processi multithread. Inoltre, la versione 2.6 del kernel Linux supporta un mutex veloce in user space chiamato `futex` che riduce l'overhead della sincronizzazione dei thread effettuando un numero ridotto di system call.

### Relazioni genitore-figlio

Le relazioni relative ai processi o thread genitori e figli sono memorizzate in una `task_struct` che conserva la conoscenza dell'albero dei processi. La struttura `task_struct` contiene un puntatore al genitore e al processo da considerare genitore, ovvero un processo cui riportare lo stato di terminazione del processo nel caso in cui il genitore sia già terminato, un puntatore al figlio creato più di recente e un puntatore al fratello più giovane e più vecchio del processo. In questo modo, l'albero dei processi di [Figura 5.2](#) sarebbe rappresentato come mostrato in [Figura 5.20](#).



**Figura 5.20** Albero dei processi in Linux relativo ai processi di [Figura 5.2\(a\)](#).

### Stati di un processo

Il campo `state` di un descrittore di processo contiene una flag che indica lo stato di un processo. Un processo può essere in uno di cinque stati:

|                      |   |
|----------------------|---|
| TASK_RUNNING         | Il processo o è stato schedulato o è in attesa di essere schedulato.  |
| TASK_INTERRUPTIBLE   | Il processo è in attesa di un evento, ma può ricevere un segnale.     |
| TASK_UNINTERRUPTIBLE | Il processo è in attesa di un evento, ma non può ricevere un segnale. |
| TASK_STOPPED         | L'esecuzione di un processo è stata interrotta con un                 |

segnale.

`TASK_ZOMBIE`

Il processo ha terminato la sua esecuzione, ma il processo genitore non ha ancora effettuato una chiamata di sistema della famiglia `wait` per controllare il suo stato di uscita.

Lo stato `TASK_RUNNING` corrisponde allo stato *running* o *ready* descritti nel Paragrafo 5.2.1. Gli stati `TASK_INTERRUPTIBLE` e `TASK_UNINTERRUPTIBLE` corrispondono entrambi allo stato *blocked*. La divisione dello stato *blocked* in due stati risolve il problema della gestione dei segnali inviati a un processo nello stato *blocked* (Paragrafo 5.2.6) – un processo può decidere se vuole essere attivato da un segnale mentre è in attesa di un evento, oppure se vuole che la consegna del messaggio sia ritardata finché non esce dallo stato *blocked*. Un processo entra nello stato `TASK_STOPPED` quando riceve un segnale `SIGSTOP` o `SIGSTP` per indicare che la sua esecuzione deve essere fermata, o un `SIGTTIN` o `SIGTTOUT` per indicare che un processo in background richiede input o output.

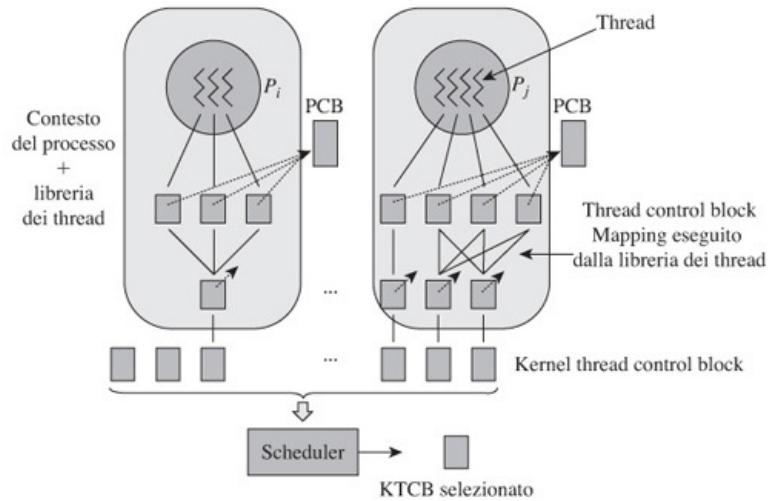
### 5.4.3 Thread in Solaris

Solaris, un sistema operativo basato su Unix 5.4, originariamente utilizzava un modello di thread ibrido che di fatto supportava tutti e tre metodi di associazione dei thread ibridi discussi nel Paragrafo 5.3.2, cioè, molti a uno, uno a uno e molti a molti. Questo modello è stato chiamato, nella letteratura Sun, il *modello M × N*. Solaris 8 ha continuato a supportare questo modello e inoltre implementa un modello alternativo 1 : 1, che risulta essere equivalente ai thread kernel. Il supporto per il modello *M × N* è stato abbandonato in Solaris 9. In questo paragrafo illustreremo il modello *M × N* e i motivi per i quali è stato abbandonato.

Il *modello M × N* adotta tre tipi di entità per governare la concorrenza e il parallelismo all'interno di un processo.

- *Thread utente*: i thread utente sono analoghi ai thread di livello utente illustrati nel Paragrafo 5.3.2; sono creati e gestiti dalla libreria dei thread, per cui non sono visibili al kernel.
- *Processi lightweight*: un processo *lightweight* (LWP) rappresenta una via di mezzo tra i thread utente e i thread kernel. Molti LWP possono essere creati per un processo; ogni LWP è un'unità di parallelismo all'interno di un processo. I thread utente sono mappati sugli LWP dalla libreria dei thread. Questa mappatura può essere uno a uno, molti a uno, molti a molti o una combinazione dei tre. Il numero di LWP per ogni processo e la natura della mappatura tra i thread utente e gli LWP viene decisa dal programmatore, che la rende nota alla libreria mediante l'uso di opportune chiamate di funzione.
- *Thread kernel*: un thread kernel è un thread di livello kernel. Il kernel crea un kernel thread per ogni LWP in un processo. Inoltre crea alcuni thread kernel per uso proprio, per esempio per gestire l'I/O sul disco.

La [Figura 5.21](#) illustra un'organizzazione dei thread utente, degli LWP e dei thread kernel. Il processo  $P_i$  ha tre thread utente e un LWP, per cui viene applicata la mappatura molti a uno. Il processo  $P_j$  ha quattro thread utente e tre LWP. Uno di questi thread utente è mappato su un LWP. I restanti tre thread utente e i due LWP sono mappati col metodo molti a molti; in questo modo ognuno dei tre thread può essere eseguito in uno dei due LWP.



**Figura 5.21** Thread in Solaris.

Gli LWP possono funzionare in parallelo poiché ognuno di essi è associato a un thread kernel. Il kernel crea un LWP control block per ogni LWP e un *kernel thread control block* (KTCB) per ogni thread kernel. Inoltre, la libreria dei thread crea un thread control block per ogni thread utente. L'informazione in questo control block è analoga a quella descritta nel Paragrafo 5.3.2. Lo scheduler esamina i KTCB e, per ogni CPU nel sistema, seleziona un thread kernel che si trova nello stato *ready*. Il LWP corrispondente a questo thread kernel viene sottoposto a dispatch. La libreria dei thread può commutare tra i thread utente mappati su questo LWP per ottenere la concorrenza tra i thread utente. Il numero di LWP per ogni processo e l'associazione dei thread utente con gli LWP viene decisa dal programmatore, così sia il parallelismo sia la concorrenza all'interno del processo sono sotto il controllo del programmatore. Un parallelismo a  $n$ -vie sarebbe possibile all'interno di un processo se il programmatore creasse  $n$  LWP per un processo,  $1 \leq n \leq p$ , dove  $p$  è il numero di CPU. Tuttavia, il grado di parallelismo sarebbe ridotto se un thread utente effettuasse una chiamata di sistema bloccante durante la sua esecuzione, poiché la chiamata bloccherebbe il LWP sul qual è mappato. Per risolvere questo problema, Solaris fornisce gli *scheduler activation*, descritti successivamente in questo paragrafo.

Per controllare la commutazione tra i thread kernel viene utilizzata una complessa organizzazione di control block. Il kernel thread control block contiene i registri kernel della CPU, il puntatore allo stack, la priorità, le informazioni per la schedulazione e un puntatore al prossimo KTCB in una lista di schedulazione. Inoltre, contiene un puntatore all'LWP control block. Il LWP control block contiene i valori salvati dei registri utente della CPU, le informazioni relative alla gestione dei segnali e un puntatore al PCB del processo all'interno del quale è in esecuzione.

### Gestione dei segnali

I segnali generati dal funzionamento di un thread, come le eccezioni aritmetiche o le violazioni di protezione della memoria, sono consegnati allo stesso thread. I segnali generati da fonti esterne, come i timer, devono essere indirizzati a un thread che ha abilitato la loro gestione. Il modello  $M \times N$  fornisce a ogni processo un LWP dedicato alla gestione dei segnali. Quando viene generato un segnale, il kernel lo mantiene in sospeso e notifica la ricezione a questo LWP, il quale attende finché non riscontra che qualche thread, abilitato alla gestione di quel segnale specifico, non sia in esecuzione su uno degli LWP del processo; a questo punto, richiede al kernel di consegnare il segnale sospeso all'LWP precedentemente individuato.

### Stati dei processi e thread kernel

Il kernel è a conoscenza solo degli stati dei processi e dei thread kernel; non è a conoscenza dell'esistenza dei thread utente. Un processo può essere in uno dei seguenti stati:

|         |   |
|---------|---|
| SIDL    | Uno stato transiente durante la creazione |
| SRUN    | Processo eseguibile                       |
| SONPROC | In esecuzione su un processore            |
| SSLEEP  | Sleeping                                  |
| SSTOP   | Stopped                                   |
| SZOMB   | Processo terminato                        |

Gli stati SRUN e SSLEEP corrispondono agli stati *ready* e *blocked*. Un thread kernel può trovarsi negli stati TS\_RUN, TS\_ONPROC, TS\_SLEEP, TS\_STOPPED e TS\_ZOMB che sono analoghi ai corrispondenti stati di un processo. Un thread kernel libero si trova nello stato TS\_FREE.

### **Scheduler activation**

Uno scheduler activation è come un thread kernel. Il kernel usa lo scheduler activation per effettuare due funzioni: (1) quando qualche LWP di un processo si blocca, il kernel usa uno scheduler activation per creare un nuovo LWP in modo che altri thread eseguibili del processo possano essere eseguiti. (2) Quando si verifica un evento relativo all'esecuzione della libreria dei thread, il kernel usa uno scheduler activation per notificarlo alla libreria dei thread.

Si consideri una associazione molti a uno tra molti thread utente e un LWP e un thread utente attualmente associato a un LWP. Un thread kernel è associato con il LWP, cosicché il thread utente venga eseguito quando il thread kernel è schedulato. Se il thread utente effettuasse una system call bloccante, il thread kernel si bloccherebbe, in quanto LWP con il qual è associato si bloccherebbe. Se qualcuno degli altri thread che sono associati allo stesso LWP fosse eseguibile, ci troveremmo nella situazione in cui un thread utente eseguibile non può essere schedulato perché il LWP è bloccato.

In questa situazione, il kernel crea uno scheduler activation quando il thread utente sta per bloccarsi, lo mette a disposizione della libreria dei thread ed effettua una *upcall* a esso. La upcall è implementata come un segnale inviato alla libreria dei thread, che esegue il signal handler utilizzando lo scheduler activation fornito dal kernel. Il gestore del segnale salva lo stato del thread utente che sta per bloccarsi, rilascia l'LWP utilizzato e lo passa al kernel per essere riutilizzato. A questo punto schedula un nuovo thread utente sul nuovo scheduler activation fornito dal kernel. In effetti, il thread utente che stava per bloccarsi viene rimosso da un LWP e un nuovo thread utente viene schedulato su un nuovo LWP del processo. Quando si verifica l'evento per il quale il thread utente si è bloccato, il kernel effettua un'altra upcall alla libreria dei thread con uno scheduler activation in modo che possa prelazionare il thread utente attualmente associato al LWP, restituire il LWP al kernel e schedulare il thread nuovamente attivato sullo scheduler activation fornito dal kernel.

### **Passaggio all'implementazione 1:1**

Il modello  $M \times N$  fu sviluppato nella convinzione che, poiché un context switch effettuato dalla libreria dei thread genera molto meno overhead rispetto al context switch effettuato dal kernel, la schedulazione a livello utente dei thread nella libreria avrebbe fornito buone prestazioni per le applicazioni. Tuttavia, come puntualizzato nel Paragrafo 5.3.2, questo è vero solo quando lo scheduler della libreria dei thread e quello del kernel lavorano in maniera coordinata. L'implementazione 1 : 1 in Solaris 8 forniva invece un efficiente context switch a livello kernel. L'utilizzo del modello 1 : 1 conduce a una gestione dei segnali più semplice, dal momento che i thread possono essere dedicati alla gestione di segnali specifici. Inoltre elimina la necessità dello scheduler activation e fornisce una migliore scalabilità. Per questi motivi, il modello  $M \times N$  è stato eliminato a partire da Solaris 9.

## **5.4.4 Processi e thread in Windows**

L'essenza dei processi e dei thread in Windows è alquanto differente da quella presentata precedentemente in questo capitolo. Windows considera un processo come una unità di allocazione delle risorse e usa un thread come unità per la concorrenza. Di conseguenza, un processo Windows non viene eseguito; deve avere almeno un thread al suo interno. Si può accedere a una risorsa solo attraverso un *resource handle*. Un processo eredita

alcuni resource handle dal suo processo genitore; può ottenere più resource handle aprendo nuove risorse. Il kernel memorizza, per ogni processo, tutti questi handle in una tabella. In questo modo, si può accedere a una risorsa semplicemente specificando un offset nella tabella degli handle.

Windows utilizza tre control block per gestire un processo. Un *executive process block* contiene i campi per memorizzare l'id del processo, le informazioni per la gestione della memoria, l'indirizzo della tabella degli handle, un kernel process block per il processo e l'indirizzo del process environment block del processo. Il *kernel process block* contiene le informazioni di scheduling per i thread del processo, come l'affinità al processore per il processo, lo stato del processo e i puntatori ai kernel thread block dei suoi thread. L'*executive process block* e il *kernel process block* sono memorizzati nello spazio di indirizzamento del sistema. Il *process environment block*, memorizzato nello spazio di indirizzamento dell'utente, contiene le informazioni usate dal loader per caricare il codice da eseguire e dal gestore dell'heap.

I control block impiegati per gestire un thread contengono le informazioni riguardanti la sua esecuzione e il processo che lo contiene. L'*executive thread block* di un thread contiene un *kernel thread block*, un puntatore all'*executive process block* del suo processo genitore e informazioni sui suoi parametri. Il *kernel thread block* contiene informazioni riguardanti lo stack kernel del thread e la memoria locale del thread, le informazioni di shedulazione per il thread e un puntatore al suo *thread environment block*, che contiene il suo id e le informazioni riguardanti i suoi requisiti di sincronizzazione.

Windows supporta la nozione di *job* come metodo di gestione di un gruppo di processi. Un job è rappresentato da un oggetto propriamente detto job, che contiene le informazioni come gli handle ai processi in esso contenuti, il limite di tempo di CPU per il job, il limite di tempo di CPU per ogni processo, la classe di scheduling del job che imposta la quantità di tempo per ogni processo del job, l'associazione al processore per i processi del job e la classe di priorità. Un processo può appartenere a un solo job; tutti i processi creati, figli di un processo, appartengono allo stesso job.

### ***Creazione di un processo***

La chiamata *create* richiede un parametro che rappresenta un handle al genitore del nuovo processo o thread. In questo modo, una chiamata *create* non ha bisogno di essere invocata dal genitore di un processo o di un thread. Un processo server utilizza questa caratteristica per creare un thread in un processo client in modo da poter accedere alle risorse con i privilegi di accesso del client, piuttosto che con i suoi privilegi.

Si ricordi dal Paragrafo 4.8.4 che il sottosistema dell'ambiente fornisce il supporto per l'esecuzione dei programmi sviluppati per altri SO come MS-DOS, Win 32 e SO/2. La semantica della creazione del processo dipende dal sottosistema dell'ambiente utilizzato da un processo. Negli ambienti di elaborazione Win/32 e SO/2, un processo possiede un thread quando viene creato; non avviene la stessa cosa in altri ambienti supportati dal SO Windows. Dunque la creazione di un processo viene di fatto gestita da una DLL del sottosistema dell'ambiente che è linkata al processo. Dopo la creazione di un processo, l'id del nuovo processo o thread viene passato al processo del sottosistema dell'ambiente in modo che possa essere gestito in modo appropriato.

La creazione di un processo figlio da parte di un processo nell'ambiente Win/32 avviene come segue: la DLL del sottosistema dell'ambiente linkata al processo effettua una system call per creare un nuovo processo. Questa chiamata viene gestita dall'*executive*, che crea un oggetto processo, lo inizializza caricando l'immagine del codice da eseguire e restituisce un handle all'oggetto processo. La DLL del sottosistema dell'ambiente a questo punto effettua una seconda system call per creare un thread e passa l'handle al nuovo processo come parametro. L'*executive* crea un thread nel nuovo processo e gli restituisce un handle. La DLL invia un messaggio al processo del sottosistema dell'ambiente, passandogli gli handle del processo e del thread e l'id del loro processo genitore. Il processo del sottosistema dell'ambiente inserisce l'handle del processo nella tabella dei processi attivi nell'ambiente e inserisce l'handle del thread nella struttura dati relativa alla schedulazione. Infine il controllo ritorna al processo.

### ***Stati dei thread e transizioni di stato***

La [Figura 5.22](#) mostra il diagramma di transizione di stato per i thread. Un thread può trovarsi in uno dei seguenti stati:



**Figura 5.22** Transizioni di stato dei thread in Windows.

1. *ready*, il thread può essere eseguito se la CPU è disponibile;
2. *standby*: il thread è stato selezionato come prossimo thread a essere eseguito su uno specifico processore. Se la sua priorità è alta, il thread attualmente in esecuzione sul processore viene prelazionato e questo thread schedulato;
3. *running*: la CPU è allocata a un thread e il thread è in esecuzione;
4. *waiting*: il thread è in attesa di un risorsa o di un evento, oppure è stato sospeso dal sottosistema dell'ambiente;
5. *transition*: l'attesa del thread è terminata, ma nel frattempo il suo stack kernel è stato rimosso dalla memoria poiché l'attesa è durata troppo tempo. Il thread entra nello stato *ready* quando lo stack kernel è riportato in memoria;
6. *terminated*: il thread ha terminato la propria esecuzione.

#### **Pool di thread**

Windows mette a disposizione un pool di thread per ogni processo. Come descritto nel Paragrafo 5.3, il pool contiene un insieme di thread, un'organizzazione di liste per le richieste in attesa e i thread inutilizzati. I thread di un pool possono essere usati per eseguire lavoro sottoposto al pool, o possono essere programmati per effettuare compiti specifici in determinati istanti di tempo, periodicamente o alla ricezione di determinati segnali da oggetti del kernel. Il numero di thread sono adattati dinamicamente al carico di lavoro del pool. Se il tasso al quale le richieste sono presentate al pool corrisponde al tasso al quale i thread completano il servizio delle richieste, i thread non sono né creati né distrutti. Tuttavia, se il tasso delle richieste supera il tasso di servizio vengono creati nuovi thread mentre un thread viene distrutto se rimane idle per più di 40 secondi. Windows Vista supporta diversi pool di thread in un processo.

## **Riepilogo**

L'utente di un computer e un sistema operativo hanno punti di vista differenti dell'esecuzione dei programmi. L'utente è interessato a ottenere l'esecuzione di un programma in maniera sequenziale o concorrente, mentre il SO è interessato all'allocazione delle risorse ai programmi e al servizio di più programmi simultaneamente, per cui può essere ottenuta una combinazione adeguata di uso efficiente e servizio per l'utente. In questo capitolo, abbiamo discusso diversi aspetti di questi due punti di vista in relazione all'esecuzione dei programmi.

L'esecuzione di un programma può essere velocizzata utilizzando sia il *parallelismo* sia la *concorrenza*. Il parallelismo implica che, all'interno del programma, vengano effettuate diverse attività allo stesso tempo. La concorrenza è invece l'illusione del parallelismo: le attività sembrano eseguite in parallelo, ma di fatto non lo sono.

Un *processo* è un modello di esecuzione di un programma. Quando un utente invoca un comando per eseguire un programma, il SO crea un processo principale. Questo processo può creare altri processi effettuando richieste al SO mediante l'uso di system call; ognuno di questi processi è chiamato *processo figlio*. Il SO può servire un

processo e alcuni dei suoi processi figli in maniera concorrente eseguendo, sulla stessa CPU, le istruzioni di ognuno di essi per un certo periodo di tempo, o servirli in parallelo eseguendo le loro istruzioni su diverse CPU allo stesso tempo. I processi all'interno di un programma devono essere eseguiti in modo armonico per un obiettivo comune, condividendo dati o coordinando le loro attività l'uno con l'altro. Questo risultato è ottenuto adoperando le tecniche di *sincronizzazione dei processi* messe a disposizione dal sistema operativo.

Il sistema operativo alloca le risorse a un processo e memorizza le informazioni a esse relative nel *contesto del processo*. Per controllare l'esecuzione di un processo il SO usa la nozione di *stato del processo*. Lo stato del processo è una descrizione dell'attività corrente del processo; lo stato del processo cambia con l'esecuzione del processo. Gli stati fondamentali di un processo sono: *ready*, *running*, *blocked*, *terminated* e *suspended*. Il SO mantiene le informazioni riguardanti ogni processo in un *process control block* (PCB). Il PCB di un processo contiene lo stato di un processo e lo stato della CPU relativo al processo se la CPU non sta eseguendo le sue istruzioni. La funzione di scheduling del kernel seleziona uno dei processi *ready* e la funzione di dispatch commuta la CPU al processo selezionato utilizzando le informazioni memorizzate nel contesto del processo e nel PCB.

Un *thread* rappresenta un modello alternativo di esecuzione di un programma. Un thread differisce da un processo nel fatto che a esso non sono allocate risorse. Questa differenza riduce l'overhead della commutazione tra thread rispetto all'overhead della commutazione tra processi. Tre sono i modelli di thread utilizzati: *thread id livello kernel*, *thread id livello utente* e *thread ibrido*. Ognuno di essi ha differenti implicazioni sull'overhead della commutazione, sulla concorrenza e sul parallelismo.

## Domande

- 5.1. Un'applicazione comprende diversi processi – un processo principale e alcuni processi figli. Questa organizzazione consente di velocizzare l'elaborazione se:
  - a. il computer dispone di molte CPU;
  - b. alcuni processi sono I/O bound;
  - c. alcuni processi sono CPU bound;
  - d. nessuna delle precedenti.
- 5.2. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. Il SO crea un singolo processo se due utenti eseguono lo stesso programma.
  - b. Lo stato di un processo *bloccato* sulla richiesta di una risorsa cambia a *running* quando viene concesso l'uso della risorsa.
  - c. Non c'è distinzione tra un processo terminato e uno sospeso.
  - d. Dopo aver gestito un evento, il kernel non ha bisogno di effettuare la schedulazione prima del dispatching se non cambia lo stato di nessun processo.
  - e. Quando un thread utente di un processo effettua una system call che porta a un blocco, tutti i thread del processo si bloccano.
  - f. I thread kernel forniscono maggiore concorrenza dei thread utente sia nei sistemi con un singolo processore che in quelli multiprocessore.
  - g. Quando un processo termina, il suo codice di terminazione dovrebbe essere memorizzato finché il suo processo genitore non termina.
- 5.3. Quale delle seguenti transizioni di stato per un processo può causare la transizione di stato *blocked* → *ready* per uno o più degli altri processi?
  - a. Un processo avvia un'operazione di I/O e passa nello stato *blocked*.
  - b. Un processo termina.
  - c. Un processo effettua la richiesta per una risorsa e passa nello stato *blocked*.
  - d. Un processo invia un messaggio.
  - e. Un processo effettua una transizione distata *blocked* → *blockedswapped*.

## Problemi

- 5.1. Descrivere le azioni del kernel quando i processi effettuano le chiamate di sistema per i seguenti scopi.

- a. Richiesta di ricevere un messaggio.
  - b. Richiesta di effettuare un'operazione di I/O.
  - c. Richiesta dell'informazione di stato riguardante un processo.
  - d. Richiesta di creazione di un processo.
  - e. Richiesta di terminazione di un processo figlio.
- 5.2. Descrivere in quali condizioni un kernel può effettuare il dispatching senza eseguire lo scheduling.
- 5.3. Scrivere un algoritmo per implementare un chiamata *wait* nello stile Unix utilizzando la struttura dati PCB mostrata nella [Tabella 5.6](#). Commentare la durata di un processo e del suo PCB.
- 5.4. Descrivere come ogni segnale elencato nella [Tabella 5.9](#) viene generato e gestito in Unix.
- 5.5. Un processo si trova nello stato *blocked swapped*.
  - a. Elencare una sequenza di stati attraverso cui il processo ha raggiunto questo stato.
  - b. Elencare una sequenza di transizioni di stato attraverso cui il processo può raggiungere lo stato *ready*.
- Esiste più di una sequenza di transizioni di stato in ognuno di questi casi?
- 5.6. Il progettista di un kernel ha deciso di utilizzare un singolo stato *swapped*. Creare un diagramma analogo a quello di [Figura 5.5](#) che mostri gli stati del processo e le transizioni di stato. Descrivere come il kernel effettuerebbe lo swapping e commentare i vantaggi dello swapping.
- 5.7. Confrontare il parallelismo intrinseco nelle seguenti applicazioni.
  - a. Un'applicazione di online banking che consente agli utenti di effettuare transazioni bancarie mediante un browser web.
  - b. Un sistema di prenotazione web-based per una compagnia area.
- 5.8. Un sistema di prenotazione per una compagnia area che utilizza un database centralizzato, fornisce servizio alle richieste degli utenti in maniera concorrente. In questa applicazione, è preferibile utilizzare i thread piuttosto che i processi? Argomentare la risposta.
- 5.9. Elencare due system call che un thread dovrebbe evitare di utilizzare se i thread sono implementati a livello utente e spiegare le ragioni.
- 5.10. Come descritto nell'Esempio 5.5 e illustrato nella [Figura 5.16](#), se un processo utilizza i thread utente, il suo stato dipende dallo stato di tutti i suoi thread. Elencare le possibili cause di ognuna delle transizioni di stato fondamentali per tale processo.
- 5.11. Commentare la seguente affermazione sulla base di quanto letto in questo capitolo: "La concorrenza aumenta l'overhead dovuto allo scheduling senza fornire alcuno speedup di un programma".
- 5.12. Sulla base del caso di studio Solaris, descrivere brevemente in che modo decidere il numero dei thread utente e dei processi lightweight (LWP) che dovrebbero essere creati in un'applicazione.
- 5.13. Un SO supporta sia i thread utente sia i thread kernel. Si è d'accordo con le seguenti raccomandazioni su quando usare i thread utente e quando usare i thread kernel? Perché o perché no?
  - a. Se l'elaborazione da eseguire in un thread è CPU bound, creare un thread kernel se il sistema dispone di più CPU; in caso contrario creare un thread utente.
  - b. Se l'elaborazione da eseguire in un thread è I/O bound, creare un thread utente se il processo che lo contiene non ha un thread kernel; in caso contrario, creare un thread kernel.
- 5.14. Commentare lo speedup dell'elaborazione delle seguenti applicazioni in un sistema che ha (i) una CPU e (ii) più CPU.
  - a. In un server che gestisce molte richieste a un tasso molto elevato vengono creati molti thread, dove il servizio di una richiesta coinvolge sia attività di CPU che di I/O.
  - b. L'elaborazione di un'espressione  $z := a * b + c * d$  viene effettuata creando due processi figli che valutano  $a * b$  e  $c * d$ .
  - c. Un server crea un nuovo thread per gestire ogni nuova richiesta di un utente e il servizio di ognuna di queste richieste richiede accessi a un database.
  - d. Due matrici contengono  $m$  righe e  $n$  colonne ognuna, dove  $m$  ed  $n$  sono entrambi numeri molto grandi. Un'applicazione ottiene il risultato della somma delle due matrici creando  $m$  thread, ognuno dei quali effettua la somma di una riga delle

matrici.

- 5.15. Calcolare il miglior speedup dell'elaborazione nell'applicazione di registrazione dati realtime dell'Esempio 5.1 nelle seguenti condizioni: l'overhead per la gestione degli eventi e per la commutazione dei processi è trascurabile. Per esempio, il processo *copia\_campione* richiede 5 microsecondi ( $\mu = \mu s$ ) di tempo di CPU e non richiede nessuna operazione di I/O, *registra\_campione* richiede 1.5 ms per memorizzare il campione e utilizza solo 1  $\mu = \mu s$  di tempo di CPU, mentre *analizza\_campione* consuma 200  $\mu = \mu s$  di tempo di CPU e l'operazione di scrittura richiede 1.5 ms.

## Laboratorio: implementare una shell

Scrivere un programma in C/C++, che simuli una shell di un sistema Unix o Linux. Quando il programma viene lanciato mostrerà il prompt all'utente, accetterà i comandi utente dalla tastiera, lo classificherà e invocherà una routine appropriata per implementarlo. Il comando "system" non deve essere utilizzato nell'implementazione di nessun comando fatta eccezione per il comando *ls*. La shell deve supportare i seguenti

| Comando                                   | Descrizione   |
|---|---|
| <code>cd &lt;directory_name&gt;</code>    | Cambia la directory corrente se l'utente ha i permessi necessari.   |
| <code>ls</code>                           | Elenca le informazioni relative ai file presenti nella directory corrente.  |
| <code>rm</code>                           | Cancella i file indicati. Supporta le opzioni <code>-r</code> , <code>-f</code> , <code>-v</code> .   |
| <code>history xyzn</code>                 | Stampa gli <i>n</i> comandi più recenti utilizzati dall'utente, con il relativo numero. Se <i>n</i> è omesso, stampa tutti i comandi eseguiti dall'utente.  |
| <code>issue xyzn</code>                   | Ripete l' <i>n</i> -esimo comando.  |
| <code>&lt;nome_programma&gt;</code>       | Crea un processo figlio per eseguire <code>&lt;nome_programma&gt;</code> . Supporta gli operatori di redirezione <code>&gt;</code> e <code>&lt;</code> per redirigere l'input e l'output del programma nel file indicato. |
| <code>&lt;nome_programma&gt; &amp;</code> | Il processo figlio per l'esecuzione di <code>&lt;nome_programma&gt;</code> deve essere eseguito in background.  |
| <code>quit</code>                         | Chiude la shell.  |

Dopo aver implementato una shell di base che supporta questi comandi, possono essere aggiunte due funzionalità avanzate:

1. Progettare un nuovo comando che fornisca una funzione utile. Per esempio, si consideri un comando `rmexcept <lista_dei_file>` che rimuove dalla directory corrente tutti i file eccetto quelli specificati in `<lista_dei_file>`.
2. Supportare un comando `<nome_programma> m` che crea un processo figlio per eseguire `nome_programma`, ma lo termina se non conclude l'esecuzione in *m* secondi. (*Suggerimento*: utilizzare una routine appropriata della libreria per consegnare un segnale SIGALRM dopo *m* secondi e utilizzare un signal handler per effettuare le azioni appropriate.)

## Note bibliografiche

Il concetto di processo è discusso in Dijkstra (1968), Brinch Hansen (1973) e Bic e Shaw (1974). Brinch Hansen (1988) descrivono l'implementazione dei processi nel sistema RC 4000.

Marsh et al. (1991) discutono dei thread utente e delle librerie dei thread. Anderson et al. (1992) descrivono l'uso degli scheduler activation per la comunicazione tra il kernel e la libreria dei thread. Engelschall (2000) descrive come i thread utente possono essere implementati in Unix utilizzando le funzionalità standard di Unix e

inoltre riassume le proprietà di altri pacchetti multithread.

Kleiman (1996), Butenhof (1997), Lewis e Berg (1997) e Nichols et al. (1996) affrontano la programmazione con i thread POSIX. Lewis e Berg (2000) descrivono il multithreading in Java.

Bach (1986), McKusick (1996) e Vahalia (1996) descrivono i processi in Unix. Beck et al. (2002) e Bovet e Cesati (2005) descrivono i processi e i thread in Linux. Stevens e Rago (2005) descrivono i processi e i thread in Unix, Linux e BSD; inoltre, illustrano i processi daemon in Unix. O'Gorman (2003) descrive l'implementazione dei segnali in Linux. Eykholt et al. (1992) descrivono i thread in SunOS, mentre Vahalia (1996) e Mauro e McDougall (2006) descrivono i thread e i LWP in Solaris. Custer (1993), Richter (1999) e Russinovich e Solomon (2005) descrivono i processi e i thread in Windows. Vahalia (1996) e Tanenbaum (2001) discutono i thread in Mach.

1. Anderson, T.E., B.N. Bershad, E.D. Lazowska, and H.M. Levy (1992): "Scheduler activations: effective kernel support for the user-level management of parallelism," *ACM Transactions on Computer Systems*, **10** (1), 53-79.
2. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
4. Bic, L., and A.C. Shaw (1988): *The Logical Design of Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
5. Brinch Hansen, P. (1970): "The nucleus of a multiprogramming system," *Communications of the ACM*, **13**, 238-241, 250.
6. Brinch Hansen, P. (1973): *Operating System Principles*, Prentice Hall, Englewood Cliffs, N.J.
7. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
8. Butenhof, D. (1997): *Programming with Pthreads*, Addison-Wesley, Reading, Mass.
9. Custer, H. (1993): *Inside Windows/NT*, Microsoft Press, Redmond, Wash.
10. Dijkstra, E. W. (1968): "The structure of THE multiprogramming system," *Communications of the ACM*, **11**, 341-346.
11. Engelschall, R.S. (2000): "Portable Multithreading: The signal stack trick for user-space thread creation," *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego.
12. Eykholt, J.R., S.R. Kleiman, S. Barton, S. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams (1992): "Beyond multiprocessing: multithreading the Sun OS kernel," *Proceedings of the Summer 1992 USENIX Conference*, 11-18.
13. Kleiman, S., D. Shah, and B. Smaalders (1996): *Programming with Threads*, Prentice Hall, Englewood Cliffs, N.J.
14. Lewis, B., and D. Berg (1997): *Multithreaded Programming with Pthreads*, Prentice Hall, Englewood Cliffs, N.J.
15. Lewis, B., and D. Berg (2000): *Multithreaded Programming with Java Technology*, Sun Microsystems.
16. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
17. Marsh, B.D., M.L. Scott, T.J. LeBlanc, and E.P. Markatos (1991): "First-class user-level thread," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991, 110-121.
18. McKusick, M.K., K. Bostic, M.J. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, Reading, Mass.
19. Nichols, B., D. Buttlar, and J.P. Farrell (1996): *Pthread Programming*, O'Reilly, Sebastopol.
20. O'Gorman, J. (2003): *Linux Process Manager: The internals of Scheduling, Interrupts and Signals*, John Wiley, New York.
21. Richter, J. (1999): *Programming Applications for Microsoft Windows*, 4th ed., Microsoft Press, Redmond, Wash.
22. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed.,

- Microsoft Press, Redmond, Wash.
- 23. Silberschatz, A., P.B. Galvin, and G. Gagne (2005): *Operating System Principles*, 7th ed., John Wiley, New York.
  - 24. Stevens, W.R., and S.A. Rago (2005): *Advanced Programming in the Unix Environment*, 2nd ed., Addison-Wesley, Reading, Mass.
  - 25. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
  - 26. Vahalia, U. (1996): *Unix Internals-The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 6

## Sincronizzazione dei processi: memoria condivisa

---

### Obiettivi di apprendimento

- Definizione del problema della sincronizzazione dei processi
- Race condition
- Sezioni critiche ed operazioni indivisibili
- Tecniche di sincronizzazione
- Problemi classici di sincronizzazione dei processi
- Implementazione della sezione critica: approccio algoritmico, semafori, monitor
- La sincronizzazione nei sistemi operativi: Posix, Unix, Linux, Solaris, Windows

I processi inter comunicanti sono processi concorrenti che condividono dati o che coordinano le loro attività. La *sincronizzazione per l'accesso ai dati* assicura che i dati condivisi non perdano la loro consistenza quando sono aggiornati dai processi inter comunicanti, e viene implementata garantendo ai processi l'accesso ai dati in maniera mutuamente esclusiva. La *sincronizzazione per il controllo* assicura che i processi che interagiscono effettuino le loro attività nell'ordine desiderato. Queste due attività realizzano la *sincronizzazione dei processi*. A tal fine, i computer mettono a disposizione alcune *istruzioni indivisibili* (dette anche *istruzioni atomiche*) per supportare la sincronizzazione dei processi.

Illustreremo le *sezioni critiche*, ovvero porzioni di codice in cui si accede a dati condivisi in maniera mutuamente esclusiva, le *operazioni di invio dei segnali* indivisibili, usate per implementare il controllo della sincronizzazione, e mostreremo come entrambe sono implementate utilizzando istruzioni indivisibili. In seguito, verranno introdotti alcuni problemi classici di sincronizzazione dei processi, che sono rappresentativi dei problemi di sincronizzazione in vari ambiti di applicazione. Analizzeremo i loro requisiti di sincronizzazione e studieremo aspetti importanti per il loro soddisfacimento.

Nella restante parte del capitolo, parleremo dei *semafori* e dei *monitor*, che sono gli strumenti principali per la sincronizzazione nei linguaggi di programmazione e nei sistemi operativi. Vedremo in che modo permettono di soddisfare i requisiti della sincronizzazione dei processi relativi ai classici problemi.

### 6.1 Che cos'è la sincronizzazione dei processi?

In questo capitolo, utilizzeremo il termine *processo* come termine generico sia per i processi che per i thread. Le applicazioni utilizzano i processi concorrenti sia per ottenere un'elaborazione più veloce ([Tabella 5.2](#)), sia per semplificare la propria progettazione, come nei server multithread (Paragrafo 5.3). Come riassunto nella [Tabella 5.7](#), i processi di un'applicazione interagiscono tra di loro per condividere dati, coordinare le loro attività e per scambiare messaggi e segnali. Utilizzeremo la seguente notazione per definire formalmente i *processi inter comunicanti*:

- |                |   |
|----------------|---|
| $read\_set_i$  | insieme di dati letti dal processo $P_i$ e inviati tra processi o segnali ricevuti da $P_i$     |
| $write\_set_i$ | insieme di dati modificati dal processo $P_i$ e inviati tra processi o segnali inviati da $P_i$ |

Utilizzeremo con il termine “aggiornamento di un dato” per la modifica del valore di un dato rispetto al suo precedente valore, per esempio  $x = x + 1$  è un aggiornamento, mentre  $x = 5$  non lo è.

**Definizione 6.1 Processi intercomunicanti** I processi  $P_i$  e  $P_j$  sono *processi intercomunicanti* se il  $write\_set$  di uno dei processi si sovrappone con il  $write\_set$  o con il  $read\_set$  dell'altro, i.e. se:

$$(read\_set_i \cup write\_set_i) \cap (read\_set_j \cup write\_set_j) \neq \emptyset$$

La natura dell'interazione tra processi quando il  $write\_set$  di uno si sovrappone al  $read\_set$  di un altro è evidente; il primo processo può impostare il valore di una variabile che l'altro processo può leggere. La situazione in cui i  $write\_set$  di due processi si sovrappongono è presente nella Definizione 6.1 poiché il modo in cui i processi effettuano le rispettive operazioni di scrittura può condurre a risultati non consistenti, per cui i processi devono cooperare per evitare queste situazioni. I processi che non interagiscono vengono definiti *processi indipendenti*; essi possono essere eseguiti liberamente in parallelo. Tipicamente si presentano due tipi di requisiti nei processi intercomunicanti:

- un processo dovrebbe effettuare una specifica operazione  $op_i$  solo quando sussistono determinate condizioni relative ai dati condivisi. Il processo deve essere ritardato se questi requisiti non sono soddisfatti, esclusivamente quando vuole eseguire l'operazione  $op_i$ , e gli deve essere consentito di riprendere l'esecuzione quando questi requisiti sono stati soddisfatti;
- un processo dovrebbe effettuare un'operazione  $op_i$  solo dopo che qualche altro processo ha effettuato un'altra specifica operazione  $op_i$ . Questo requisito è soddisfatto utilizzando dati condivisi per annotare quale operazione  $op_i$  è stata effettuata, cosicché il processo possa essere ritardato e ripristinato successivamente.

La *sincronizzazione dei processi* è equivalente a individuare le tecniche usate per ritardare e ripristinare i processi e per implementare le interazioni tra processi. La velocità di esecuzione di un processo, o le velocità di esecuzione relative dei processi intercomunicanti, non può essere conosciuta in anticipo a causa di alcuni fattori come la slice di tempo, le priorità dei processi e le attività di I/O dei processi. Pertanto, la tecnica di sincronizzazione dei processi deve essere progettata in modo tale che funzionerà correttamente indipendentemente dalle velocità di esecuzione relative dei processi.

In questo capitolo, utilizzeremo le convenzioni mostrate nella [Figura 6.1](#) per lo pseudocodice dei processi concorrenti.

- La struttura di controllo **Parbegin** *<elenco istruzioni>* **Parend** racchiude il codice che deve essere eseguito in parallelo. (Parbegin sta per parallel-begin e Parend per parallel-end.) Se *<elenco delle istruzioni>* contiene  $n$  elementi, l'esecuzione della struttura di controllo **Parbegin-Parend** produce  $n$  processi e ogni processo consiste nell'esecuzione di un'istruzione dell'*<elenco istruzioni>*. Ad esempio, **Parbegin**  $S_1, S_2, S_3, S_4$  **Parend** crea quattro processi che eseguono, rispettivamente,  $S_1, S_2, S_3$  e  $S_4$ .

L'istruzione che raggruppa le funzioni di un linguaggio di programmazione, come **begin-end**, viene usata se un processo consiste di un singolo blocco di codice invece che di una singola istruzione. Per una questione visiva, rappresentiamo i processi concorrenti creati in una struttura di controllo **Parbegin-Parend** come segue:

|                 |                                  |                                  |                                  |          |
|-----------------|----------------------------------|----------------------------------|----------------------------------|----------|
| <b>Parbegin</b> | $S_{11}$                         | $S_{21}$                         | $\dots$                          | $S_{n1}$ |
|                 | $\vdots$                         | $\vdots$                         |                                  | $\vdots$ |
|                 | $S_{1m}$                         | $S_{2m}$                         | $\dots$                          | $S_{nm}$ |
| <b>Parend</b>   | <u>Processo <math>P_1</math></u> | <u>Processo <math>P_2</math></u> | <u>Processo <math>P_n</math></u> |          |

dove le istruzioni  $S_{11} \dots S_{1m}$  rappresentano le istruzioni del processo  $P_1$ , ecc.

- Le variabili condivise vengono dichiarate prima di un **Parbegin**.
- Le variabili locali vengono dichiarate all'inizio di un processo.
- I commenti sono racchiusi tra “{ }”.
- L'indentazione è utilizzata per mostrare le strutture di controllo.

**Figura 6.1** Convenzioni per la descrizione in pseudocodice dei programmi concorrenti.

## 6.2 Race condition

Nel Paragrafo 5.2.5 abbiamo che l'accesso non coordinato ai dati condivisi può inficiare la consistenza dei dati. Per illustrare tale problema, si consideri che i processi  $P_i$  e  $P_j$  aggiornano il valore dell'elemento condiviso  $d_s$  rispettivamente mediante le operazioni  $a_i$  e  $a_j$ .

$$\begin{aligned} \text{Operazione } a_i: \quad & d_s = d_s + 10; \\ \text{Operazione } a_j: \quad & d_s = d_s + 5; \end{aligned}$$

Sia  $(d_s)_{\text{iniziale}}$  il valore iniziale di  $d_s$  e sia il processo  $P_i$  il primo a effettuare la sua operazione. Il valore di  $d_s$  dopo l'operazione  $a_i$ , sarà  $(d_s)_{\text{iniziale}} + 10$ . Se il processo  $P_j$  effettua l'operazione  $a_j$  ora, il valore risultante di  $d_s$  sarà  $d_{s\text{new}} = ((d_s)_{\text{iniziale}} + 10) + 5$ , ovvero  $d_{s\text{iniziale}} + 15$ . Se i processi effettuano le operazioni in ordine inverso, il nuovo valore di  $d_s$  sarebbe identico.

Se i processi  $P_i$  e  $P_j$  effettuano le loro operazioni in maniera concorrente, potremmo aspettarci che il risultato sia  $(d_s)_{\text{iniziale}} + 15$ ; tuttavia, questo non è garantito. Questa situazione viene definita una *condizione critica* (*race condition*). Questo termine è preso in prestito dall'elettronica, che fa riferimento al principio in base al quale un tentativo di esaminare un valore, o effettuare una misurazione su un'onda, mentre sta cambiando, può condurre a risultati errati. La race condition può essere spiegata come segue: l'operazione  $a_i$  viene solitamente implementata utilizzando tre istruzioni macchina. La prima istruzione carica il valore di  $d_s$  in un registro dati, per esempio il registro  $r_1$ , la seconda istruzione aggiunge 10 al contenuto di  $r_1$  e la terza istruzione memorizza il contenuto di  $r_1$  nuovamente nella locazione assegnata a  $d_s$ . Tale sequenza di istruzioni è indicata come sequenza *load-add-store*. L'operazione  $a_j$  è implementata in maniera simile con una sequenza *load-add-store*. Il risultato dell'esecuzione delle operazioni  $a_i$  e  $a_j$  sarebbe errato se sia  $a_i$  che  $a_j$  venissero eseguite sulla base del vecchio valore  $d_s$ . Questa situazione potrebbe verificarsi se un processo è coinvolto nell'esecuzione della sequenza *load-add-store*, mentre per l'altro processo viene eseguita un'istruzione di *load* prima che questa sequenza sia completata. In questo caso il valore di  $d_s$  alla fine di entrambe le

operazioni sarebbe  $(d_s)_{\text{iniziale}} + 5$  oppure  $(d_s)_{\text{iniziale}} + 10$ , in base a quale operazione è stata completata dopo.

Definiamo in maniera formale una race condition come segue: sia  $f_i(d_s)$  la funzione che rappresenta l'operazione  $a_i$  su  $d_s$ , ovvero per un dato valore di  $d_s$ ,  $f_i(d_s)$  indica il valore che  $d_s$  assume dopo l'esecuzione dell'operazione  $a_i$ . In modo analogo, la funzione  $f_j(d_s)$  rappresenta l'operazione  $a_j$  su  $d_s$ . Sia il processo  $P_i$  il primo a eseguire una operazione. Il valore di  $d_s$  dopo l'operazione sarebbe  $f_i(d_s)$ . Se il processo  $P_j$  esegue l'operazione  $a_j$ , ora, l'operazione  $a_j$  verrà eseguita su  $f_i(d_s)$ , per cui il valore risultante di  $d_s$  sarà  $f_j(f_i(d_s))$ . Se i processi effettuano le loro operazioni nell'ordine inverso, il nuovo valore di  $d_s$  sarà  $f_i(f_j(d_s))$ .

**Definizione 6.2 Race condition** Una condizione secondo la quale il valore di un dato condiviso  $d_s$ , ottenuto come risultato dell'esecuzione delle operazioni  $a_i$  e  $a_j$  su  $d_s$  nell'ambito di processi intercomunicanti, può essere differente sia da  $f_i(f_j(d_s))$  che da  $f_j(f_i(d_s))$

L'esempio successivo illustra una race condition in un'applicazione di prenotazione aerea e le sue conseguenze.

### Esempio 6.1 - Race condition in un'applicazione di prenotazione aerea

La colonna di sinistra nella metà superiore della Figura 6.2 mostra il codice eseguito dai processi utilizzati in un'applicazione di prenotazione aerea. I processi usano lo stesso codice, pertanto  $a_i$  e  $a_j$  le operazioni eseguite dai processi  $P_i$  e  $P_j$  sono identiche. Ognuna di queste operazioni esamina il valore di *nextseatno* e lo incrementa di una unità se è disponibile un posto. La colonna di destra della Figura 6.2 mostra le istruzioni macchina corrispondenti al codice. L'istruzione  $S_3$  corrisponde a tre istruzioni  $S_{3.1}$ ,  $S_{3.2}$  e  $S_{3.3}$  che implementano la sequenza di istruzioni load-add-store per l'aggiornamento del valore di *nextseatno*.

| Codice dei processi |  | Istruzioni macchina corrispondenti |  |
|---------------------|--|------------------------------------|--|
| $S_1$               | if $nextseatno \leq capacity$          | $S_{1.1}$                          | Carica <i>nextseatno</i> in $reg_k$          |
|                     | then                                   | $S_{1.2}$                          | If $reg_k > capacity$ goto $S_{4.1}$         |
| $S_2$               | $allotedno := nextseatno;$             | $S_{2.1}$                          | Assegna <i>nextseatno</i> a <i>allotedno</i> |
| $S_3$               | $nextseatno := nextseatno + 1;$        | $S_{3.1}$                          | Carica <i>nextseatno</i> in $reg_j$          |
|                     |  | $S_{3.2}$                          | Aggiungi 1 a $reg_j$                         |
|                     |  | $S_{3.3}$                          | Immagazzina $reg_j$ in <i>nextseatno</i>     |
|                     |  | $S_{3.4}$                          | Go to $S_{5.1}$                              |
|                     | else                                   |                                    |  |
| $S_4$               | Stampa "non ci sono posti disponibili" | $S_{4.1}$                          | Stampa "non . . ."                           |
| $S_5$               | ...                                    | $S_{5.1}$                          | ...  |

#### Alcuni casi di esecuzione



**Figura 6.2** Condivisione dei dati da parte dei processi dell'applicazione di prenotazione.

La metà inferiore della [Figura 6.2](#) è un diagramma dei tempi impiegati dalle applicazioni. Mostra tre possibili sequenze secondo cui i processi  $P_i$  e  $P_j$  potrebbero eseguire le loro istruzioni quando  $nextseatno = 200$  e  $capacity = 200$ . Nel caso 1, il processo  $P_i$  esegue l'istruzione **if** che confronta i valori di  $nextseatno$  con  $capacity$  e procede a eseguire le istruzioni  $S_{2.1}, S_{3.1}, S_{3.2}$  e  $S_{3.3}$  che allocano un posto e incrementano  $nextseatno$ . Quando un processo  $P_j$  esegue l'istruzione **if**, verifica che non ci sono posti disponibili per cui non alloca un posto.

Nel caso 2, il processo  $P_i$  esegue l'istruzione **if** e verifica che può essere allocato un posto. Tuttavia, viene prelazionato prima che possa eseguire l'istruzione  $S_{2.1}$ . Il processo  $P_i$  ora esegue l'istruzione **if** e verifica che è disponibile un posto. Alloca un posto eseguendo le istruzioni  $S_{2.1}, S_{3.1}, S_{3.2}$  e  $S_{3.3}$  ed esce.  $nextseatno$  vale ora 201. Quando il processo  $P_i$  viene ripristinato, procede a eseguire l'istruzione  $S_{2.1}$ , che alloca un posto. In questo modo, i posti sono allocati a entrambe le richieste. Questa è una race condition poiché quando  $nextseatno = 200$ , solo un posto dovrebbe essere allocato.

Nel caso 3, il processo  $P_i$  viene prelazionato dopo che ha caricato 200 in  $reg_j$  mediante l'istruzione  $S_{3.1}$ . A questo punto, sia  $P_i$  che  $P_j$  allocano un posto ognuno, realizzando una race condition.

Un programma che contiene una race condition può produrre risultati corretti o meno in base all'ordine in cui vengono eseguite le istruzioni dei suoi processi. Questa caratteristica complica sia la fase di test che di debug dei programmi concorrenti, per cui le race condition dovrebbero essere evitate.

### **Sincronizzazione per l'accesso ai dati**

Le race condition possono essere evitate se assicuriamo che le operazioni  $a_i$  e  $a_j$  della Definizione 6.2 non vengono eseguite in maniera concorrente – ovvero, solo una delle operazioni può avere accesso ai dati condivisi  $d_s$  a ogni istante. Questo requisito è denominato *mutua esclusione*. Quando viene assicurata la mutua esclusione, possiamo essere sicuri che il risultato dell'esecuzione delle operazioni  $a_i$  e  $a_j$  sarà o  $f_i(f_j(d_s))$  o  $f_j(f_i(d_s))$ . La *sincronizzazione per l'accesso ai dati* è il coordinamento dei processi per implementare la mutua esclusione relativa ai dati condivisi. Una tecnica di sincronizzazione per l'accesso ai dati è pertanto utilizzata per ritardare un processo che vuole accedere a  $d_s$  se un altro processo sta accedendo a  $d_s$  e per ripristinare la sua esecuzione quando l'altro processo termina l'accesso a  $d_s$ .

Per prevenire le race condition, prima si verifica se l'esecuzione dei processi in un'applicazione causa una race condition. In particolare va determinato per ogni coppia di processi  $P_i$  e  $P_j$  se vengono soddisfatte le seguenti condizioni di Bernstein:

$$\begin{aligned} (read\_set_i \cap write\_set_j) &\neq \emptyset \\ (write\_set_i \cap read\_set_j) &\neq \emptyset \\ (write\_set_i \cap write\_set_j) &\neq \emptyset \end{aligned}$$

Quando si hanno più processi viene adottato il seguente metodo:

- determinare se una coppia di processi  $P_i$  e  $P_j$  richiede sincronizzazione nell'accesso ai dati;
- assicurare che i processi accedano dati condivisi in modo mutuamente esclusivo.

A questo scopo si utilizza la seguente notazione:

$update\_set_i$       insieme di dati aggiornati dal processo  $P_i$ , ovvero l'insieme di dati i cui valori sono letti, modificati e riscritti dal processo  $P_i$

L'esecuzione di una coppia di processi  $P_i$  e  $P_j$  causa una race condition se  $update\_set_i \cap update\_set_j \neq \emptyset$  ovvero, se alcune variabili sono aggiornate sia da  $P_i$  che da  $P_j$ .

L'esecuzione dei processi  $P_i$  e  $P_j$  nell'applicazione per la prenotazione aerea dell'Esempio 6.1 causa una race condition poiché  $update\_set_i = update\_set_j = \{nextseatno\}$ . Una volta

che si conosce il dato il cui aggiornamento causa una race condition, è possibile utilizzare tecniche di sincronizzazione per l'accesso ai dati per assicurare che l'accesso a questi dati avvenga in maniera mutuamente esclusiva. Il prossimo paragrafo discute una base concettuale per l'accesso sincronizzato ai dati.

### 6.3 Sezioni critiche

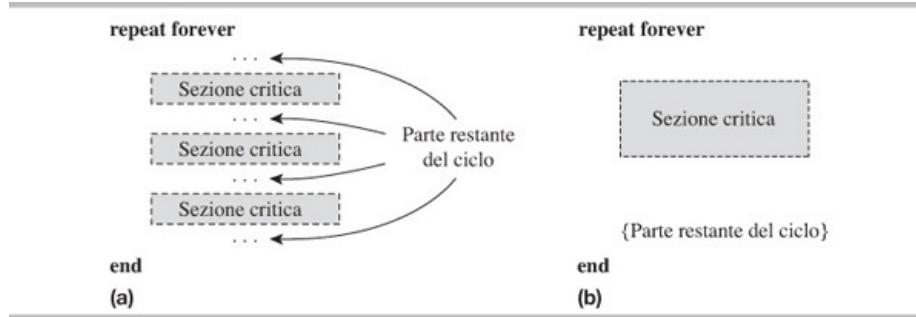
La mutua esclusione tra le azioni di processi concorrenti viene implementata usando le *sezioni critiche* del codice. Una sezione critica è generalmente conosciuta con il suo acronimo CS (Critical Section).

**Definizione 6.3 Sezione critica** Una *sezione critica* per un dato  $d_s$  è una sezione del codice progettata in modo tale che non possa essere eseguita concorrentemente sia con se stessa sia con altre sezioni critiche per  $d_s$ .

Se un processo  $P_i$  sta eseguendo una sezione critica per  $d_s$ , un altro processo che vuole eseguire una sezione critica per  $d_s$  dovrà attendere finché  $P_i$  termini l'esecuzione della sua sezione critica. In questo modo, una sezione critica per un  $d_s$  è una regione di mutua esclusione rispetto agli accessi a  $d_s$ .

La sezione critica in un segmento di codice verrà evidenziata da un rettangolo tratteggiato. È importante notare che i processi possono condividere una singola copia del segmento di codice che contiene una sezione critica, nel qual caso nell'applicazione esiste solo una sezione critica per  $d_s$ . In tutti gli altri casi, nell'applicazione possono esistere molte sezioni critiche per  $d_s$ . La Definizione 6.3 copre entrambe le situazioni. Per un processo che sta eseguendo una sezione critica si indica che esso si trova "in una sezione critica". Inoltre vengono usati i termini "entrare in una sezione critica" e "uscire da una sezione critica" per le situazioni che descrivono quando un processo avvia e termina un'esecuzione di una sezione critica.

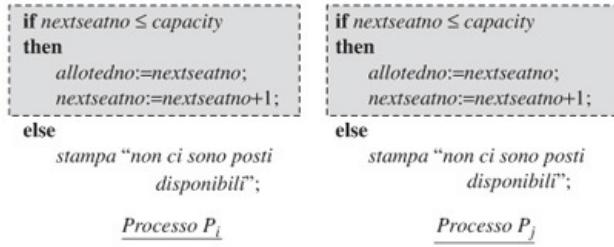
La [Figura 6.3\(a\)](#) mostra il codice di un processo che contiene diverse sezioni critiche. Il processo ha un andamento ciclico dovuto all'istruzione **repeat forever**. A ogni iterazione, entra in una sezione critica quando ha bisogno di accedere a dati condivisi. Altre volte, esegue altre parti di codice, che insieme costituiscono il "resto del ciclo". Per semplicità, quando possibile, utilizziamo la forma semplice del processo mostrata in [Figura 6.3\(b\)](#) per rappresentare un processo. L'esempio seguente illustra l'uso di una sezione critica per evitare le race condition.



**Figura 6.3** (a) Un processo con molte sezioni critiche; (b) un modo più semplice di rappresentare questo processo.

#### Esempio 6.2 - Prevenire una race condition mediante una sezione critica

La [Figura 6.4](#) mostra l'uso delle sezioni critiche nel sistema di prenotazione aerea di [Figura 6.2](#). Ogni processo contiene una sezione critica alla quale accede ed aggiorna la variabile condivisa *nextseatno*. Siano  $f_i(nextseatno)$  e  $f_j(nextseatno)$  rispettivamente le operazioni effettuate nelle sezioni critiche di  $P_i$  e  $P_j$ . Se  $P_i$  e  $P_j$  tentano di eseguire le loro sezioni critiche in maniera concorrente, uno dei due sarà ritardato. Dunque, il valore risultante di *nextseatno* sarà o  $f_i(f_j(nextseatno))$  o  $f_i(f_j(nextseatno))$ . Dalla Definizione 6.2, non si verifica una race condition.



**Figura 6.4** Uso delle sezioni critiche in un sistema di prenotazione area.

L'uso delle sezioni critiche causa ritardi nell'esecuzione dei processi; pertanto entrambi i processi ed il kernel devono cooperare per ridurre questo ritardo. Un processo non deve essere eseguito troppo a lungo all'interno di una sezione critica e non deve effettuare chiamate di sistema che lo facciano passare nello stato *blocked*. Il kernel non deve prelazionare un processo che sia impegnato nell'esecuzione di una sezione critica. Questa condizione richiede che il kernel venga sempre a conoscenza di quando un processo è in una sezione critica e non può esser soddisfatta se i processi implementano le sezioni critiche da soli, ovvero senza il supporto del kernel. Ciò nonostante, in questo capitolo dovremmo assumere che un processo trascorre solo un tempo limitato all'interno di una sezione critica.

### 6.3.1 Proprietà dell'implementazione di una sezione critica

Quando diversi processi vogliono usare le sezioni critiche per il dato  $d_s$ , l'implementazione di una sezione critica deve assicurare l'accesso alla sezione critica per tutti i processi in base alle nozioni di correttezza ed egualanza. La [Tabella 6.1](#) riassume le tre proprietà essenziali che un'implementazione di una sezione critica deve possedere per soddisfare questi requisiti. La proprietà di *mutua esclusione* garantisce che due o più processi non si troveranno simultaneamente nella sezione critica per  $d_s$ , che è il punto cruciale della [Definizione 6.3](#), e ciò assicura la correttezza dell'implementazione. La seconda e la terza proprietà della [Tabella 6.1](#) garantiscono insieme che nessun processo che vuole entrare in una sezione critica sia ritardato indefinitamente; ovvero che non si verificherà *starvation*. Discuteremo questo aspetto in seguito.

| Proprietà        | Descrizione   |
|------------------|---|
| Mutua esclusione | In ogni istante, al più un processo può eseguire una CS per un dato $d_s$ .   |
| Progresso        | Quando nessun processo sta eseguendo una CS per un dato $d_s$ , sarà concesso l'accesso a uno dei processi che vogliono entrare in una CS per $d_s$ .   |
| Attesa limitata  | Dopo che un processo $P_i$ ha manifestato il suo interesse a entrare in una CS per $d_s$ , il numero di volte che gli altri processi possono ottenere l'accesso a una CS per $d_s$ prima di $P_i$ è limitato da un intero finito. |

**Tabella 6.1** Proprietà essenziali dell'implementazione di una CS.

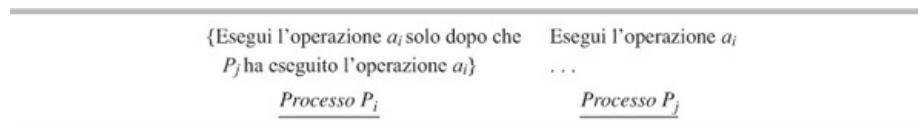
La proprietà del *progresso* assicura che se alcuni processi sono interessati a entrare nelle sezioni critiche per un dato  $d_s$ , uno di loro otterrà l'accesso se nessun processo è attualmente all'interno di una sezione critica per  $d_s$ ; in modo equivalente, questo vuol dire che l'uso di una CS non può essere "riservato" per un processo che attualmente non è interessato a entrare in una sezione critica. Tuttavia, questa proprietà da sola non può prevenire la *starvation* poiché un processo potrebbe non ottenere mai l'accesso a una CS se l'implementazione della sezione critica favorisce sempre non accada limitando il numero di volte che gli altri processi possono ottenere l'acaltri processi per entrare nella CS. La proprietà di *attesa limitata* assicura che questo cessa alla sezione critica prima di un processo  $P_i$  che ne abbia fatto richiesta. In questo modo, le proprietà di progresso e attesa limitata assicurano che ogni processo che richiede l'accesso lo otterrà in un tempo finito; tuttavia, queste proprietà non garantiscono un limite specifico al ritardo

nell'accesso alla CS.

## 6.4 Controllo della sincronizzazione e operazioni indivisibili

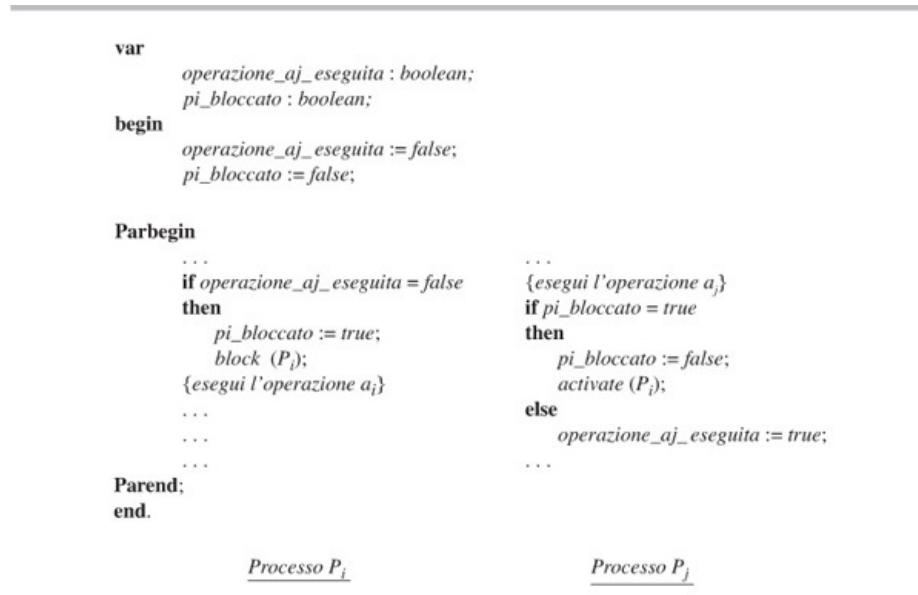
I processi intercomunicanti hanno bisogno di coordinare la loro esecuzione tra loro, in modo da eseguire le loro azioni nell'ordine desiderato. Questo requisito è soddisfatto mediante il *controllo della sincronizzazione*.

La [Figura 6.5](#) mostra uno pseudocodice per i processi  $P_i$  e  $P_j$ , in cui il processo  $P_i$  intende eseguire un'operazione  $a_i$  solo dopo che il processo  $P_j$  abbia effettuato un'operazione  $a_j$ . L'utilizzo dei *segnali* è una tecnica generale di controllo della sincronizzazione e può essere utilizzata per soddisfare i requisiti della [Figura 6.5](#) come segue: quando un processo  $P_i$  raggiunge il punto in cui vuole eseguire un'operazione  $a_i$ , controlla se il processo  $P_j$  ha effettuato l'operazione  $a_j$ . In caso affermativo,  $P_i$  effettua l'operazione  $a_i$  immediatamente; in caso contrario, si blocca in attesa che il processo  $P_j$  effettui l'operazione  $a_j$ . Dopo aver eseguito l'operazione  $a_j$ , il processo  $P_j$  controlla se  $P_i$  è in attesa. In caso affermativo, segnala al processo  $P_i$  di riprendere l'esecuzione.



**Figura 6.5** Le operazioni indivisibili  $check\_a_j$  e  $post\_a_j$  per la segnalazione.

La [Figura 6.6](#) mostra un tentativo maldestro di utilizzare i segnali. Il dato utilizzato per la sincronizzazione consiste di due variabili booleane: `operation_aj_performed` è una flag che indica se il processo  $P_j$  ha effettuato l'operazione  $a_j$  e `pi_blocked` è una flag che indica se il processo  $P_i$  è bloccato in attesa dell'esecuzione della operazione  $a_j$  da parte del processo  $P_j$ . Entrambe queste flag sono inizializzate a `false`. Il codice effettua delle chiamate di sistema per bloccare e attivare i processi al fine di ottenere il controllo della sincronizzazione desiderato.



**Figura 6.6** Un tentativo maldestro di utilizzare i segnali mediante variabili booleane.

Prima di effettuare l'operazione  $a_i$ , il processo  $P_i$  legge il valore della variabile `operazione_aj_eseguita` per controllare se il processo  $P_j$  ha già eseguito l'operazione  $a_j$ . In caso affermativo, prosegue con l'esecuzione dell'operazione  $a_i$ ; in caso contrario, imposta `pi_blocked` a `true` ed effettua una system call per bloccarsi. Il processo  $P_j$  effettua l'operazione  $a_j$  e verifica se il processo  $P_i$  si è già bloccato per attendere il

completamento dell'operazione  $a_j$ . In caso affermativo, effettua una system call per attivare  $P_i$ ; in caso contrario, imposta *operazione\_aj\_eseguita* a *true* in modo tale che il processo  $P_i$  venga a conoscenza dell'esecuzione dell'operazione  $a_j$ .

Tuttavia, questo uso maldestro dei segnali non funziona poiché il processo  $P_i$  può, in alcune situazioni, incorrere in un blocco indefinito. La [Tabella 6.2](#) mostra questa situazione. Il processo  $P_i$  controlla il valore di *operazione\_aj\_eseguita* e verifica che l'operazione  $a_j$  non è stata effettuata. Al tempo  $t_2$ , è pronto a impostare la variabile *pi\_bloccato* a *true*, ma allo stesso tempo viene prelazionato. Il processo  $P_j$  viene schedulato. Effettua l'operazione  $a_j$ , e controlla se il processo  $P_i$  è bloccato. Tuttavia, *pi\_bloccato* è *false*, per cui  $P_j$  semplicemente imposta *operazione\_aj\_eseguita* a *true* e continua la sua esecuzione.  $P_i$  viene schedulato al tempo  $t_{20}$ . Imposta *pi\_bloccato* a *true* ed effettua una system call per bloccarsi. Il processo  $P_i$  rimarrà bloccato per sempre!

| Tempo    | Azioni del processo $P_i$                      | Azioni del processo $P_j$                  |
|----------|--|--|
| $t_1$    | <code>if operazione_aj_eseguita = false</code> |  |
| $t_2$    |  | {esegui operazione $a_j$ }                 |
| $t_3$    |  | <code>if pi_bloccato = true</code>         |
| $t_4$    |  | <code>operazione_aj_eseguita = true</code> |
| $\vdots$ |  |  |
| $t_{20}$ | <code>pi_bloccato = true;</code>               |  |
| $t_{21}$ | <code>blocca (P_i);</code>                     |  |

**Tabella 6.2** Race condition nella sincronizzazione dei processi.

Nella notazione del Paragrafo 1.2, si considera che le istruzioni **if** nei processi  $P_i$  e  $P_j$  rappresentino le operazioni  $f_i$  e  $f_j$  nello stato del sistema. Il risultato della loro esecuzione dovrebbe essere uno dei seguenti: il processo  $P_i$  si blocca, viene attivato da  $P_j$  ed effettua l'operazione  $a_i$ ; oppure il processo  $P_i$  verifica che  $P_j$  ha già effettuato  $a_j$  e prosegue con l'esecuzione di  $a_i$ . Tuttavia, nell'esecuzione mostrata nella [Tabella 6.2](#), il processo  $P_i$  si blocca e non è mai attivato. Dalla Definizione 6.4, questa è una race condition.

La race condition ha due cause principali: il processo  $P_i$  può essere prelazionato dopo aver verificato che *operazione\_aj\_eseguita* = *false* ma prima di impostare *pi\_bloccato* a *true* oppure il processo  $P_j$  può essere prelazionato dopo aver verificato che *pi\_bloccato* = *false* ma prima di impostare *operazione\_aj\_eseguita* a *true*. La race condition può essere evitata se possiamo assicurare che i processi  $P_i$  e  $P_j$  non vengano prelazionati prima che impostino le rispettive flag a *true*. Un'*operazione indivisibile* (detta anche *operazione atomica*) è il mezzo che assicura l'esecuzione di una sequenza di azioni senza essere prelazionati.

**Definizione 6.4 Operazione indivisibile** Un'operazione su un insieme di dati che non può essere eseguita concorrentemente né con se stessa né con ogni altra operazione su un qualsiasi dato incluso nell'insieme.

Dal momento che un'operazione indivisibile non può essere eseguita in maniera concorrente con ogni altra operazione che coinvolge gli stessi dati, essa deve essere completata prima che ogni altro processo abbia accesso ai dati. La situazione mostrata nella [Tabella 6.2](#) non si verificherebbe se le istruzioni **if** nella [Figura 6.6](#) fossero implementate come operazioni indivisibili sui dati *operazione\_aj\_eseguita* e *pi-bloccato*, poiché se il processo  $P_i$  trovasse *operazione\_aj\_eseguita* = *false*, sarebbe in grado di impostare *pi\_bloccato* a *true* senza essere prelazionato e se il processo  $P_j$  trovasse *pi\_bloccato* a *false*, sarebbe in grado di impostare *operazione\_aj\_eseguita* a *true* senza essere prelazionato. Di conseguenza, definiamo due operazioni indivisibili *check\_aj* e *post\_aj* per effettuare le istruzioni **if**, rispettivamente dei processi  $P_i$  e  $P_j$  e sostituiamo le istruzioni **if** con l'invocazione di queste operazioni indivisibili. La [Figura 6.7](#) mostra i dettagli delle operazioni indivisibili *check\_aj* e *post\_aj*.

---

```

procedure check_aj
begin
  if operazione_aj_eseguita=false
  then
    pi_bloccato:=true;
    block (Pi)
  end;
procedure post_aj
begin
  if pi_bloccato=true
  then
    pi_bloccato:=false;
    activate(Pj)
  else
    operazione_aj_eseguita:=true;
  end;

```

---

**Figura 6.7** Le operazioni indivisibili *check\_aj* e *post\_aj* per la segnalazione.

Quando *operazione\_aj\_eseguita* è *false*, l'operazione indivisibile *check\_aj* viene considerata completata dopo che il processo  $P_i$  si è bloccato; questo consente al processo  $P_j$  di effettuare l'operazione *post\_aj*.

Un'operazione indivisibile sull'insieme di dati  $\{d_s\}$  è come una sezione critica su  $\{d_s\}$ . Tuttavia, facciamo distinzione tra i due termini perché una sezione critica deve essere implementata esplicitamente in un programma, mentre l'hardware o il software di un computer possono fornire alcune operazioni indivisibili tra le proprie operazioni primitive.

## 6.5 Approcci alla sincronizzazione

In questo paragrafo discuteremo come possono essere implementate le sezioni critiche e le operazioni indivisibili richieste per la sincronizzazione dei processi.

### 6.5.1 Ciclare o bloccare

Una sezione critica per  $\{d_s\}$  ed un'operazione indivisibile di segnalazione su  $\{d_s\}$  hanno lo stesso requisito di base: i processi non devono essere in grado di eseguire alcuna sequenza di istruzioni in maniera concorrente o parallela. Dunque entrambe potrebbero essere implementate mediante la mutua esclusione come segue:

**while** (qualche processo è nella sezione critica su  $\{d_s\}$  o sta eseguendo un'operazione indivisibile utilizzando  $\{d_s\}$ ) { non fare niente }

Sezione critica oppure operazione indivisibile tramite  $\{d_s\}$

Nel ciclo while, il processo controlla se qualche processo è in una sezione critica per lo stesso dato, o sta eseguendo un'operazione indivisibile utilizzando lo stesso dato. In caso affermativo, continua a ciclare finché l'altro processo termina. Questa situazione prende il nome di *busy wait* poiché mantiene la CPU occupata nell'esecuzione di un processo anche se il processo non fa nulla! L'attesa attiva termina solo quando il processo verifica che nessun altro processo è in una sezione critica o sta eseguendo un'operazione indivisibile.

Un'attesa attiva in un processo ha molte conseguenze negative. Un'implementazione delle sezioni critiche che utilizza le attese attive non può soddisfare la proprietà dell'attesa limitata poiché quando molti processi sono in attesa attiva per una CS, l'implementazione non può controllare quale processo otterrebbe l'accesso alla CS quando il processo, attualmente nella CS, ne esce. In un SO time-sharing, un processo che entra in attesa attiva per ottenere l'accesso a una CS utilizzerebbe tutta la sua time slice senza ottenere accesso alla CS, la qual cosa farebbe degradare le prestazioni del sistema.

In un SO che utilizza lo scheduling a priorità, un'attesa attiva può portare a una situazione in cui i processi si attendono l'un l'altro indefinitamente. Si consideri la seguente situazione: un processo ad alta priorità  $P_i$  è bloccato su un'operazione di I/O e un processo a bassa priorità  $P_j$  entra in una sezione critica per un dato  $d_s$ . Quando viene completata l'operazione di  $P_i$ ,  $P_j$  viene prelazionato e  $P_i$  viene schedulato. Se  $P_i$  prova ora a entrare in una sezione critica per  $d_s$  utilizzando il ciclo **while** descritto in precedenza, affronterebbe una attesa attiva. Tale attesa attiva nega la CPU a  $P_j$ , per cui non è in grado di completare la sua esecuzione ed esce dalla sezione critica, ma ciò evita che  $P_i$  entri nella sua sezione critica. I processi  $P_i$  e  $P_j$  ora attendono l'un l'altro indefinitamente. Poiché un processo ad alta priorità attende un processo a priorità più bassa, questa situazione viene chiamata *inversione della priorità*. Il problema dell'inversione della priorità tipicamente viene risolto mediante il *priority inheritance protocol*, in cui un processo a bassa priorità che detiene una risorsa acquisisce temporaneamente la priorità del processo a più alta priorità che richiede quella risorsa. Nel nostro esempio, il processo  $P_j$  otterrebbe temporaneamente la priorità del processo  $P_i$  che gli consentirebbe di essere schedulato e di uscire dalla sua sezione critica. Tuttavia, l'uso del priority inheritance protocol non è pratico in queste situazioni poiché richiede che il kernel conosca i dettagli dell'esecuzione dei processi.

Per evitare le attese attive, un processo in attesa di entrare in una sezione critica deve andare nello stato *blocked*. Il suo stato dovrebbe essere impostato a *ready* solo quando gli è consentito di entrare nella CS. Questo approccio può essere realizzato facendo sì che il processo che intende entrare in CS esegua la seguente procedura:

**if** (qualche processo è in una sezione critica su  $\{d_s\}$  o sta eseguendo un'operazione indivisibile usando  $\{d_s\}$ ) *effettua una chiamata di sistema per bloccarsi*;

Sezione critica oppure operazione indivisibile tramite  $\{d_s\}$

Secondo tale approccio, il kernel riattiva il processo bloccato solo quando nessun altro processo è in esecuzione in una sezione critica su  $\{d_s\}$  o sta eseguendo un'operazione indivisibile utilizzando  $\{d_s\}$ .

Quando una sezione critica o un'operazione indivisibile è realizzata mediante una delle precedenti procedure, un processo che desidera entrare in una CS deve controllare se qualche altro processo si trova nella CS e di conseguenza decidere se ciclare (o bloccarsi). Questa azione coinvolge l'esecuzione di poche istruzioni in maniera mutuamente esclusiva per evitare race condition (Paragrafo 6.4). Come è possibile evitarla? Ciò può essere fatto in due modi. Nel primo approccio, detto *l'approccio algoritmico*, viene adottata una complessa organizzazione di controlli per evitare race condition. Discuteremo le caratteristiche di questo approccio e i suoi svantaggi nel Paragrafo 1.8. Il secondo approccio utilizza alcune caratteristiche dell'hardware per semplificare questo controllo. Discuteremo questo approccio nel prossimo paragrafo.

### 6.5.2 Supporto hardware per la sincronizzazione dei processi

La sincronizzazione dei processi coinvolge l'esecuzione di alcune sequenze di istruzioni in maniera mutuamente esclusiva. Su un sistema monoprocessore, questo può essere ottenuto disabilitando gli interrupt mentre un processo esegue tale sequenza di istruzioni, in modo che non possa essere prelazionato. Tuttavia, questo approccio utilizza il meccanismo dell'overhead delle chiamate di sistema per disabilitare gli interrupt ed abilitarli nuovamente ed inoltre ritarda l'elaborazione degli interrupt, che può portare a conseguenze indesiderabili per le prestazioni del sistema o per il servizio per l'utente. Inoltre, non è applicabile ai sistemi multiprocessore. Per queste ragioni, i sistemi operativi implementano le sezioni critiche e le operazioni indivisibili attraverso *istruzioni indivisibili* fornite dai computer, insieme a variabili condivise chiamate *variabili di lock*. In questo paragrafo, useremo le illustrazioni dell'approccio basato sui cicli per la sincronizzazione dei processi; tuttavia, le tecniche qui discusse sono egualmente applicabili all'approccio basato sul blocco per la sincronizzazione dei processi. Va notato che le istruzioni indivisibili semplicemente aiutano l'implementazione delle sezioni critiche; le proprietà dell'implementazione della CS riassunte nella [Tabella 6.1](#) devono essere assicurate separatamente consentendo ai processi di accedere alle CS nel modo

appropriato (Problema 6.12).

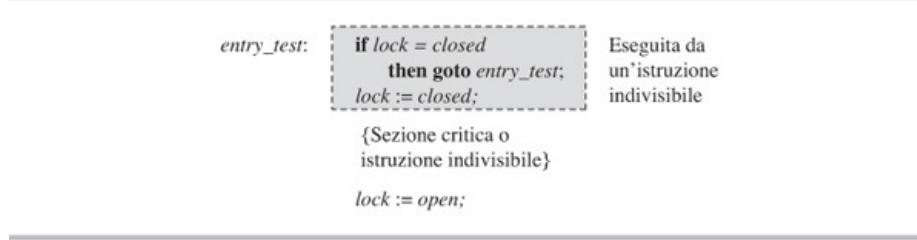
### Istruzioni indivisibili

Sin dalla metà degli anni '60, sono state previste nei computer caratteristiche speciali dell'hardware per prevenire le race condition durante l'accesso alle locazioni di memoria contenenti dati condivisi. L'assunzione di base è che tutti gli accessi alla locazione di memoria effettuati da un'istruzione dovrebbero essere implementati senza permettere a un'altra CPU di accedere alla stessa locazione. Due tecniche note usate a tale scopo consistono nel bloccare il bus di memoria durante un'istruzione (per esempio, nei processori Intel 80x86) e nel fornire istruzioni speciali che effettuano alcune operazioni specifiche sulle locazioni di memoria in modo da evitare race condition (per esempio, nei processori IBM/370 e M68000). Utilizzeremo il termine *istruzione indivisibile* come termine generico per tutte queste istruzioni.

### Uso della variabile di lock

Una *variabile di lock* è una variabile a due stati utilizzata per colmare il gap semantico (Definizione 4.1) tra le sezioni critiche o le operazioni indivisibili da una parte e le istruzioni indivisibili messe a disposizione dal computer dall'altra. Per implementare le sezioni critiche per un dato  $d_s$ , un'applicazione associa una variabile di lock a  $d_s$ . La variabile di lock può assumere solo due possibili stati: aperto e chiuso. Quando un processo vuole eseguire una sezione critica per  $d_s$ , legge il contenuto della variabile di lock e, se il lock è aperto, lo imposta a chiuso, esegue la sezione critica e lo imposta nuovamente ad aperto al termine della sezione critica. Per evitare race condition nell'impostazione della variabile di lock, viene utilizzata un'operazione indivisibile per la lettura e la chiusura. Le variabili di lock, in maniera simile, aiutano l'implementazione delle operazioni indivisibili.

La [Figura 6.8](#) illustra come una sezione critica o un'operazione indivisibile possono essere implementate utilizzando un'istruzione indivisibile ed una variabile di lock. L'istruzione indivisibile effettua le azioni indicate nel rettangolo tratteggiato: se il lock è chiuso, ricomincia l'esecuzione dell'istruzione; in caso contrario, chiude il lock. Di seguito, illustriamo l'uso di due istruzioni indivisibili, chiamate istruzioni *test-and-set* e *swap*, per implementare le sezioni critiche e le operazioni indivisibili.



**Figura 6.8** Implementazione di una sezione critica o di un'operazione indivisibile mediante l'uso di una variabile di lock.

### Istruzione Test-and-Set (TS)

Questa istruzione indivisibile effettua due azioni. "Esamina" il valore di un byte di memoria ed imposta il campo *condition code* (ovvero il campo *flag*) del PSW per indicare se il valore era zero o diverso da zero. Inoltre imposta tutti i bit nel byte a uno. Nessun'altra CPU può accedere al byte finché entrambe le azioni non sono completate. Questa istruzione può essere usata per implementare le istruzioni racchiuse nel rettangolo tratteggiato di [Figura 6.8](#).

La [Figura 6.9](#) è un segmento di un programma in linguaggio assembly per un IBM/370 che implementa una sezione critica o un'operazione indivisibile. LOCK è una variabile di lock utilizzata con la convenzione che un valore diverso da zero implica che il lock è chiuso e uno zero implica che il lock è aperto. La prima linea del programma assembly dichiara LOCK e la inizializza a zero. L'istruzione TS imposta il codice di condizione in base al valore di LOCK e successivamente imposta il valore di LOCK a *chiuso*. In questo modo, il codice di condizione indica se il lock era chiuso prima che fosse eseguita l'istruzione TS. L'istruzione di salto BC 7, ENTRY\_TEST controlla il codice di condizione e ripete l'istruzione TS se il lock era chiuso. In questo modo un processo che trova il lock

chiuso esegue il ciclo con una attesa attiva finché il lock non viene aperto. L'istruzione MVI imposta a zero tutti i bit di LOCK, ovvero apre il lock. Questa azione permetterebbe solo a uno dei processi impegnati nel loop ENTRY\_TEST di procedere con l'esecuzione.

---

|            |                  |   |
|------------|------------------|---|
| LOCK       | DC X'00'         | Lock è inizializzato a aperto               |
| ENTRY_TEST | TS LOCK          | Test-and-set lock                           |
|            | BC 7, ENTRY_TEST | Cicla se lock è chiuso                      |
|            | ...              | {Sezione critica o operazione indivisibile} |
|            | MVI LOCK, X'00'  | Apre il lock (spostando gli 0)              |

---

**Figura 6.9** Implementazione di una sezione critica o di un'operazione indivisibile utilizzando l'istruzione test-and-set.

### Istruzione swap

L'istruzione SWAP scambia il contenuto di due locazioni di memoria. È un'istruzione indivisibile; nessun'altra CPU può accedere a nessuna delle locazioni durante lo swapping. La [Figura 6.10](#) mostra come una sezione critica o un'operazione indivisibile può essere implementata utilizzando l'istruzione swap (per convenienza, usiamo le stesse convenzioni per la codifica usate per l'istruzione TS). La locazione temporanea TEMP viene inizializzata a un valore diverso da zero. L'istruzione SWAP scambia il suo contenuto con LOCK, chiudendo il lock. In tal modo, il vecchio valore di LOCK è ora disponibile in TEMP. Viene utilizzato per verificare se il lock era già chiuso. In caso affermativo, il processo cicla sull'istruzione SWAP finché il lock viene aperto. Il processo che esegue la sezione critica o l'operazione indivisibile apre il lock al termine dell'operazione. Questa azione consente a un processo di proseguire oltre l'istruzione BC ed entrare nella sezione critica o nell'operazione indivisibile.

---

|            |                  |   |
|------------|------------------|---|
| TEMP       | DS 1             | Riserva un byte per TEMP                    |
| LOCK       | DC X'00'         | Lock è inizializzato a aperto               |
|            | MVI TEMP, X'FF'  | X'FF' è utilizzato per chiudere il lock     |
| ENTRY_TEST | SWAP LOCK, TEMP  |   |
|            | COMP TEMP, X'00' | Esamina il vecchio valore di lock           |
|            | BC 7, ENTRY_TEST | Cicla se lock è chiuso                      |
|            | ...              | {Sezione critica o operazione indivisibile} |
|            | MVI LOCK, X'00'  | Apre il lock                                |

---

**Figura 6.10** Implementazione di una sezione critica o di un'operazione indivisibile utilizzando l'istruzione swap.

Molti computer forniscono un'istruzione *compare-and-swap*. Questa istruzione ha tre operandi. Se i primi due operandi sono uguali, la funzione copia il valore del terzo operando nella locazione del secondo; in caso contrario, copia il valore del secondo operando nella locazione del primo operando. È semplice riscrivere il programma di [Figura 6.10](#) utilizzando l'istruzione Compare-and-swap `first_opd, LOCK, third_opd` dove il valore di `first_opd` e `third_opd` corrisponde ai valori aperto e chiuso del lock. In effetti, questa istruzione chiude il lock e mette il suo vecchio valore in `first_opd`.

### 6.5.3 Approcci algoritmici, primitive di sincronizzazione e costrutti di programmazione concorrente

Storicamente, l'implementazione della sincronizzazione dei processi si è mossa attraverso tre tappe fondamentali: gli approcci algoritmici, le primitive di sincronizzazione e i costrutti di programmazione concorrente. Ogni approccio ha risolto nel tempo difficoltà pratiche che si erano presentate negli approcci precedenti.

Gli *approcci algoritmici* furono ampiamente utilizzati per l'implementazione della mutua esclusione. Non utilizzavano nessuna caratteristica dell'architettura del computer,

dei linguaggi di programmazione o del kernel per ottenere la mutua esclusione; dipendevano, invece, da una complessa organizzazione di controlli per assicurare che i processi accedessero ai dati condivisi in maniera mutuamente esclusiva. Per questo motivo gli approcci algoritmici erano indipendenti dalle piattaforme hardware e software. Tuttavia, la correttezza della mutua esclusione dipendeva dalla correttezza di questi controlli ed era difficile da dimostrare a causa della complessità logica di questi controlli. Questo problema ha inibito lo sviluppo di grandi applicazioni. Dal momento che gli approcci algoritmici lavoravano indipendentemente dal kernel, non potevano adottare l'approccio bloccante per la sincronizzazione dei processi (Paragrafo 6.5.1), per cui usavano l'approccio basato sui cicli con tutti gli svantaggi derivanti.

Per superare le mancanze dell'approccio algoritmico fu sviluppato un insieme di *primitive di sincronizzazione*, ove ogni primitiva era una semplice operazione che contribuiva alla sincronizzazione dei processi ed era realizzata utilizzando le istruzioni indivisibili implementate in hardware ed il supporto del kernel per il blocco e l'attivazione dei processi. Le primitive possedevano proprietà utili per implementare sia la mutua esclusione sia le operazioni indivisibili e si sperava che queste proprietà potessero essere usate per costruire le dimostrazioni di correttezza di un programma concorrente. Tuttavia, l'esperienza ha mostrato che queste primitive potevano essere utilizzate in maniera errata, cosa che rendeva difficile la correttezza dei programmi. Molti sistemi operativi moderni mettono a disposizione le primitive dei *semafori wait* e *signal*; tuttavia, sono utilizzate solo dai programmati del sistema a causa dei problemi sopra menzionati.

In seguito furono sviluppati i *costrutti di programmazione concorrente*, che fornivano le caratteristiche di astrazione dei dati e di encapsulamento specificamente adatte per la costruzione di programmi concorrenti. Avevano una semantica ben definita che era implementata dal compilatore del linguaggio, dove i costrutti di programmazione concorrente incorporavano funzioni analoghe a quelle messe a disposizione dalle primitive di sincronizzazione, ma includevano anche caratteristiche per garantire che queste funzioni non potessero essere utilizzate in maniera errata o indiscriminata. Queste proprietà aiutavano ad assicurare la correttezza dei programmi, che rendeva possibile la realizzazione di applicazioni di grosse dimensioni. A tal fine, molti moderni linguaggi di programmazione quale Java prevedono un costrutto di programmazione concorrente chiamato *monitor*.

Discuteremo gli approcci algoritmici alla sincronizzazione dei processi nel Paragrafo 6.8, mentre i semafori e le primitive di sincronizzazione verranno affrontati nel Paragrafo 6.9. Il Paragrafo 6.11 descrive i monitor.

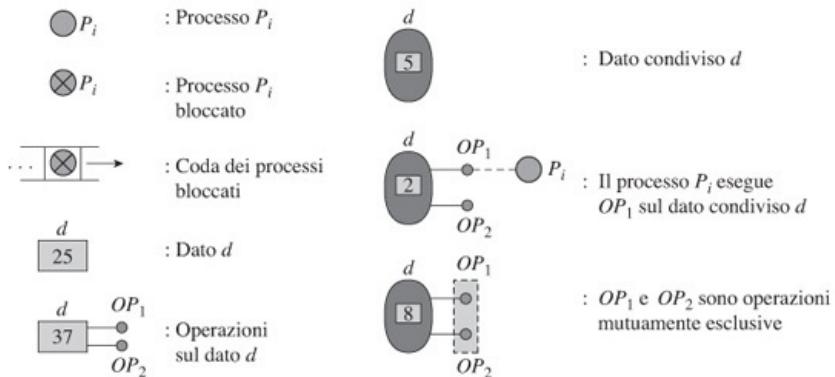
## 6.6 Struttura dei sistemi concorrenti

Un sistema concorrente si compone di tre elementi principali:

- dati condivisi;
- operazioni sui dati condivisi;
- processi intercomunicanti.

I dati condivisi includono due tipi di dati: dati dell'applicazione usati e manipolati dai processi e dati di sincronizzazione, ovvero i dati utilizzati per la sincronizzazione tra processi. Un'operazione è una specifica unità di codice, tipicamente una funzione o una procedura in un linguaggio di programmazione, che accede e manipola dati condivisi. Un'*operazione di sincronizzazione* è un'operazione sui dati di sincronizzazione.

Un'*istantanea* di un sistema concorrente è una rappresentazione del sistema a uno specifico istante di tempo che mostra le relazioni tra i dati condivisi, le operazioni e i processi a quell'istante di tempo. Si utilizzano le illustrazioni convenzionali mostrate in [Figura 6.11](#) per rappresentare un'istantanea. Un processo è rappresentato da un cerchio. Un cerchio con una croce all'interno indica un processo bloccato. Un dato, o un insieme di dati, è rappresentato da un rettangolo. I valori dei dati, se noti, sono mostrati all'interno del rettangolo.



**Figura 6.11** Illustrazioni convenzionali per le istantanee dei sistemi concorrenti.

Le operazioni sui dati sono mostrate come connettori ai dati. Una figura ovale che racchiude un dato indica che il dato è condiviso. Un linea tratteggiata connette un processo ed un'operazione sul dato se il processo è attualmente impegnato nell'esecuzione dell'operazione. Va ricordato che un rettangolo tratteggiato racchiude il codice eseguito come una sezione critica. Estendiamo questa convenzione alle operazioni sui dati. Dunque le operazioni mutuamente esclusive sui dati sono racchiuse in rettangoli tratteggiati; una coda di processi bloccati è associata a un rettangolo tratteggiato per indicare i processi in attesa di eseguire una delle operazioni.

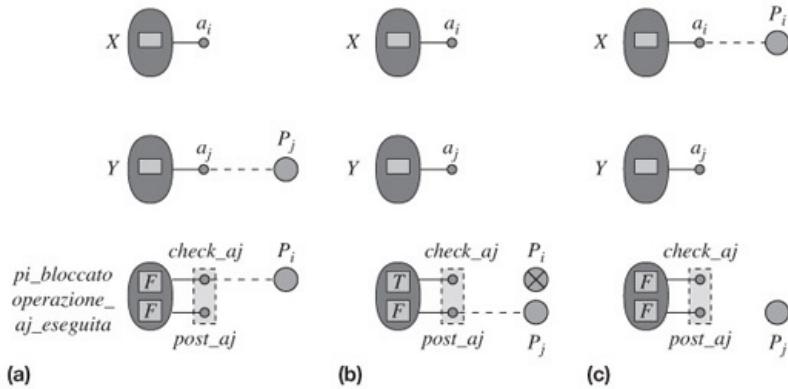
L'esecuzione di un sistema concorrente è rappresentata da una serie di istantanee.

### Esempio 6.3 - Instantanea di un sistema concorrente

Si consideri il sistema di [Figura 6.5](#), dove il processo  $P_i$  esegue l'operazione  $a_i$  solo dopo che il processo  $P_j$  ha eseguito l'azione  $a_j$ . Assumiamo che le operazioni  $a_i$  e  $a_j$  lavorino, rispettivamente, sui dati condivisi  $X$  e  $Y$ . Si consideri che il sistema sia implementato utilizzando le operazioni *check\_aj* e *post\_aj* di [Figura 6.5](#). Le componenti del sistema sono le seguenti.

|                                       |  |
|---------------------------------------|--|
| Dati condivisi                        | Le variabili booleane <i>operazione_aj_eseguita</i> e <i>pi_bloccato</i> , entrambe inizializzate a <i>false</i> ed i dati $X$ e $Y$ . |
| Operazioni sui dati dell'applicazione | Operazioni $a_i$ e $a_j$ .   |
| Operazioni di sincronizzazione        | Operazioni <i>check_aj</i> e <i>post_aj</i> .  |
| Processi                              | Processi $P_i$ e $P_j$ .   |

La [Figura 6.12](#) mostra tre istantanee del sistema.  $T$  e  $F$  indicano, rispettivamente, i valori *true* e *false*. Le operazioni *check\_aj* e *post\_aj* usano entrambe le variabili booleane *operazione\_aj\_eseguita* e *pi\_bloccato*. Queste operazioni sono indivisibili, per cui sono mutuamente esclusive. Di conseguenza, sono racchiuse in un rettangolo tratteggiato. La [Figura 6.12\(a\)](#) mostra la situazione secondo la quale il processo  $P_j$  è impegnato nell'esecuzione dell'operazione  $a_j$  e il processo  $P_i$  vuole effettuare l'operazione  $a_i$ , per cui invoca l'operazione *check\_aj*. L'operazione *check\_aj* riscontra che *operazione\_aj\_eseguita* è *false*, per cui imposta *pi\_bloccato* a *true*, blocca il processo  $P_i$  ed esce. Quando  $P_j$  termina l'esecuzione dell'operazione  $a_j$  invoca l'operazione *post\_aj* ([Figura 6.12\(b\)](#)). Questa operazione trova che *pi\_bloccato* è *true*, per cui imposta *pi\_bloccato* a *false*, attiva il processo  $P_i$  ed esce. Il processo  $P_i$  ora effettua l'operazione  $a_i$ , ([Figura 6.12\(c\)](#)).



**Figura 6.12** Istantanee del sistema dell’Esempio 6.3.

## 6.7 Problemi classici di sincronizzazione dei processi

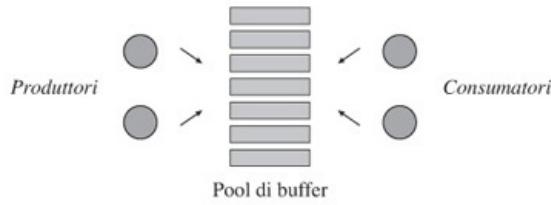
Una soluzione a un problema di sincronizzazione dei processi dovrebbe soddisfare tre criteri importanti:

- *correttezza*: l’accesso sincronizzato ai dati ed il controllo della sincronizzazione dovrebbero essere effettuati secondo i requisiti di sincronizzazione del problema;
- *massima concorrenza*: un processo dovrebbe essere in grado di funzionare liberamente fatta eccezione per i periodi in cui attende che altri processi eseguano le azioni di sincronizzazione;
- *nessuna attesa attiva*: per evitare il degrado delle prestazioni, la sincronizzazione dovrebbe essere eseguita mediante blocchi piuttosto che utilizzando l’attesa attiva (Paragrafo 6.5.1).

Come discusso nei Paragrafi 6.3 e 6.4, le sezioni critiche e l’utilizzo dei segnali rappresentano gli elementi chiave per la sincronizzazione dei processi, per cui una soluzione a un problema di sincronizzazione dovrebbe incorporare una combinazione adeguata di questi elementi. In questo paragrafo, analizzeremo alcuni problemi classici di sincronizzazione dei processi, rappresentativi dei problemi di sincronizzazione di vari campi di applicazione, e discuteremo le questioni (e gli errori comuni) relativi alla progettazione delle loro soluzioni. Nei paragrafi successivi implementeremo le soluzioni di questi problemi utilizzando varie tecniche di sincronizzazione messe a disposizione dai linguaggi di programmazione.

### 6.7.1 Produttori-consumatori con buffer limitati

Un *sistema produttori-consumatori con buffer limitati* si compone di un numero non specificato di processi *produttori* e *consumatori* ed un insieme finito di buffer (Figura 6.13). Ogni buffer è in grado di contenere un elemento di informazione – si dice che è *pieno* quando un produttore scrive un nuovo elemento al suo interno e diventa *vuoto* quando un consumatore estraie un elemento contenuto in esso; è pertanto vuoto quando il sistema produttori-consumatori inizia l’esecuzione. Un processo produttore produce un elemento di informazione alla volta e lo inserisce in un buffer vuoto. Un processo consumatore estraie l’informazione, un elemento alla volta, da un buffer pieno.



**Figura 6.13** Produttore-consumatori con buffer limitati.

Un sistema produttore-consumatore con buffer limitati è un'utile astrazione per molti problemi di sincronizzazione. Un servizio di stampa è un buon esempio nell'ambito dei SO: una coda, a dimensione fissa, di richieste di stampa rappresenta il buffer limitato, un processo che aggiunge una richiesta di stampa nella coda è un processo produttore ed un demone di stampa rappresenta un processo consumatore. L'applicazione per il log dei dati dell'Esempio 5.1 potrebbe essere considerata un'istanza del problema produttori-consumatori se il processo *pulisci* fosse omesso - il processo *copia\_campione* è il produttore poiché scrive un campione nel buffer. Il processo *record\_campione* è un consumatore poiché preleva campioni dal buffer e li scrive nel file sul disco.

Una soluzione al problema produttori-consumatori deve soddisfare le seguenti condizioni:

1. un produttore non deve sovrascrivere un buffer pieno;
2. un consumatore non deve consumare un buffer vuoto;
3. i produttori ed i consumatori devono accedere ai buffer in maniera mutuamente esclusiva.

Alcune volte viene imposta anche la seguente condizione:

4. l'informazione deve essere consumata nello stesso ordine con il quale è stata inserita nei buffer, ovvero secondo l'ordine FIFO.

La [Figura 6.14](#) mostra una soluzione al problema dei produttori-consumatori. Un produttore entra nella sezione critica e controlla se è disponibile un buffer vuoto. In caso affermativo, inserisce un elemento in questo buffer; in caso contrario, esce dalla sezione critica. Questa sequenza viene ripetuta finché non trova un buffer vuoto. La variabile booleana *prodotto* viene utilizzata per interrompere il ciclo **while** dopo che il produttore ha inserito un elemento nel buffer vuoto. In modo analogo, un consumatore controlla ripetutamente il buffer finché non ne trova uno pieno da consumare.

```

begin
Parbegin
  var prodotto : boolean;
repeat
  prodotto := false
  while prodotto = false
    if esiste un buffer vuoto
    then
      { Inserisci nel buffer }
      prodotto := true;
      { Parte restante del ciclo }
  forever;
Parend;
end.

var consumato : boolean;
repeat
  consumato := false;
  while consumato = false
    if esiste un buffer pieno
    then
      { Estrai dal buffer }
      consumato := true;
      { Parte restante del ciclo }
  forever;

```

Produttore                    Consumatore

**Figura 6.14** Una soluzione per i produttori-consumatori utilizzando le sezioni critiche.

Questa soluzione presenta due problemi, uno relativo alla scarsa concorrenza e l'altro relativo all'attesa attiva. Il pool contiene molti buffer per cui sarebbe possibile per i produttori e i consumatori accedere in maniera concorrente, rispettivamente, ai buffer vuoti e pieni. Tuttavia entrambe le azioni di produzione e di consumo avvengono in

sezioni critiche per l'intero buffer, per cui solo un processo, sia esso produttore o consumatore, può accedere al buffer a ogni istante di tempo.

L'attesa attiva è presente sia nei produttori che nei consumatori. Un produttore controlla ripetutamente se esiste un buffer vuoto ed un consumatore controlla ripetutamente se esiste un buffer pieno. Per evitare le attese attive, un processo produttore dovrebbe bloccarsi se non è disponibile un buffer vuoto. Quando un consumatore consuma un elemento del buffer, dovrebbe attivare un produttore in attesa di un buffer vuoto. Analogamente, un consumatore dovrebbe bloccarsi se non è disponibile un buffer pieno; un produttore dovrebbe attivare tale consumatore dopo aver inserito un elemento nel buffer.

Analizzando nuovamente il problema dei produttori-consumatori tenendo conto di questo aspetto, è possibile notare che sebbene vi sia una mutua esclusione tra un produttore ed un consumatore che utilizzano lo stesso buffer, in realtà si tratta di un problema di segnalazione. Dopo aver inserito un elemento nel buffer, un produttore dovrebbe segnalare la disponibilità di un elemento a un consumatore che vuole prelevare un dato da quel buffer. In modo analogo, dopo aver consumato un elemento di un buffer, un consumatore dovrebbe segnalare la disponibilità di un buffer vuoto a un produttore che vuole inserire un elemento nel buffer. Questi requisiti possono essere soddisfatti usando i metodi di segnalazione discussi nel Paragrafo 6.4.

Una soluzione migliore che utilizza questo approccio è mostrata nella [Figura 6.15](#) per un semplice sistema produttori-consumatori che si compone di un singolo produttore, un singolo consumatore e un singolo buffer. L'operazione *controllo\_vuoto* eseguita dal produttore lo blocca se il buffer è pieno, mentre l'operazione *post\_pieno* imposta *buffer\_pieno* a *true* ed attiva il consumatore se il consumatore è bloccato in attesa che il buffer si riempia. Operazioni analoghe *controllo\_pieno* e *post\_vuoto* sono definite per il processo consumatore. Le flag booleane *produttore\_bloccato* e *consumatore\_bloccato* sono utilizzate da queste operazioni per annotare a ogni istante se il processo produttore o, rispettivamente, il consumatore è bloccato. La [Figura 6.16](#) mostra i dettagli delle operazioni indivisibili. Questa soluzione dovrà essere estesa per gestire buffer multipli o più processi produttori/consumatori. Discuteremo questo aspetto nel Paragrafo 6.9.2.

---

```

var
  buffer: . . .;
  buffer_full: boolean;
  produttore_bloccato, consumatore_bloccato: boolean;
begin
  buffer_full := false;
  produttore_bloccato := false;
  consumatore_bloccato := false;
Parbegin
  repeat
    check_b_empty;
    {Inserisci nel buffer}
    post_b_full;
    {Parte restante del ciclo}
  all'infinito;
  Parend;
end.

```

Produttore
Consumatore

---

**Figura 6.15** Una soluzione migliore per il problema produttore-consumatore con singolo buffer utilizzando metodi di segnalazione.

```

procedure check_b_empty
begin
  if buffer_full = true
  then
    produttore_bloccato := true;
    block (produttore);
  end;

procedure post_b_full
begin
  buffer_full := true;
  if consumatore_bloccato = true
  then
    consumatore_bloccato := false;
    activate (consumatore);
  end;

```

Operazioni del produttore

```

procedure check_b_full
begin
  if buffer_full = false
  then
    consumatore_bloccato := true;
    block (consumatore);
  end;

procedure post_b_empty
begin
  buffer_full := false;
  if produttore_bloccato = true
  then
    produttore_bloccato := false;
    activate (produttore);
  end;

```

Operazioni del consumatore

**Figura 6.16** Operazioni indivisibili per il problema dei produttori-consumatori.

### 6.7.2 Lettori e scrittori

Si consideri un *sistema lettore-scrittori* composto da dati condivisi, un numero indefinito di processi *lettori* che possono esclusivamente leggere i dati e un numero indefinito di processi *scrittori* che modificano o aggiornano i dati. Si utilizzeranno i termini *leggere* e *scrivere* per gli accessi ai dati condivisi eseguiti, rispettivamente, dai processi lettori e scrittori. Una soluzione al problema dei lettore-scrittori deve soddisfare le seguenti condizioni:

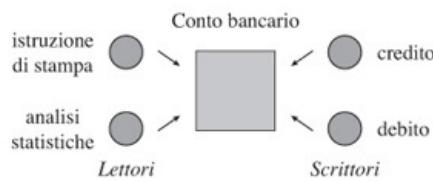
1. molti lettori possono effettuare la lettura in maniera concorrente;
2. la lettura è proibita quando uno scrittore sta eseguendo una scrittura;
3. solo uno scrittore per volta può eseguire la scrittura.

Le condizioni 1-3 non specificano quale processo dovrebbe essere preferito se un lettore e uno scrittore intendono accedere contemporaneamente ai dati condivisi. L'ulteriore condizione riportata di seguito viene richiesta se è importante dare una priorità maggiore ai lettori per soddisfare obiettivi specifici:

4. un lettore ha una priorità non prelazionabile rispetto agli scrittori; ovvero un lettore ottiene l'accesso ai dati condivisi prima di uno scrittore in attesa, ma non può prelazionare uno scrittore in esecuzione.

Questo sistema è chiamato *sistema lettore-scrittori con preferenza per i lettori*. Si può definire in maniera analoga un *sistema lettore-scrittori con preferenza per i lettori*.

La [Figura 6.17](#) illustra un esempio di sistema lettore-scrittori. I lettori e gli scrittori condividono un conto bancario. I processi lettori *stampa estratto conto* e *analisi statistica* leggono i dati dal conto bancario; pertanto possono essere eseguiti in maniera concorrente. *Credito* e *debito* modificano il bilancio del conto. Chiaramente solo uno dovrebbe essere attivo a ogni istante di tempo e nessuno dei lettori dovrebbe essere in esecuzione concorrente con uno scrittore che esegue queste funzioni. In un sistema di prenotazione, i processi che semplicemente richiedono la disponibilità di posti su di un volo sono i processi lettori, mentre i processi che effettuano le prenotazioni sono i processi scrittori poiché modificano porzioni del database delle prenotazioni.



**Figura 6.17** Lettori e scrittori in un sistema bancario.

I requisiti di sincronizzazione per il sistema lettori-scrittori vengono determinati come segue: le condizioni 1-3 consentono o a uno scrittore di eseguire la scrittura o a molti lettori di eseguire letture concorrenti. Dunque le scritture dovrebbero essere eseguite in una sezione critica per i dati condivisi. Quando uno scrittore termina la scrittura, dovrebbe o riattivare un altro scrittore, in modo tale che possa entrare nella sua sezione critica, o attivare *tutti* i lettori in attesa utilizzando una tecnica basata sulla segnalazione e un contatore dei lettori in attesa. Se i lettori stanno leggendo, uno scrittore in attesa può effettuare la scrittura quando l'ultimo lettore termina la lettura. Ciò richiede di memorizzare il numero di lettori concorrenti.

La [Figura 6.18](#) mostra una soluzione al problema dei lettori-scrittori. La scrittura viene effettuata in una sezione critica. Nei lettori non viene utilizzata una sezione critica, poiché questo non consentirebbe la concorrenza tra lettori. Per gestire il blocco e l'attivazione dei lettori e degli scrittori vengono adottati i segnali. Per semplicità, i dettagli relativi al conteggio dei lettori in attesa e dei lettori impegnati nella fasi di lettura non sono mostrati nella soluzione; saranno affrontati nel [Paragrafo 6.9.3](#). La soluzione di [Figura 6.18](#) non rispetta il requisito dell'attesa limitata per i lettori e gli scrittori; tuttavia, fornisce la massima concorrenza. Questa soluzione non favorisce né i lettori né gli scrittori.

---

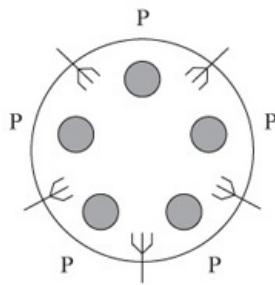
|  |  |
|--|--|
| <pre> Parbegin repeat   If uno scrittore sta scrivendo   then     { attendi };     { leggi }   If nessun lettore sta leggendo   then     if uno scrittore/i è in attesa     then       attiva uno scrittore in attesa;     forever;   Paren;   end. </pre> | <pre> repeat   If un lettore/i sta leggendo, o uno   scrittore sta scrivendo   then     { attendi };     { scrivi }   If un lettore/i o scrittore/i sono in attesa   then     attiva uno scrittore     o un lettore in attesa;   forever; </pre> |
| <u>Lettore/i</u>   | <u>Scrittore/i</u>   |

---

**Figura 6.18** Una soluzione del problema dei lettori-scrittori.

### 6.7.3 I filosofi a cena

Introduciamo un altro problema. Cinque filosofi, seduti intorno a un tavolo, discutono di filosofia. Ogni filosofo ha davanti a sé un piatto di spaghetti, mentre tra ogni coppia di filosofi c'è una forchetta ([Figura 6.19](#)). Per mangiare, un filosofo deve prendere, una alla volta, le due forchette che si trovano alla sua destra ed alla sua sinistra. Il problema consiste nel progettare i processi che rappresentano i filosofi in modo tale che ogni filosofo possa mangiare quando è affamato e nessuno muoia di fame.



**Figura 6.19** I filosofi a cena.

La condizione di correttezza nel sistema dei filosofi a cena è che un filosofo affamato non dovrebbe attendere indefinitamente quando decide di mangiare. L'obiettivo è di

progettare una soluzione che non soffra né di *deadlock*, in cui i processi si bloccano in attesa l'uno dell'altro (Paragrafo 1.4.2), né di *livelock*, in cui i processi non sono bloccati ma ritardano l'esecuzione a favore degli altri in modo indefinito. Si consideri la struttura di un processo filosofo  $P_i$  mostrato in [Figura 6.20](#), in cui sono stati omessi i dettagli relativi alla sincronizzazione dei processi. Un filosofo preleva le forchette una alla volta in qualsiasi ordine, per esempio, prima la forchetta di sinistra e poi quella di destra. Con questa soluzione è possibile che si verifichi una soluzione di deadlock, poiché se tutti i filosofi prelevano simultaneamente la loro forchetta di sinistra, nessuno sarebbe in grado di prendere la forchetta di destra! Inoltre questa soluzione contiene delle race condition poiché i filosofi vicini potrebbero contendersi una forchetta condivisa. È possibile evitare un deadlock modificando il processo filosofo in modo che se la forchetta di destra non è disponibile, il filosofo favorisce il suo vicino di sinistra posando la forchetta di sinistra e ripetendo l'operazione successivamente. Tuttavia, questo approccio soffre di livelock poiché la stessa situazione può verificarsi per ogni filosofo.

---

```

repeat
  if la forchetta di sinistra non è disponibile
  then
    block ( $P_i$ );
    prendi la forchetta di sinistra;
  if la forchetta di destra non è disponibile
  then
    block ( $P_i$ );
    prendi la forchetta di destra;
    { mangia }
    posa entrambe le forchette
  if il vicino di sinistra in attesa della sua forchetta di destra
  then
    activate (vicino di sinistra);
  if il vicino di destra in attesa della sua forchetta di sinistra
  then
    activate (vicino di destra);
    { pensa }
  forever

```

---

**Figura 6.20** Struttura del processo filosofo  $P_i$ .

Una soluzione più corretta per il problema dei filosofi a cena è fornita in [Figura 6.21](#). Un filosofo controlla la disponibilità delle forchette in una sezione critica nella quale prende anche le forchette. Per cui non si possono verificare race condition. Questa struttura assicura che almeno alcuni filosofi possono mangiare a ogni istante di tempo e previene il verificarsi di deadlock. Un filosofo che non riesce a prendere entrambe le forchette allo stesso tempo si blocca. Verrà riattivato quando uno dei suoi vicini posa una forchetta condivisa; pertanto il processo bloccato deve controllare nuovamente la disponibilità delle forchette. Questo è lo scopo del ciclo **while**. Tuttavia, il ciclo causa una condizione di attesa attiva. Alcune soluzioni innovative per il problema dei filosofi a cena prevengono i deadlock senza utilizzare l'attesa attiva (Problema 6.14). La prevenzione dei deadlock è discussa in dettaglio nel [Capitolo 8](#).

---

```

var      successful : boolean;
repeat
    successful := false;
    while (not successful)
        if entrambe le forchette sono disponibili then
            prendi le forchette una alla volta;
            successful := true;
        if successful = false
        then
            block (Pi);
            { mangia }
            posa entrambe le forchette;
        if il vicino di sinistra è in attesa della sua forchetta di destra
        then
            activate (vicino di sinistra);
        if il vicino di destra è in attesa della sua forchetta di sinistra
        then
            activate (vicino di destra);
            { pensa }
        all'infinito

```

---

**Figura 6.21** Una struttura migliore di un processo filosofo.

## 6.8 Approccio algoritmico per implementare le sezioni critiche

L'approccio algoritmico per l'implementazione delle sezioni critiche non utilizza né il blocco dei processi né i servizi di attivazione del kernel per ritardare un processo, o le istruzioni indivisibili per evitare le race condition. Di conseguenza, la sincronizzazione dei processi implementata con questo approccio è indipendente sia dal SO sia dal computer. Tuttavia, queste caratteristiche utilizzano l'attesa attiva per ritardare un processo in prossimità di un punto di sincronizzazione (Paragrafo 6.5.1) ed impiegano una complessa organizzazione di condizioni logiche per assicurare l'assenza di race condition, la qual cosa complica le dimostrazioni di correttezza. Per questo motivo, l'approccio algoritmico non è molto utilizzato.

Questo paragrafo descrive l'approccio algoritmico per l'implementazione delle sezioni critiche che, come mostrato nel Paragrafo 6.5.2, può essere usato sia per l'accesso sincronizzato ai dati sia per il controllo della sincronizzazione. Questo studio fornisce un approfondimento su come assicurare la mutua esclusione ed allo stesso tempo evitare sia i deadlock sia i livelock. Iniziamo affrontando gli schemi di implementazione della sezione critica utilizzati da due processi. Successivamente vedremo come estendere alcuni di questi schemi in modo da essere utilizzati da più di due processi.

### 6.8.1 Algoritmi per due processi

#### Algoritmo 6.1 Primo tentativo

```

var      turn : 1..2;
begin
    turn := 1;
Parbegin
    repeat
        while turn = 2
            do { niente };
        { Sezione critica }
        turn := 2;
        { Resto del ciclo }
    forever;
Parend;
end.

```

Processo  $P_1$

```

repeat
    while turn = 1
        do { niente };
    { Sezione critica }
    turn := 1;
    { Resto del ciclo }
forever;

```

Processo  $P_2$

La variabile *turn* è una variabile condivisa. La notazione 1..2 nella dichiarazione indica che può assumere o il valore 1 o il valore 2. Viene inizializzata a 1 prima che i processi  $P_1$  e  $P_2$  vengano creati. Ogni processo contiene una sezione critica per alcuni dati condivisi  $d_s$ . La variabile condivisa *turn* indica quale sarà il prossimo processo ad accedere alla sezione critica. Si supponga che il processo  $P_1$  voglia accedere alla sua sezione critica. Se *turn* = 1,  $P_1$  può entrare immediatamente. Una volta uscito dalla sua sezione critica,  $P_1$  imposta *turn* a 2 in modo tale che  $P_2$  possa entrare nella sua sezione critica. Se  $P_1$  trova *turn* = 2 quando vuole entrare nella sua sezione critica, attende nel ciclo **while** *turn* = 2 **do** { *niente* } finché  $P_2$  esce dalla sua sezione critica ed esegue l'assegnazione *turn* := 1. In questo modo la condizione di correttezza è soddisfatta.

L'Algoritmo 6.1 viola la condizione di progresso dell'implementazione della sezione critica descritta nella Tabella 6.1 a causa del modo in cui utilizza la variabile condivisa *turn*. Sia il processo  $P_1$  nella sua sezione critica ed il processo  $P_2$  nella parte restante del ciclo. Quando  $P_1$  esce dalla sua sezione critica, imposta *turn* a 2. Se completa la parte restante del ciclo e vuole entrare nuovamente nella sua sezione critica, sarà bloccato in attesa attiva finché il processo  $P_2$  non termina la sua sezione critica e imposta *turn* a 1. In questo modo, al processo  $P_1$  non è garantito l'ingresso nella sua sezione critica sebbene non ci siano altri processi interessati a utilizzare la propria sezione critica. L'Algoritmo 6.2 è un tentativo di eliminare questo problema.

### Algoritmo 6.2 Secondo tentativo

```

var      c1, c2 : 0..1;
begin
    c1 := 1;
    c2 := 1;
Parbegin
    repeat
        while c2 = 0
            do { niente };
        c1 := 0;
        { Sezione critica }
        c1 := 1;
        { Resto del ciclo }
    forever;
Parend;
end.

```

Processo  $P_1$

```

repeat
    while c1 = 0
        do { niente };
    c2 := 0;
    { Sezione critica }
    c2 := 1;
    { Resto del ciclo }
forever;

```

Processo  $P_2$

---

L'algoritmo utilizza due variabili condivise  $c_1$  e  $c_2$ , i cui valori sono ristretti a 0 o 1. Queste variabili possono essere considerate rispettivamente come flag di stato per i processi  $P_1$  e  $P_2$ .  $P_1$  imposta  $c_1$  a 0 quando entra nella sua sezione critica e la riporta a 1 una volta uscito dalla sua sezione critica. In questo modo, il valore di  $c_1 = 0$  indica che  $P_1$  è nella sua sezione critica e  $c_1 = 1$  indica che non è nella sua sezione critica.

Analogamente, il valore di  $c_2$  indica se  $P_2$  si trova nella propria sezione critica. Prima di entrare nella propria sezione critica, ogni processo controlla se l'altro processo è nella sezione critica. In caso negativo, entra subito nella propria sezione critica; altrimenti cicla finché l'altro processo non esce dalla sezione critica prima di entrare nella propria sezione critica. La violazione della condizione di progresso della Algoritmo 6.1 viene eliminata poiché i processi non sono costretti a un'alternanza stretta nell'utilizzo delle rispettive sezioni critiche.

L'Algoritmo 6.2 viola la condizione di mutua esclusione quando entrambi i processi tentano di entrare allo stesso tempo nelle rispettive sezioni critiche. Sia  $c_1$  che  $c_2$  saranno pari a 1 (poiché nessuno dei due processi si trova nella propria sezione critica) per cui entrambi i processi entreranno nelle sezioni critiche. Per evitare questo problema, le istruzioni "**while**  $c_2 = 0$  **do** { *niente* };" e " $c_1 := 0$ ;" nel processo  $P_1$  possono essere invertite; allo stesso modo, le istruzioni "**while**  $c_1 = 0$  **do** { *niente* };" e " $c_2 := 0$ ;" possono essere invertite nel processo  $P_2$ . In questo modo  $c_1$  sarà impostato a 0 prima che  $P_1$  controlli il valore di  $c_2$  e dunque entrambi i processi non potranno trovarsi nelle rispettive sezioni critiche allo stesso tempo. Tuttavia, se entrambi i processi cercano di entrare nelle loro sezioni critiche allo stesso tempo, sia  $c_1$  che  $c_2$  saranno a 0 per cui entrambi i processi aspetteranno l'un l'altro indefinitamente. In questo modo si crea una situazione di *stallo* (deadlock) (Paragrafo 1.4.2).

Sia la violazione della correttezza sia la possibilità di deadlock possono essere eliminate se un processo favorisce l'altro processo quando riscontra la volontà di quest'ultimo di entrare nella sezione critica. Questo può essere ottenuto come segue: se  $P_1$  riscontra che  $P_2$  sta cercando di entrare nella sua sezione critica, può impostare  $c_1$  a 0. Questo permetterà a  $P_2$  di entrare nella sua sezione critica.  $P_1$  può aspettare del tempo ed effettuare un altro tentativo per entrare nella sua sezione critica dopo aver impostato  $c_1$  a 1. Analogamente,  $P_2$  può impostare  $c_2$  a 0 se verifica che  $P_1$  sta cercando di entrare nella sua sezione critica. Tuttavia, questo approccio può portare a una situazione in cui entrambi i processi favoriscono l'altro indefinitamente. Questa è una situazione di *livelock* che abbiamo discusso precedentemente nel contesto del problema dei filosofi a cena (Paragrafo 6.7.3).

### **Algoritmo di Dekker**

L'algoritmo di Dekker combina le soluzioni adottate negli Algoritmi 6.1 e 6.2 per evitare una situazione di livelock. Se entrambi i processi tentano di entrare nelle loro sezioni critiche contemporaneamente, *turn* indica a quale processo dovrebbe essere consentita l'entrata. Nel caso in cui non ci sia competizione per entrare nella sezione critica, *turn* non ha alcun effetto.

Le variabili  $c_1$  e  $c_2$  vengono utilizzate come flag di stato dei processi come nell'Algoritmo 6.2. L'istruzione **while**  $c_2 = 0$  **do** in  $P_1$  controlla se è sicuro per  $P_1$  entrare nella sua sezione critica. Per evitare il problema della correttezza dell'Algoritmo 6.2, l'istruzione  $c_1 = 0$  in  $P_1$  precede l'istruzione **while**. Se  $c_2 = 1$  quando  $P_1$  vuole entrare in una sezione critica,  $P_1$  salta il ciclo **while** ed entra direttamente nella sezione critica. Se entrambi i processi tentano di entrare nelle rispettive sezioni critiche allo stesso tempo, il valore di *turn* forza uno dei due a favorire l'altro. Per esempio, se  $P_1$  trova  $c_2 = 0$ , favorisce  $P_2$  solo se *turn* = 2; altrimenti, semplicemente attende che  $c_2$  valga 1 prima di entrare nella sezione critica. Il processo  $P_2$ , che allo stesso tempo sta tentando di accedere alla sezione critica, è forzato a preferire  $P_1$  solo se *turn* = 1. In questo modo l'algoritmo soddisfa la mutua esclusione ed inoltre evita le condizioni di deadlock e livelock. Il valore attuale di *turn* in ogni istante non è determinante per la correttezza dell'algoritmo.

|  |
|--|
| <b>Algoritmo 6.3 Algoritmo di Dekker</b> |
|--|

```

var   turn : 1..2;
       c1, c2 : 0..1;
begin
  c1 := 1;
  c2 := 1;
  turn := 1;
Parbegin
  repeat
    c1 := 0;
    while c2 = 0 do
      if turn = 2 then
        begin
          c1 := 1;
          while turn = 2
            do { niente };
          c1 := 0;
        end;
        { Sezione critica }
        turn := 2;
        c1 := 1;
        { Resto del ciclo }
      forever;
Parend;
end.

```

Processo P<sub>1</sub>

```

repeat
  c2 := 0;
  while c1 = 0 do
    if turn = 1 then
      begin
        c2 := 1;
        while turn = 1
          do { niente }
        c2 := 0;
      end;
      { Sezione critica }
      turn := 1;
      c2 := 1;
      { Resto del ciclo }
    forever;

```

Processo P<sub>2</sub>

### Algoritmo di Peterson

L'algoritmo di Peterson è più semplice dell'algoritmo di Dekker. Utilizza un array di *flag* booleane che contengono una flag per ogni processo; queste flag sono equivalenti alle variabili di stato  $c_1, c_2$  dell'algoritmo di Dekker. Un processo imposta la sua flag a *true* quando vuole entrare in una sezione critica e la imposta a *false* quando esce dalla sezione critica. Si assume che i processi abbiano gli id  $P_0$  e  $P_1$ . L'id del processo è utilizzato come indice per accedere alla flag di stato di un processo nell'array *flag*. La variabile *turn* viene usata per evitare i livelock; tuttavia, è utilizzata in maniera differente rispetto all'algoritmo di Dekker.

Un processo che desidera entrare in una sezione critica comincia concedendo l'accesso all'altro processo impostando *turn* in modo che contenga l'id dell'altro processo. Tuttavia, prosegue ed entra nella sezione critica se verifica che l'altro processo non è interessato a utilizzare la sua sezione critica. Se entrambi i processi cercano di entrare nelle rispettive sezioni critiche allo stesso tempo, il valore di *turn* determina quale processo può entrare. Per esempio, si consideri il processo  $P_0$ . Questo imposta  $flag[0]$  a *true* e *turn* a 1 quando vuole entrare nella sua sezione critica. Se il processo  $P_2$  non è interessato a utilizzare la sua sezione critica,  $flag[1]$  sarà falso, per cui  $P_0$  uscirà dal ciclo **while** per entrare direttamente nella sua sezione critica. Se anche  $P_1$  è interessato a entrare nella sua sezione critica,  $flag[1]$  varrà *true*. In questo caso, il valore di *turn* determina quale processo può entrare nella propria sezione critica.

È interessante considerare il funzionamento dell'algoritmo di Peterson nel caso in cui  $P_0$  e  $P_1$  abbiano velocità relative differenti. Si consideri la situazione in cui entrambi i processi  $P_0$  e  $P_1$  vogliono usare le loro sezioni critiche e  $P_0$  sia leggermente in anticipo rispetto a  $P_1$ . Se entrambi i processi vengono eseguiti alla stessa velocità,  $P_0$  entrerà nella propria sezione critica prima di  $P_1$  poiché  $P_1$  dovrà impostare *turn* a 0 nel momento in cui  $P_1$  raggiunge l'istruzione **while**.  $P_1$  attende nel ciclo **while** finché  $P_0$  non esce dalla sua sezione critica. Se, tuttavia,  $P_0$  è più lento di  $P_1$ , imposterà *turn* a 1 a un certo momento dopo che  $P_1$  lo abbia impostato a 0. Dunque  $P_0$  attenderà nel ciclo **while** e  $P_1$  entrerà nella sua sezione critica.

### Algoritmo 6.4 Algoritmo di Peterson

|   |  |
|---|--|
| <pre> <b>var</b>      flag : <b>array</b> [0..1] <b>of</b> boolean;           turn : 0..1; <b>begin</b>           flag[0] := false;           flag[1] := false; <b>Parbegin</b>           <b>repeat</b>           flag[0] := true;           turn := 1;           <b>while</b> flag[1] <b>and</b> turn = 1           <b>do</b> { niente };           { Sezione critica }           flag[0] = false;           { Resto del ciclo } <b>forever</b>; <b>Parend</b>; <b>end.</b> </pre> | <pre> <b>repeat</b>           flag[1] := true;           turn := 0;           <b>while</b> flag[0] <b>and</b> turn =           <b>do</b> { niente };           { Sezione critica }           flag[1] = false;           { Resto del ciclo } <b>forever</b>; </pre> |
| <i>Processo P<sub>1</sub></i>   | <i>Processo P<sub>2</sub></i>  |

### 6.8.2 Algoritmi per $n$ processi

In una implementazione algoritmica di una sezione critica, l'algoritmo deve conoscere il *numero* di processi che utilizzano la sezione critica per gli stessi dati. Questa conoscenza è in relazione con molte caratteristiche del codice: la dimensione dell'array delle flag di stato, i controlli per determinare se qualche altro processo desidera entrare in una sezione critica ed il meccanismo in base al quale un processo ne favorisce un altro. Ognuna di queste caratteristiche deve essere modificata se cambia il numero di processi che devono essere gestiti dalla sezione critica. Per esempio, in un'implementazione della sezione critica per due processi, ogni processo deve controllare lo stato di un solo processo e nel caso favorirlo per l'esecuzione, per assicurare la correttezza e l'assenza di deadlock e livelock. In un'implementazione a  $n$  processi della sezione critica, un processo deve controllare lo stato di  $n - 1$  altri processi e deve farlo in modo da prevenire race condition. Questo rende l'algoritmo per  $n$  processi più complesso. Analizziamo questa condizione nel contesto dell'algoritmo di Eisenberg e McGuire (1972), che estende a  $n$  processi la soluzione per due processi dell'algoritmo di Dekker.

#### Algoritmo 6.5 Un Algoritmo per $n$ processi [Eisenberg e McGuire (1972)]

```

const       $n = \dots;$ 
var         $flag : \text{array}[0..n-1] \text{ of } (idle, want\_in, in\_CS);$ 
             $turn : 0..n-1;$ 
begin
    for  $j := 0$  to  $n-1$  do
         $flag[j] := idle;$ 
Parbegin
    processo  $P_i$  :
        repeat
            repeat
                 $flag[i] := want\_in;$ 
                 $j := turn;$ 
                while  $j \neq i$ 
                    do if  $flag[j] \neq idle$ 
                    then  $j := turn \{ Cicla qui! \}$ 
                    else  $j := j + 1 \bmod n;$ 
                 $flag[i] := in\_CS;$ 
                 $j := 0;$ 
                while  $(j < n) \text{ e } (j = i \text{ or } flag[j] \neq in\_CS)$ 
                    do  $j := j + 1;$ 
                until  $(j \geq n) \text{ and } (turn = i \text{ or } flag[turn] = idle);$ 
                 $turn := i;$ 
                 $\{ \text{Sezione critica} \}$ 
                 $j := turn + 1 \bmod n;$ 
                while  $(flag[j] = idle) \text{ do } j := j + 1 \bmod n;$ 
                 $turn := j;$ 
                 $flag[i] := idle;$ 
                 $\{ \text{Resto del ciclo} \}$ 
            forever
    processo  $P_k : \dots$ 
Parend;
end.

```

La variabile  $turn$  indica qual è il prossimo processo che può entrare nella sua sezione critica. Il suo valore iniziale non è determinante per la correttezza dell'algoritmo. Ogni processo ha una flag di stato che può assumere tre possibili valori  $idle$ ,  $want\_in$  e  $in\_CS$  ed è inizializzata a  $idle$ . Un processo imposta la propria flag a  $want\_in$  quando vuole entrare in una sezione critica. A questo punto deve decidere se può impostare il flag come  $in\_CS$ . Per effettuare questa decisione, controlla le flag degli altri processi secondo l'ordine modulo  $n$ . L'ordine modulo  $n$  è  $P_{turn}, P_{turn+1}, \dots, P_{n-1}, P_0, P_1, \dots, P_{turn-1}$ . Nel primo ciclo **while**, il processo controlla se qualche altro processo che lo precede secondo l'ordine modulo  $n$  vuole utilizzare la sua sezione critica. In caso negativo, imposta la sua flag come  $in\_CS$ .

Poiché i processi possono effettuare questo controllo in maniera concorrente, più di un processo può arrivare simultaneamente alla stessa conclusione. Per questo motivo viene effettuato un altro controllo per assicurare la correttezza. Il secondo ciclo **while** controlla se qualche altro processo ha impostato la propria flag come  $in\_CS$ . In caso affermativo, il processo imposta la propria flag nuovamente come  $want\_in$  e ripete i controlli. Questi processi non competono nuovamente per entrare nella sezione critica poiché hanno tutti le proprie flag impostate a  $want\_in$  e dunque solo uno di loro sarà in grado di superare il primo ciclo **while**. Questa caratteristica evita la condizione di livelock. Il primo processo nell'ordine modulo  $n$  a partire da  $P_{turn}$  otterrà l'accesso ed entrerà nella sua sezione critica prima degli altri processi. Successivamente, cambia il proprio flag a  $idle$  quando lascia la sua sezione critica. In questo modo la flag contiene il valore  $idle$  quando un processo si trova nella parte restante del suo ciclo.

Questa soluzione non è equa poiché i processi non accedono alle proprie sezioni critiche nello stesso ordine in cui hanno richiesto l'accesso alla sezione critica. Questo problema viene eliminato nell'algoritmo del "panettiere" (Bakery) progettato da Lamport (1974).

### Algoritmo del panettiere (Bakery)

Quando un processo vuole accedere a una sezione critica, sceglie un numero maggiore dei numeri scelti dai processi precedenti. A tal fine, adottiamo un array di flag booleane denominato *choosing*. *choosing[i]* viene usato per indicare se il processo  $P_i$  al momento è impegnato nella scelta di un numero. *number[i]* contiene il numero scelto dal processo  $P_i$ .  $number[i] = 0$  se  $P_i$  non ha scelto un numero dall'ultima volta in cui ha avuto accesso alla sezione critica. L'idea alla base dell'algoritmo è che i processi dovrebbero accedere alle proprie sezioni critiche secondo l'ordine dei numeri scelti. Nel seguito discuteremo il funzionamento dell'algoritmo.

Un processo che vuole entrare in una sezione critica favorisce un processo con un numero minore. Tuttavia, una regola di "spareggio" è necessaria poiché i processi che scelgono i numeri in maniera concorrente possono ottenere lo stesso numero. A questo scopo l'algoritmo utilizza la coppia  $(number[i], i)$ . Un processo accede alla sezione critica se la sua coppia precede ogni altra coppia, dove la relazione di precedenza  $<$  è definita come segue:

$$(number[j], j) < (number[i], i) \text{ se} \\ number[j] < number[i], \text{ oppure} \\ number[j] = number[i] \text{ and } j < i.$$

In questo modo, se più processi ottengono lo stesso numero, il processo con l'id minore entra per primo nella sua sezione critica. In tutti gli altri casi, i processi entrano nelle rispettive sezioni critiche nell'ordine in cui hanno presentato la richiesta di accesso.

#### Algoritmo 6.6 Algoritmo del panettiere (Bakery) (Lamport [1974])

```

const n = ...;
var   choosing : array [0 .. n - 1] of boolean;
        number : array [0 .. n - 1] of integer;
begin
    for j := 0 to n - 1 do
        choosing[j] := false;
        number[j] := 0;
Parbegin
    process  $P_i$  :
        repeat
            choosing[i] := true;
            number[i] := max (number[0], ..., number[n - 1]) + 1;
            choosing[i] := false;
            for j := 0 to n - 1 do
                begin
                    while choosing[j] do { nothing };
                    while number[j] ≠ 0 and (number[j], j) < (number[i], i)
                        do { nothing };
                end;
                { Sezione critica }
                number[i] := 0;
                { Resto del ciclo }
            forever;
            process  $P_j$  : ...
Parend;
end.

```

## 6.9 Semafori

Come anticipato nel Paragrafo 6.5.3, le *primitive di sincronizzazione* sono state sviluppate per superare le limitazioni delle implementazioni algoritmiche. Le primitive sono semplici operazioni che possono essere usate per implementare sia la mutua esclusione sia il controllo della sincronizzazione. Un *semaforo* è un tipo speciale di struttura di sincronizzazione che può essere usato solo mediante specifiche primitive di sincronizzazione.

**Definizione 6.5 Semaforo** Una variabile intera condivisa a valori non negativi che può essere soggetta solo alle seguenti operazioni:

1. Inizializzazione (specificata come parte della sua dichiarazione)
2. Le operazioni indivisibili *wait* e *signal*

Le operazioni *wait* e *signal* su un semaforo erano originariamente chiamate da Dijkstra operazioni P e V dai termini Proberen e Verlagen. La loro semantica è mostrata nella **Figura 6.22**. Quando un processo effettua un'operazione *wait* su un semaforo, l'operazione verifica se il valore del semaforo è  $> 0$ . In caso affermativo, decremente il valore del semaforo e consente al processo di proseguire la sua esecuzione; altrimenti, blocca il processo sul semaforo. Un'operazione *signal* su un semaforo attiva un processo bloccato sul semaforo, se ve ne sono, oppure incrementa il valore del semaforo di una unità. A causa di queste semantiche, i semafori sono anche chiamati *semafori contatori*. L'*indivisibilità delle operazioni wait e signal* è assicurata dal linguaggio di programmazione o dal sistema operativo che le implementa, assicurando che non si possano verificare race condition su un semaforo (Paragrafo 6.9.4).

```
procedure wait (S)
begin
  if S > 0
    then S := S-1;
    else blocca il processo su S;
end;

procedure signal (S)
begin
  if qualche processo è bloccato su S
    then attiva un processo bloccato;
    else S := S+1;
end;
```

**Figura 6.22** Semantica delle operazioni *wait* e *signal* su un semaforo.

I processi utilizzano le operazioni *wait* e *signal* per sincronizzare la loro esecuzione nei confronti l'uno dell'altro. Il valore iniziale di un semaforo determina quanti processi possono effettuare l'operazione *wait* senza essere bloccati. Un processo che non passa l'operazione *wait* viene bloccato sul semaforo. Questa caratteristica evita le attese attive. Il Paragrafo 6.9.1 descrive l'uso dei semafori. I Paragrafi 6.9.2 e 6.9.3 descrivono l'implementazione dei problemi produttori-consumatori e lettori-scrittori utilizzando i semafori.

### 6.9.1 Uso dei semafori nei sistemi concorrenti

La **Tabella 6.3** riassume tre utilizzi dei semafori nell'implementazione dei sistemi concorrenti. La *mutua esclusione* è utile per implementare le sezioni critiche. La *concorrenza limitata* è importante quando una risorsa può essere condivisa da al più  $c$  processi, dove  $c$  è una costante  $\geq 1$ . La *segnalazione* è utile nel controllo della sincronizzazione. Nel seguito del paragrafo discuteremo i loro dettagli.

| Uso              | Descrizione  |
|------------------|--|
| Mutua esclusione | La mutua esclusione può essere implementata usando un semaforo inizializzato a 1. Un processo effettua |

|                      |   |
|----------------------|---|
|                      | un'operazione <i>wait</i> sul semaforo prima di entrare in una CS e un'operazione <i>signal</i> al termine della CS. Uno speciale tipo di semaforo chiamato <i>semaforo binario</i> semplifica ulteriormente l'implementazione della CS.  |
| Concorrenza limitata | La concorrenza limitata implica che una funzione può essere eseguita, o una risorsa può essere utilizzata, da $n$ processi concorrentemente, $1 \leq n \leq c$ , con $c$ costante. Un semaforo inizializzato a $c$ può essere utilizzato per implementare la concorrenza limitata.  |
| Segnalazione         | La segnalazione viene usata quando un processo $P_i$ vuole effettuare un'operazione $a_i$ solo dopo che il processo $P_j$ ha effettuato una operazione $a_j$ . Viene implementata utilizzando un semaforo inizializzato a 0. $P_i$ effettua una <i>wait</i> sul semaforo prima di effettuare l'operazione $a_i$ . $P_j$ effettua una <i>signal</i> sul semaforo dopo aver effettuato l'operazione $a_j$ . |

**Tabella 6.3** Uso dei semafori nell'implementazione dei sistemi concorrenti.

#### Mutua esclusione

La [Figura 6.23](#) mostra l'implementazione di una sezione critica nei processi  $P_i$  e  $P_j$  utilizzando un semaforo chiamato *sem\_CS*. *sem\_CS* è inizializzato a 1. Ogni processo effettua un'operazione *wait* su *sem\_CS* prima di entrare nella propria sezione critica ed un'operazione *signal* all'uscita dalla sezione critica. Il primo processo a effettuare la *wait(sem\_CS)* trova che *sem\_CS* è  $> 0$ . Dunque decremente *sem\_CS* di una unità e procede entrando nella sua sezione critica. Quando il secondo processo effettua la chiamata *wait(sem\_CS)*, si blocca su *sem\_CS* poiché il suo valore è 0. Viene riattivato quando il primo processo effettua la chiamata *signal(sem\_CS)*, una volta terminata l'esecuzione della sua sezione critica; il secondo processo allora entra nella sua sezione critica. Se non ci sono processi bloccati su *sem\_CS* quando viene effettuata la chiamata *signal(sem\_CS)*, il valore di *sem\_CS* diventa 1. Questo valore di *sem\_CS* consente a un processo, che successivamente effettua un'operazione di *wait*, di entrare immediatamente nella propria sezione critica. Più processi che utilizzano codice simile possono essere aggiunti al sistema senza causare problemi di correttezza. Il prossimo esempio illustra il funzionamento di questo sistema.

```

var sem_CS : semaforo := 1;
Parbegin
repeat
  wait (sem_CS);
  { Sezione critica }
  signal (sem_CS);
  { Parte restante del ciclo }
  all'infinito;
Parend;
end.

```

*Processo P<sub>i</sub>*

```

repeat
  wait (sem_CS);
  { Sezione critica }
  signal (sem_CS);
  { Parte restante del ciclo }
  all'infinito;

```

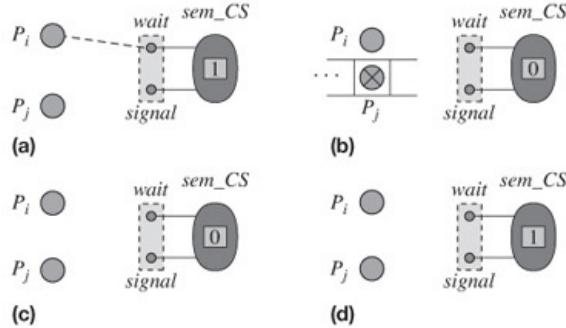
*Processo P<sub>j</sub>*

**Figura 6.23** Implementazione della CS con semafori.

#### Esempio 6.4 - Sezioni critiche mediante semafori

La [Figura 6.24](#) mostra le istantanee prese durante il funzionamento del sistema mostrato in [Figura 6.23](#). Le operazioni *wait* e *signal* su *sem\_CS* sono racchiuse in un rettangolo tratteggiato poiché sono mutuamente esclusive (si faccia riferimento alle convenzioni della [Figura 6.11](#)). Si consideri che il processo  $P_i$  abbia eseguito la *wait(sem\_CS)*. La [Figura 6.24\(a\)](#) illustra la situazione all'inizio dell'operazione *wait* e seguita da  $P_j$ . La [Figura 6.24\(b\)](#) mostra la situazione dopo che  $P_i$  ha completato l'operazione *wait* e dopo che  $P_j$  ha completato un'operazione di *wait*: l'operazione *wait(sem\_CS)* effettuata da  $P_j$  ha portato il valore di *sem\_CS* a 0, per cui  $P_j$  si blocca sull'operazione *wait*. La [Figura 6.24\(c\)](#) mostra la situazione dopo che il processo  $P_i$  ha

è eseguito un'operazione *signal*. Il valore di *sem\_CS* resta a 0, ma il processo  $P_j$  è stato riattivato. Il processo  $P_j$  effettua un'operazione *signal* uscendo dalla sua sezione critica. Poiché non ci sono processi bloccati su *sem\_CS*, l'esecuzione dell'operazione *signal* da parte del processo  $P_j$  risulta semplicemente nell'incremento del valore di *sem\_CS* di una unità (Figura 6.24(d)).



**Figura 6.24** Instantanee del sistema concorrente di Figura 6.23.

È interessante verificare quali proprietà delle implementazioni della sezione critica menzionate nella Tabella 6.1 sono soddisfatte dall'implementazione della Figura 6.23. La mutua esclusione segue dal fatto che *sem\_CS* è inizializzato a 1. L'implementazione rispetta la condizione di progresso poiché un processo che effettua l'operazione *wait* ottiene l'accesso alla propria sezione critica se non c'è un altro processo nella propria sezione critica. Tuttavia, la proprietà di attesa limitata non è verificata poiché l'ordine con cui i processi bloccati vengono riattivati dalle operazioni *signal* non è definito nella semantica dei semafori. Per questo motivo un processo bloccato potrebbe soffrire di starvation se altri processi effettuassero ripetutamente le operazioni *wait* e *signal*.

Possono verificarsi problemi di correttezza poiché le operazioni *wait* e *signal* sono *primitive*, per cui un programma potrebbe utilizzarle in maniera non corretta. Per esempio, il processo  $P_i$  di Figura 6.23 potrebbe essere stato implementato come segue:

```
repeat
    signal(sem_CS);
    { Sezione critica }
    signal(sem_CS);
    { Resto del ciclo }
forever
```

in cui, all'ingresso della sezione critica di  $P_i$ , è stata usata una *signal(sem\_CS)* al posto di una *wait(sem\_CS)*. A questo punto la sezione critica non sarebbe stata implementata correttamente poiché molti processi sarebbero in grado di entrare nelle rispettive sezioni critiche allo stesso tempo. Come altro esempio, si consideri cosa succederebbe se il codice del processo  $P_i$  utilizzasse erroneamente una *wait(sem\_CS)* al posto di una *signal(sem\_CS)* all'uscita della sezione critica. Al termine dell'esecuzione della sua sezione critica,  $P_i$  resterebbe bloccato sull'operazione *wait* poiché il valore di *sem\_CS* è 0. Altri processi che volessero accedere alla sezione critica resterebbero bloccati sull'operazione *wait* che precede la sezione critica. Dal momento che nessun processo esegue un'operazione *signal* su *sem\_CS*, tutti questi processi rimarrebbero bloccati indefinitamente, generando una situazione di *deadlock*.

### Semafori binari

Un *semaforo binario* è un tipo speciale di semaforo utilizzato per implementare la mutua esclusione. Dunque è spesso chiamato *mutex*. Un semaforo binario è inizializzato a 1 ed assume solo i valori 0 e 1 durante l'esecuzione del programma. Le operazioni *wait* e *signal* su un semaforo binario sono leggermente differenti da quelle mostrate nella Figura 6.22; l'istruzione  $S = S - 1$  nell'operazione *wait* è sostituita dall'istruzione  $S = 0$  e l'istruzione  $S = S + 1$  nella *signal* è sostituita dall'istruzione  $S = 1$ .

### Concorrenza limitata

Utilizziamo il termine *concorrenza limitata* per il caso in cui al più  $c$  processi possono effettuare in maniera concorrente un'operazione  $op_i$ , con  $c$  costante  $\geq 1$ . La concorrenza limitata viene implementata inizializzando un semaforo *sem-c* a  $c$ . Ogni processo che voglia effettuare  $op_i$  richiama una  $wait(sem\_c)$  prima di effettuare  $op_i$ , ed una  $signal(sem\_c)$  al termine di  $op_i$ . Dalla semantica della *wait* e della *signal*, risulta chiaro che al più  $c$  processi possono eseguire concorrentemente  $op_i$ . I semafori usati per implementare la concorrenza limitata sono denominati *semafori contatore*.

### Segnalazione tra processi

Si considerino i requisiti di sincronizzazione dei processi  $P_i$  e  $P_j$  mostrati in [Figura 6.6](#) – il processo  $P_i$  dovrebbe effettuare un'operazione  $a_i$  solo dopo che il processo  $P_j$  ha effettuato un'operazione  $a_j$ . Un semaforo può essere utilizzato per realizzare tale sincronizzazione come mostrato in [Figura 6.25](#). Qui il processo  $P_i$  esegue una  $wait(sync)$  prima di eseguire un'operazione  $a_i$  e  $P_j$  effettua una  $signal(sync)$  dopo aver eseguito l'operazione  $a_j$ . Il semaforo *sync* è inizializzato a 0 per cui  $P_j$  verrà bloccato su  $wait(sync)$  se  $P_j$  non ha già effettuato una  $signal(sync)$ . Diversamente dalla soluzione di [Figura 6.6](#), non si possono verificare race condition poiché le operazioni *wait* e *signal* sono indivisibili. L'organizzazione per la segnalazione può essere utilizzata ripetutamente, dal momento che l'operazione *wait* porta il valore di *sync* nuovamente a 0.

---

```
var sync : semaforo := 0;
Parbegin
  ...
  wait (sync);
  { Esegui l'azione ai }
  ...
  signal (sync);
  { Esegui l'azione aj }
Parend;
end.
```

---

*Processo P<sub>i</sub>*                            *Processo P<sub>j</sub>*

---

**Figura 6.25** Segnalazione mediante semafori.

### 6.9.2 Produttori-consumatori usando i semafori

Come discusso nel [Paragrafo 6.7.1](#), il problema dei produttori-consumatori è un problema di segnalazione. Dopo aver inserito un elemento di informazione in un buffer, un produttore segnala l'evento a un consumatore in attesa di consumare un elemento dal buffer. In maniera analoga, un consumatore segnala a un produttore l'estrazione di un elemento dal buffer. Dovremmo pertanto implementare i produttori-consumatori utilizzando l'organizzazione per la segnalazione mostrata in [Figura 6.25](#).

Per semplicità, illustriamo la soluzione per il caso con singolo buffer mostrata in [Figura 6.26](#). Il pool di buffer è rappresentato da un array di buffer con un singolo elemento all'interno. Vengono dichiarati due semafori *pieno* e *vuoto* e sono utilizzati per indicare, rispettivamente, il numero di buffer pieni e vuoti. Un produttore effettua una  $wait(vuoto)$  prima di iniziare l'operazione di inserimento nel buffer ed un consumatore effettua una  $wait(pieno)$  prima di iniziare l'operazione di estrazione dal buffer.

---

```

type    item = . . .;
var
    full : Semaforo := 0; { Inizializzazione }
    empty : Semaforo := 1;
    buffer : array [0] of item;
begin
Parbegin
repeat
    wait (vuoto);
    buffer [0] := . . .;
    { i.e. produci }
    signal (pieno);
    { Parte restante del ciclo }
all'infinito;
Parend;
end.

Produttore
Consumatore
repeat
    wait (full);
    x := buffer [0];
    { i.e. consuma }
    signal (vuoto);
    { Parte restante del ciclo }
all'infinito;

```

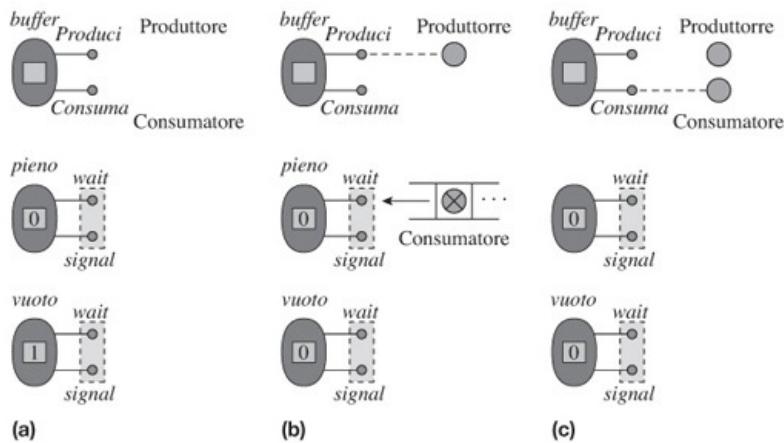
---

**Figura 6.26** Produttori-consumatori con un singolo buffer.

Inizialmente, il semaforo *pieno* ha valore 0. Dunque il consumatore si bloccherà sulla *wait(piemo)*. *vuoto* ha valore 1, per cui un produttore effettuerà l'operazione *wait(vuoto)*. Dopo aver completato l'operazione di inserimento effettua una *signal(piemo)*. Questo consente a un consumatore di entrare, immediatamente oppure successivamente, nella propria sezione critica. Quando il consumatore termina l'estrazione dal buffer, effettua una *signal(vuoto)* che consente a un produttore di effettuare un'operazione di inserimento. Questa soluzione evita le attese attive poiché i semafori sono utilizzati per controllare i buffer pieni o vuoti, per cui un processo sarà bloccato se non trova un buffer vuoto o pieno come richiesto. Il livello di concorrenza in questo sistema è 1; a volte viene eseguito un produttore a volte viene eseguito un consumatore. L'Esempio 6.5 descrive il funzionamento di questa soluzione.

#### Esempio 6.5 - Produttori-consumatori con singolo buffer mediante semafori

L'istantanea di [Figura 6.27\(a\)](#) mostra la situazione iniziale nel sistema produttori-consumatori di [Figura 6.26](#). La [Figura 6.27\(b\)](#) mostra la situazione in cui i processi produttore e consumatore tentano, rispettivamente, di produrre e consumare. Il processo produttore ha superato l'operazione *wait* su *vuoto* poiché *vuoto* era inizializzato a 1. Il valore del semaforo *vuoto* diventa 0 ed il produttore inizia l'inserimento nel buffer. Il processo consumatore è bloccato sull'operazione *wait(piemo)* poiché *piemo* è 0. Quando il produttore esegue una *signal(piemo)* dopo l'operazione di inserimento, il processo consumatore viene attivato ed inizia l'estrazione dal buffer. La [Figura 6.27\(c\)](#) illustra questa situazione.



**Figura 6.27** Instantanee del problema produttori-consumatori con singolo buffer utilizzando i semafori.

La [Figura 6.28](#) mostra come i semafori possano essere usati per implementare una soluzione del problema produttori-consumatori con  $n$  buffer,  $n \geq 1$ , prevedendo un processo produttore ed un processo consumatore. Questa soluzione è una semplice estensione della soluzione al problema con singolo buffer mostrata in [Figura 6.26](#). I valori dei semafori *vuoto* e  *pieno* indicano il numero di buffer, rispettivamente, vuoti e pieni, per cui sono inizializzati, rispettivamente, a  $n$  e a 0. *prod\_ptr* e *cons\_ptr* sono usati come indici dell'array *buffer*. Il produttore inserisce un elemento in *buffer[prod\_ptr]* e incrementa *prod\_ptr*. Il consumatore estraе da *buffer[cons\_ptr]* e incrementa *cons\_ptr* allo stesso modo. Questa caratteristica assicura che i buffer sono utilizzati in ordine FIFO. Un produttore ed un consumatore possono funzionare concorrentemente finché vi sono buffer pieni e vuoti nel sistema.

---

```

const          n = . . .;
type           item = . . .;
var
  buffer : array [0..n - 1] of item;
  full : Semaforo := 0; { Inizializzazioni }
  empty : Semaforo := n;
  prod_ptr, cons_ptr : integer;
begin
  prod_ptr := 0;
  cons_ptr := 0;
Parbegin
repeat          repeat
  wait (vuoto);    wait ( pieno);
  buffer [prod_ptr] := . . .;  x := buffer [cons_ptr];
  { i.e. produci }  { i.e. consuma }
  prod_ptr := prod_ptr + 1 mod n;  cons_ptr := cons_ptr + 1 mod n;
  signal ( pieno);  signal ( vuoto);
  { Parte restante del ciclo }  { Parte restante del ciclo }
  all'infinito;        all'infinito;
Parend;
end.

```

Produttore
Consumatore

---

**Figura 6.28** Produttori-consumatori a buffer limitati utilizzando i semafori.

È facile verificare che questa soluzione implementa le condizioni di correttezza del problema del buffer limitato descritto nel [Paragrafo 6.7.1](#). Tuttavia, se molti processi produttori e consumatori sono attivi nel sistema, è necessario garantire la mutua esclusione tra i produttori per evitare race condition su *prod\_ptr*. Analogamente, la mutua esclusione dovrebbe essere garantita tra i consumatori per evitare race condition su *cons\_ptr*.

### 6.9.3 Lettori-scrrittori utilizzando i semafori

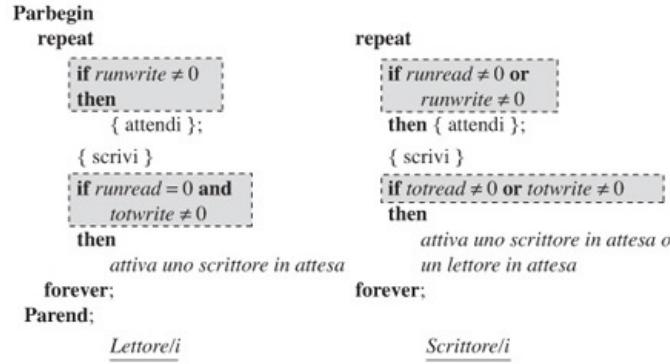
Una caratteristica essenziale del problema dei lettori-scrrittori è che i lettori e gli scrittori devono attendere mentre uno scrittore sta scrivendo e, quando lo scrittore termina, o vengono attivati tutti i lettori in attesa o viene attivato uno scrittore in attesa ([Figura 6.18](#)). Per implementare questa caratteristica, utilizziamo quattro contatori come segue:

```

runread numero di lettori in lettura
totread numero di lettori in attesa di lettura o attualmente in lettura
runwrite numero di scrittori attualmente in scrittura
totwrite numero di scrittori in attesa di scrittura o attualmente in scrittura

```

Con questi contatori, la [Figura 6.18](#) viene modificata come mostrato in [Figura 6.29](#) (non mostriamo i dettagli di come i contatori vengono aggiornati). A un lettore è consentito di iniziare la lettura quando *runread* = 0 e *runwrite* = 0. Il valore di *totread* viene usato per attivare tutti i lettori in attesa quando uno scrittore termina la scrittura. Questa soluzione non usa una sezione critica esplicita per gli scrittori. Al contrario gli scrittori sono bloccati finché non gli è consentito di iniziare a scrivere.



**Figura 6.29** Bozza di una soluzione per i lettori-scrrittori.

Il blocco dei lettori e degli scrittori ricorda il blocco dei produttori e dei consumatori nel problema dei produttori-consumatori, per cui può essere gestito utilizzando i semafori per la segnalazione. Introduciamo due semafori chiamati *reading* e *writing*. Un processo lettore esegue una *wait(reading)* prima di iniziare a leggere. Questa operazione dovrebbe bloccare il processo lettore se non sono soddisfatte le condizioni per la lettura; altrimenti, il lettore dovrebbe essere in grado di iniziare a leggere. Analogamente, un processo scrittore esegue una *wait(writing)* prima di iniziare a scrivere e si blocca se non sono soddisfatte le condizioni appropriate. Le condizioni su cui si bloccano lettori e scrittori possono cambiare quando uno dei contatori cambia, ovvero, quando un lettore termina la lettura o uno scrittore termina la scrittura. Per questo motivo i processi lettore e scrittore dovrebbero eseguire le appropriate operazioni di *signal* dopo aver completato un'operazione di lettura o scrittura.

Questa soluzione è implementata come segue (Figura 6.30): per evitare race condition tutti i contatori vengono esaminati e modificati all'interno di sezioni critiche implementate utilizzando un semaforo binario chiamato *sem\_CS*. Quando un lettore vuole iniziare una lettura, entra nella sezione critica protetta da *sem\_CS* per controllare se *runwrite* = 0. In caso affermativo, incrementa *runread*, esce dalla sezione critica e inizia a leggere. In caso negativo, deve eseguire una *wait(reading)*; tuttavia, eseguire un'operazione *wait(reading)* all'interno della sezione critica protetta da *sem\_CS* può causare un deadlock, per cui esegue una *signal(reading)* dopo essere uscito dalla sezione critica. Se le condizioni consentono l'inizio di un'operazione di lettura quando il processo esamina il contatore, dovrebbe eseguire una *signal(reading)* all'interno della sezione critica. Questo lettore supererebbe l'operazione *wait(reading)*. Uno scrittore, in maniera analoga, effettua una *signal\_writing* all'interno della propria sezione critica protetta da *sem\_CS* sussistendo il corretto insieme di condizioni ed una *wait\_writing* al termine della sezione critica.

```

var
  totread, runread, totwrite, runwrite : integer;
  reading, writing : semaforo := 0;
  sem_CS : semaforo := 1;
begin
  totread := 0;
  runread := 0;
  totwrite := 0;
  runwrite := 0;
Parbegin
repeat
  wait (sem_CS);
  totread := totread + 1;
  if runwrite = 0 then
    runread := runread + 1;
    signal (reading);
    signal (sem_CS);
    wait (reading);
    { Leggi }
    wait (sem_CS);
    runread := runread - 1;
    totread := totread - 1;
    if runread = 0 and
      totwrite > runwrite
    then
      runwrite := 1;
      signal (writing);
      signal (sem_CS);
forever;
repeat
  wait (sem_CS);
  totwrite := totwrite + 1;
  if runread = 0 and runwrite = 0 then
    runwrite := 1;
    signal (writing);
    signal (sem_CS);
  while (runread < totread) do
begin
  runread := runread + 1;
  signal (reading);
end;
  if runread = 0 and
    totwrite > runwrite then
    runwrite := 1;
    signal (writing);
    signal (sem_CS);
forever;
Parend;
end.

```

Lettore/i

Scrittore/i

**Figura 6.30** Una soluzione al problema dei lettori-scrittori che favorisce i lettori, utilizzando i semafori.

I lettori e gli scrittori che si bloccano sulle rispettive operazioni *wait* sono attivati come segue: quando il lettore termina una lettura, esegue un'operazione di *signal* per attivare uno scrittore se non ci sono lettori attivi e uno scrittore è in attesa. Quando uno scrittore termina la scrittura, effettua tante operazioni di *signal* quanti sono i lettori in attesa; altrimenti esegue una *signal* per risvegliare uno scrittore in attesa, se esiste. Dunque il sistema risultante è un sistema lettori-scrittori che favorisce i lettori.

La soluzione sembra possedere due caratteristiche ridondanti (Problema 6.10). Possiamo soffermare l'attenzione su due punti: il primo è che vengono utilizzati due semafori, *reading* e *writing*, sebbene solo una risorsa condivisa debba essere controllata. Secondo, ogni lettore esegue un'operazione *wait*(*reading*) sebbene l'operazione sia chiaramente ridondante quando altri lettori sono già impegnati nella lettura. Tuttavia, entrambe le caratteristiche sono necessarie per implementare un sistema lettori-scrittori che favorisce gli scrittori (Problema 6.11).

#### 6.9.4 Implementazione dei semafori

La [Figura 6.31](#) mostra uno schema per implementare i semafori: viene definito un tipo semaforo e sono presenti campi per il valore del semaforo, una lista utilizzata per memorizzare gli "id" dei processi bloccati sul semaforo ed una variabile di lock usata per garantire l'indivisibilità delle operazioni *wait* e *signal* sul semaforo. Le operazioni *wait* e *signal* sul semaforo sono implementate come procedure che ricevono come parametro una variabile di tipo semaforo. Un programma concorrente dichiara i semafori come variabili di tipo semaforo ed i suoi processi richiamano le procedure *wait* e *signal* per utilizzarli.

---

Dichiarazione del tipo semaforo

```
type
  semaforo = record
    value : integer;  { valore del semaforo }
    list : . . .       { lista dei processi bloccati }
    lock : boolean;   { variabile di lock per le operazioni su questo semaforo }
  end;
```

Procedure per implementare le operazioni wait e signal

```
procedura wait (sem)
begin
  Close_lock (sem.lock);
  if sem.value > 0
    then
      sem.value := sem.value-1;
      Open_lock (sem.lock);
    else
      Aggiunge l'id del processo alla lista dei processi bloccati su sem;
      block_me (sem.lock);
  end;
  procedura signal (sem)
begin
  Close_lock (sem.lock);
  if qualche processo è bloccato su sem
    then
      proc_id := id di un processo bloccato su sem;
      activate (proc_id);
    else
      sem.value := sem.value + 1;
      Open_lock (sem.lock);
  end;
```

---

**Figura 6.31** Uno schema di implementazione delle operazioni *wait* e *signal* su un semaforo.

Per evitare race condition durante l'accesso al valore del semaforo, le procedure *wait* e *signal* invocano prima la funzione *Close\_lock* per impostare la variabile di lock *sem.lock*. *Close\_lock* utilizza un'istruzione indivisibile e l'attesa attiva; tuttavia, le attese attive sono brevi poiché le operazioni *wait* e *signal* sono brevi. Le procedure invocano la funzione *Open\_lock* per reimpostare il lock al termine della loro esecuzione. Si ricordi dal Paragrafo 6.5.1 che un'attesa attiva può causare l'inversione delle priorità in un SO che utilizza uno scheduling basato su priorità; ipotizziamo che il protocollo di ereditarietà della priorità (priority inheritance protocol, PIP) venga utilizzato per evitare questo problema. In un sistema time-sharing, un'attesa attiva può causare ritardi nella sincronizzazione, ma non causa altri problemi.

La procedura *wait* controlla se il valore di *sem* è  $> 0$ . In caso affermativo, ne decremente il valore e ritorna. Se il valore è 0, la procedura *wait* aggiunge l'id del processo alla lista dei processi bloccati su *sem* ed esegue la chiamata di sistema *block\_me* con la variabile di lock come parametro. Tale chiamata blocca il processo che ha effettuato la chiamata *wait* ed inoltre rilascia il lock passatogli come parametro. Va notato che la procedura *wait* non avrebbe potuto effettuare queste azioni da sola: se rilasciasse il lock prima di effettuare un chiamata *block\_me* si potrebbero verificare delle race condition, mentre si creerebbe una situazione di deadlock se richiamasse la *block\_me* prima di rilasciare il lock!

La procedura *signal* controlla se qualche processo è bloccato su *sem*. In caso affermativo, seleziona uno di questi processi e lo riattiva eseguendo la chiamata di sistema *activate*. Se non ci sono processi in attesa su *sem*, incrementa il valore di *sem* di una unità. È conveniente mantenere la lista dei processi bloccati come una coda ed attivare il primo processo bloccato con un'operazione di *signal*. In questo modo, l'implementazione del semaforo rispetterebbe anche la proprietà dell'attesa limitata. Tuttavia, la semantica dell'operazione *signal* non specifica l'ordine in cui i processi dovrebbero essere riattivati, per cui un'implementazione potrebbe scegliere un qualsiasi

ordine.

L'operazione *wait* ha un tasso di fault molto basso in molti sistemi che utilizzano i semafori, ovvero i processi che effettuano le operazioni di *wait* raramente si bloccano. Questa caratteristica viene sfruttata in alcuni metodi di implementazione dei semafori per ridurre l'overhead. Nel seguito verranno descritti tre metodi di implementazione dei semafori ed esamineremo le loro implicazioni in termini di overhead. Si ricordi che il termine *processo* viene utilizzato per indicare sia i processi che i thread.

### **Implementazione a livello kernel**

Il kernel implementa le procedure *wait* e *signal* di [Figura 6.31](#). Tutti i processi in un sistema possono condividere un semaforo di livello kernel. Tuttavia, ogni operazione *wait* ed ogni operazione *signal* corrisponde a una system call; questo comporta un alto overhead per l'utilizzo dei semafori. In un SO monoprocesso con un kernel non prelazionabile, non sarebbe necessario utilizzare una variabile di lock per evitare le race condition, per cui l'overhead dovuto alle operazioni *Close\_lock* e *Open\_lock* può essere eliminato.

### **Implementazione a livello utente**

Le operazioni *wait* e *signal* sono codificate come procedure di libreria collegate a un'applicazione, per cui i processi dell'applicazione possono condividere i semafori di livello utente.

Le chiamate *block\_me* e *activate* sono di fatto chiamate di funzioni di libreria, che gestiscono in proprio il blocco e la riattivazione dei processi finché possibile ed effettuano chiamate di sistema solo quando necessitano dell'assistenza del kernel. Questo metodo di implementazione si adatterebbe ai thread di livello utente poiché la libreria dei thread fornirebbe già il blocco, l'attivazione e la schedulazione dei thread. La libreria dei thread effettuerebbe una chiamata di sistema *block\_me* solo quando tutti i thread di un processo sono bloccati.

### **Implementazione ibrida**

Le operazioni *wait* e *signal* sono anche in questo caso codificate come procedure di libreria ed i processi di un'applicazione possono condividere i semafori ibridi. *block\_me* e *activate* sono chiamate di sistema fornite dal kernel e le operazioni *wait* e *signal* eseguono queste chiamate solo quando i processi sono bloccati o riattivati. In conseguenza del basso tasso di fallimento dell'operazione *wait*, queste system call sarebbero effettuate raramente, per cui un'implementazione ibrida dei semafori genererebbe un overhead inferiore rispetto a un'implementazione a livello kernel.

## **6.10 Un esempio di problema di sincronizzazione**

Si consideri il problema della mensa universitaria e la fase di consegna dei vassoi da parte di  $N$  utenti dopo aver effettuato il pranzo. Vi sono  $M$  contenitori di vassoi a disposizione, ognuno con  $K$  vani per i singoli vassoi. Alcuni utenti ripongono anche una bottiglia d'acqua che occupa, insieme al vassoio, 2 vani nel contenitore di vassoi. Periodicamente, un addetto svuota 1 contenitore di vassoi alla volta, scelto tra quelli in cui non ci sono più vani liberi, e lava i piatti. Si assuma, per semplicità, che le operazioni di riporre vassoi e liberare un contenitore di vassoi siano atomiche. Formuliamo nel seguito una soluzione che ottimizza l'uso e l'accesso alle risorse, usando semafori e processi/thread. Viene proposta inoltre una implementazione della soluzione nel linguaggio di programmazione C con l'ausilio della libreria pthread POSIX.

### **6.10.1 Soluzione**

Si fanno le seguenti ipotesi:

- inizialmente tutti i contenitori sono vuoti;
- ogni utente accede al primo degli  $M$  contenitori che:
  - non è già occupato da un altro utente;
  - non è pieno in attesa di essere svuotato;
  - non è occupato dall'addetto;
- l'addetto ogni  $time$  secondi esegue le seguenti operazioni:

- verifica che vi siano contenitori pieni; se non ce ne sono ritorna a lavare i piatti;
- svuota solo il primo contenitore vuoto che incontra tra gli  $M$ , iniziando la ricerca dal contenitore successivo all'ultimo contenitore svuotato, dopo di che ritorna a lavare i piatti;
- accede al contenitore che svuota in modo esclusivo;
- non controlla se un contenitore, attualmente occupato da un utente, è diventato pieno dopo il rilascio da parte di quest'ultimo;
- l'utente che accede al contenitore  $i$  ha accesso esclusivo a esso sia rispetto agli altri utenti sia rispetto all'addetto.

Le strutture dati utilizzate sono:

- $residuo[M]$ : array globale condiviso inizializzato a  $K$ . Tiene traccia dei vani rimasti liberi per ogni contenitore. L'accesso a esso avviene in mutua esclusione e ogni suo elemento è:
  - decrementato di 1 o 2 unità dall'utente che vi accede;
  - posto uguale a  $K$  dall'addetto che lo trova vuoto;
- $contenitori[M]$ : array di semafori inizializzati a 1, utilizzati per consentire l'accesso esclusivo a un suo elemento da parte degli utenti e per verificare se un contenitore è pieno e quindi deve essere svuotato dall'addetto;
- $pieni$ : semaforo inizializzato a 0. Tiene traccia dei contenitori pieni;
- $vuoti$ : semaforo inizializzato a  $M$ . Tiene traccia dei contenitori vuoti;
- $accedi$ : mutex che regola la sincronizzazione tra utente e addetto. Consente all'addetto di distinguere se l'accesso a un contenitore gli è stato negato perché è pieno o perché è occupato da un utente. Nel primo caso svuota il contenitore, mentre nel secondo analizza il contenitore successivo.

## 6.10.2 Il codice C

```
[obeytabs=true,tabsize=4,showtabs=true]
/*variabili locali*/
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>

#define N 50
#define M 5
#define K 7

sem_t pieni, vuoti, contenitori[M];
pthread_mutex_t accedi;

int residuo[M];

void utente();
void addetto();

/*MAIN*/
```

```

int main()
{
    pthread_attr_t attr;
    pid_t putenti[N], paddetto;
    int i;

    sem_init(&pieni, 1, 0);
    sem_init(&vuoti, 1, M);

    pthread_mutexattr_init(&attr, PTHREAD_PSOIX_SHARED);
    pthread_mutex_init(&accedi, &attr);

    for(i=0; i<M; i++)
        sem_init(&accedi, &attr);

    for(i=0; i<N; i++)
        if( putenti[i] = fork() == 0)
        {
            utente();
            exit(0);
        }
    if( addetto == fork() ) > 0
    {
        addetto();
        exit(0);
    }
    for( i=0; i < N; i++)
        wait(putenti[i], NULL);
    exit(0);
}

/*PROCEDURA UTENTE*/
void utente()
{
    int i, riponi;

    riponi = (rand() % 1) + 1;
    while(riponi)
    {
        sem_wait(&vuoti);

        if( sem_trywait(&contenitore[i]) == 0)
        {
            pthread_mutex_lock(&accedi);
            rediduo[i]--;
            riponi--;
            if(residuo[i] > 0)
            {
                sem_post(&contenitori[i]);
                sem_post(&vuoti);
            }
        }
        else
            sem_post(&pieni);
    }
}

```

```

        pthread_mutex_unlock(&accedi);
        i=M;
    }
}
/* PROCEDURA ADDETTO */
void addetto()
{
    int p, i, time;

    time= rand();
    p= M-1;

    while(1)
    {
        sleep(time);

        if( sem_trywait(&pieni) == -1)
            continue;

        for( i = 0; i < M; i++)
        {
            p = (p+1)% M
            if ( sem_trywait(&contenitori[p]) == -1)
            {
                if(pthread_mutex_trylock(&accedi) == 0)
                {
                    residuo[p] = K;
                    sem_post(&contenitro[i]);
                    sem_post(&vuoti);
                    pthread_mutex_unlock(&accedi);
                    i = M;
                }
            }
            else
                sem_post(&continue[p]);
        }
    }
}

```

## 6.11 Monitor

Si ricordi dal Paragrafo 6.5.3 che un costrutto di programmazione concorrente fornisce caratteristiche di astrazione dei dati e encapsulamento adatte alla costruzione di programmi concorrenti. Definiamo un altro tipo di “monitor” che contiene le dichiarazioni dei dati condivisi. Inoltre può contenere le dichiarazioni di speciali dati di sincronizzazione chiamati *variabili di condizione* su cui possono essere effettuate solo le operazioni built-in **wait** e **signal**; queste operazioni forniscono modi convenienti per la segnalazione nella sincronizzazione dei processi. Le procedure del tipo monitor codificano le operazioni per manipolare i dati condivisi ed effettuare la sincronizzazione dei processi attraverso le variabili di condizione. In questo modo, il tipo monitor fornisce due delle tre componenti che costituiscono un sistema concorrente (Paragrafo 6.6).

Un sistema concorrente è impostato come segue: un programma concorrente ha un tipo monitor. Il programma crea un oggetto di tipo monitor durante l'esecuzione. Ci riferiamo all'oggetto come una variabile monitor, o semplicemente come *monitor*. Il monitor contiene una copia locale dei dati condivisi e di sincronizzazione dichiarati nel tipo monitor. Le procedure definite nel tipo monitor diventano operazioni del monitor ed

operano sui dati locali. Il programma concorrente crea i processi mediante chiamate di sistema. Questi processi invocano le operazioni del monitor per effettuare la condivisione dei dati ed il controllo della sincronizzazione; si bloccano o si attivano quando le operazioni del monitor eseguono le funzioni **wait** o **signal**.

Le caratteristiche di astrazione dei dati e di encapsulamento del monitor supportano la sincronizzazione come segue: solo le operazioni di un monitor possono accedere ai suoi dati condivisi e di sincronizzazione. Per evitare race condition, il compilatore del linguaggio di programmazione implementa la mutua esclusione sulle operazioni di un monitor assicurando che al più un processo possa eseguire un'operazione del monitor in ogni istante di tempo. Le chiamate delle operazioni sono servite in ordine FIFO per soddisfare la proprietà di attesa limitata.

### **Variabili di condizione**

Una *condizione* è una situazione di interesse in un monitor. Una *variabile di condizione*, che è semplicemente una variabile con l'attributo **condizione**, è associata a una condizione nel monitor. Solo le operazioni built-in **wait** e **signal** possono essere effettuate su una variabile di condizione. Il monitor associa una coda di processi a ogni variabile di condizione. Se un'operazione del monitor invocata da un processo esegue un'operazione *wait* sulla variabile di condizione, il monitor blocca il processo, inserisce il suo id nella coda dei processi associata alla variabile di condizione e schedula uno dei processi, se presenti, in attesa di iniziare o proseguire l'esecuzione di un'operazione del monitor. Se un'operazione del monitor esegue un'operazione *signal* su una variabile di condizione, il monitor attiva il primo processo nella coda associata alla variabile di condizione. Quando schedulato nuovamente, questo processo riprende l'esecuzione dell'operazione del monitor nella quale si era bloccato. L'operazione **signal** non ha effetto se la coda dei processi associata con la variabile di condizione è vuota nel momento in cui viene segnalata la condizione. L'implementazione di un monitor mantiene diverse code di processi, una per ogni variabile di condizione ed una per i processi in attesa di eseguire le operazioni del monitor. Per assicurare che i processi non restino bloccati durante l'esecuzione di un'operazione, il monitor favorisce i processi che sono stati attivati dalle operazioni **signal** rispetto a quelli che vogliono iniziare l'esecuzione delle operazioni del monitor.

L'esempio seguente descrive l'uso di un monitor per implementare un semaforo binario. Discuteremo un interessante aspetto dell'implementazione dopo l'esempio.

### **Esempio 6.6 - Implementazione di un semaforo binario mediante monitor**

La parte superiore della [Figura 6.32](#) mostra un tipo monitor *Sem\_Mon\_type* che implementa un semaforo binario e la parte inferiore mostra tre processi che utilizzano una variabile monitor *binary\_sem*. Si ricordi dal Paragrafo 6.9.1 che un semaforo binario assume solo i valori 0 e 1 ed è utilizzato per implementare una sezione critica. La variabile booleana *busy* è utilizzata per indicare se qualche processo al momento sta utilizzando la sezione critica. In questo modo, i suoi valori *vero* e *falso* corrispondono, rispettivamente, ai valori 0 e 1 del semaforo binario. La variabile di condizione *non\_busy* corrisponde alla condizione in cui la sezione critica non è occupata; viene utilizzata per bloccare i processi che tentano di accedere a una sezione critica mentre *busy = true*. Le procedure *sem\_wait* e *sem\_signal* implementano le operazioni *wait* e *signal* sul semaforo binario, *binary\_sem* è una variabile monitor. La parte di inizializzazione del tipo monitor, che contiene le istruzioni *busy = false*, è invocata quando viene creato *binary\_sem*. Dunque la variabile *busy* di *binary\_sem* è inizializzata a *false*.

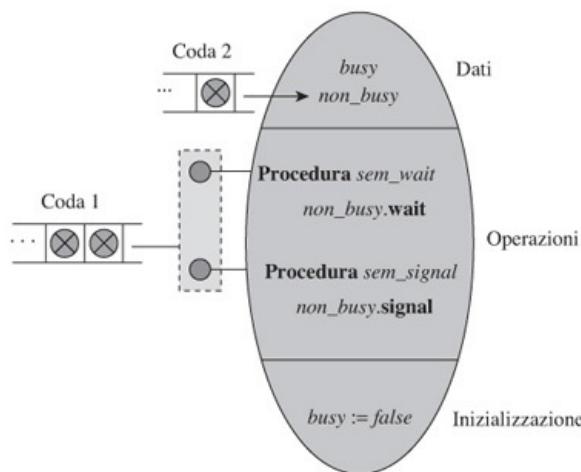
```

type Sem_Mon_type = monitor
  var
    busy : boolean;
    non_busy : condition;
  procedura sem_wait;
  begin
    if busy = true then non_busy.wait;
    busy := true;
  end;
  procedura sem_signal;
  begin
    busy := false;
    non_busy.signal;
  end;
  begin { inizializzazione }
    busy := false;
  end;
var binary_sem : Sem_Mon_type;
begin
  Parbegin
  repeat           repeat           repeat
    binary_sem.sem_wait;   binary_sem.sem_wait;   binary_sem.sem_wait;
    { Sezione critica }   { Sezione critica }   { Sezione critica }
    binary_sem.sem_signal; binary_sem.sem_signal; binary_sem.sem_signal;
    { Parte restante      { Parte restante      { Parte restante
      del ciclo }         del ciclo }         del ciclo }
  forever;         forever;         forever;
  Parend;
end.
  
```

Processo P<sub>1</sub>      Processo P<sub>2</sub>      Processo P<sub>3</sub>

**Figura 6.32** Un monitor per implementare un semaforo binario.

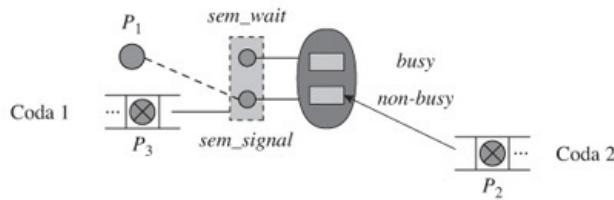
La Figura 6.33 mostra il monitor *Sem\_Mon\_type*. Il monitor mantiene due code di processi. La coda 1 contiene i processi in attesa di eseguire l'operazione del monitor *sem\_wait* o *sem\_signal*, mentre la coda 2 contiene i processi in attesa dell'esecuzione di una istruzione *non\_busy.signal*.



**Figura 6.33** Un monitor per implementare un semaforo binario.

Sia  $P_1$  il primo processo a eseguire *binary\_sem.sem\_wait*. Poiché *busy* è *false*, cambia *busy* a *true* ed entra nella sua sezione critica. Se  $P_2$  esegue *binary\_sem.sem\_wait* mentre  $P_1$  si trova ancora all'interno della sua sezione critica, si bloccherà sull'istruzione *non\_busy.wait*. Aspetterà nella coda 2. Si consideri che  $P_1$  inizi a eseguire *binary\_sem.sem\_signal* e che  $P_3$  cerchi di eseguire

`binary_sem.sem_wait` prima che  $P_1$  termini l'esecuzione di `binary_sem.sem_signal`. A causa della mutua esclusione sulle operazioni del monitor,  $P_1$  resterà bloccato ed inserito nella coda associata con l'accesso al monitor, ovvero la coda 1. La Figura 6.34 mostra un'istantanea del sistema a tale istante. Quando il processo  $P_1$  esegue l'istruzione `non_busy.signal` ed esce dal monitor,  $P_2$  verrà attivato prima di  $P_3$  poiché le code associate con le variabili di condizione hanno la priorità rispetto alla coda associata con l'ingresso al monitor. Il processo  $P_3$  comincerà l'esecuzione `binary_sem.sem_wait` solo quando il processo  $P_2$  completa l'esecuzione di `binary_sem.sem_wait`, esce dal monitor ed entra nella sua sezione critica.  $P_3$  si bloccherà sulla condizione `non_busy`. Sarà attivato quando  $P_2$  esegue l'operazione `binary_sem.sem_signal`.



**Figura 6.34** Un'istantanea del sistema dell'Esempio 6.6.

Se la procedura `sem_signal` dell'Esempio 6.6 contenesse delle istruzioni dopo l'istruzione **signal**, si verificherebbe un interessante problema di sincronizzazione nel momento in cui  $P_1$  esegue `binary_sem.sem_signal` e l'istruzione `non_busy.signal`. L'istruzione **signal** attiverebbe il processo  $P_2$ , che riprenderebbe la propria esecuzione di `binary_sem.sem_wait`. Nello stesso momento, il processo  $P_1$  continuerebbe la propria esecuzione di `binary_sem.sem_signal` eseguendo le istruzioni successive all'istruzione `non_busy.signal`. Dal momento che le operazioni nel monitor sono effettuate in maniera mutuamente esclusiva, solo uno dei due processi può essere eseguito e l'altro deve attendere. Quale dei due dovrebbe essere selezionato per l'esecuzione?

Selezionare per l'esecuzione il processo  $P_2$  ritarderebbe il processo  $P_1$  che aveva inviato il segnale, cosa che appare non equa. Selezionare  $P_1$  implicherebbe una non reale attivazione di  $P_2$  finché  $P_1$  non lascia il monitor. Hoare (1974) propose la prima alternativa. Brinch Hansen (1973) propose che un'istruzione **signal** dovrebbe essere l'ultima istruzione della procedura di un monitor, in modo da consentire al processo che esegue la **signal** di uscire dal monitor immediatamente e al processo attivato dalla **signal** di essere schedulato. Seguiremo questa convenzione nei nostri esempi.

#### Esempio 6.7 - Produttori-consumatori mediante monitor

La Figura 6.35 mostra una soluzione al problema produttori-consumatori mediante l'utilizzo dei monitor. Segue lo stesso approccio della soluzione di Figura 6.28 che utilizzava i semafori. La parte superiore di Figura 6.35 mostra un tipo monitor *Bounded\_buffer\_type*. La variabile  *pieno* è un intero che indica il numero di buffer pieni. Nella procedura *produce*, un produttore esegue una *buffer\_vuoto.wait* se  *pieno* =  $n$ . Sarebbe attivato solo se esistesse almeno un buffer vuoto nel pool. Analogamente, il consumatore esegue una *buffer\_pieno.wait* se  *pieno* = 0. I consumatori in attesa ed i produttori sono attivati dalle istruzioni *buff\_pieno.signal* e *buff\_vuoto.signal* rispettivamente nelle procedure *produce* e *consume*.

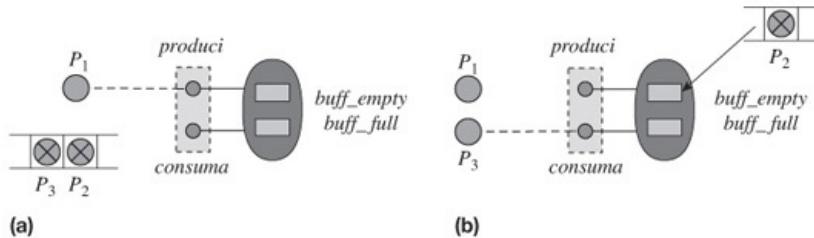
```

type Bounded_buffer_type = monitor
const
  n = . . .;                                { Numeri di buffer }
type
  item = . . .;
var
  buffer : array [0..n-1] of item;
  full, prod_ptr, cons_ptr : integer;
  buff_full : condition;
  buff_empty : condition;
procedura produci (produced_info : item);
begin
  if full = n then buff_empty.wait;
  buffer [prod_ptr] := produced_info;          { i.e. produci }
  prod_ptr := prod_ptr + 1 mod n;
  full := full + 1;
  buff_full.signal;
end;
procedura consuma (for_consumption : item);
begin
  if full = 0 then buff_full.wait;
  for_consumption := buffer[cons_ptr];          { i.e. consuma }
  cons_ptr := cons_ptr + 1 mod n;
  full := full - 1;
  buff_empty.signal;
end;
begin { inizializzazione }
  full := 0;
  prod_ptr := 0;
  cons_ptr := 0;
end;
begin
  var B_buf: Bounded_buffer_type;
  Parbegin
    var info : item;          var info : item;          var area : item;
    repeat          repeat          repeat
      info := . . .           info := . . .           B_buf.consume (area);
      B_buf.produce (info);  B_buf.produce (info);  { Area di consumo }
      { Parte restante       { Parte restante       { Parte restante
        del ciclo }          del ciclo }          del ciclo }
      all'infinito;          all'infinito;          all'infinito;
    Parend;
  end.
  Produttore P1          Produttore P2          Consumatore P3

```

**Figura 6.35** Produttori-consomatori mediante monitor.

La parte inferiore della [Figura 6.35](#) mostra un sistema composto di due processi produttori  $P_1$  e  $P_2$  ed un processo consumatore  $P_3$ . Il funzionamento di un sistema con buffer singolo, ovvero con  $n = 1$ , in [Figura 6.35](#), può essere descritto come mostrato in [Figura 6.36](#). Si consideri che i processi  $P_1$  e  $P_2$  cercano di produrre ed il processo  $P_3$  di consumare, tutti allo stesso tempo. Si consideri che il processo  $P_1$  entri nella procedura *produce*, superi l'istruzione **wait** e inizi a produrre, mentre i processi  $P_2$  e  $P_3$  sono bloccati all'ingresso del monitor (vedi la Parte (a) dell'istantanea).  $P_1$  esegue **buff.pieno.signal** ed esce. Il processo  $P_2$  viene ora attivato. Tuttavia, si blocca nuovamente su **buff.vuoto:wait** poiché *pieno* = 1. Il processo  $P_3$  viene attivato quando  $P_2$  si blocca ed inizia a consumare [[Figura 6.36\(b\)](#)]. Il processo  $P_2$  sarà attivato quando  $P_3$  termina dopo aver consumato.



**Figura 6.36** Instantanee del monitor dell'Esempio 6.7 con un singolo buffer.

### 6.11.1 Monitor in Java

In Java si crea una classe monitor quando l'attributo `synchronized` viene associato con uno o più metodi della classe; inoltre un oggetto di questa classe è a sua volta un monitor. La Java virtual machine garantisce la mutua esclusione dei metodi `synchronized` del monitor come segue: quando un thread richiama un metodo `synchronized` di un oggetto, la Java virtual machine controlla se l'oggetto attualmente è occupato (locked). Se è libero (unlocked), viene impostato il lock ed al thread è consentita l'esecuzione del metodo; in caso contrario, il thread deve attendere finché l'oggetto viene sbloccato. Quando un thread termina l'esecuzione di un metodo `synchronized`, l'oggetto viene sbloccato e, se presente, viene attivato un thread in attesa.

Ogni monitor contiene una singola variabile di condizione senza nome. Un thread attende sulla variabile di condizione eseguendo la chiamata `wait()`. La chiamata `notify()` è analoga alla operazione **signal** descritta nel Paragrafo 6.11. Riattiva uno dei thread in attesa sulla variabile di condizione, se presente. La Java virtual machine non implementa l'ordine FIFO per le chiamate `wait` e `notify`. Per questo motivo, la `wait` e la `notify` non soddisfano la proprietà di attesa limitata. La chiamata `notifyAll()` attiva tutti i thread in attesa della variabile di condizione.

L'utilizzo di una sola variabile di condizione in un monitor può portare ad attese attive in un'applicazione. Si consideri come esempio il problema lettori-scrittori. Quando uno scrittore è attivo, tutti i lettori che vogliono leggere e tutti gli scrittori che vogliono scrivere devono attendere sulla variabile di condizione. Quando lo scrittore termina la scrittura, dovrebbe utilizzare una chiamata `notifyAll()` per attivare tutti i thread in attesa. Se i lettori hanno precedenza, tutti i thread scrittori dovranno effettuare una nuova chiamata `wait()`. Se si dà precedenza agli scrittori, tutti i thread lettori ed alcuni thread scrittori dovranno eseguire una nuova `wait()`. In questo modo, un thread lettore o scrittore può essere attivato molte volte prima di avere l'opportunità di eseguire una lettura o una scrittura. Un sistema produttori-consumatori con molti thread produttori e consumatori soffrirebbe analogamente di attese attive.

## 6.12 Casi di studio relativi alla sincronizzazione dei processi

### 6.12.1 Sincronizzazione dei thread POSIX

Come menzionato nel Paragrafo 5.3.1, nel caso di thread POSIX sono a disposizione i mutex per la mutua esclusione e le variabili di condizione per il controllo della sincronizzazione tra processi. Un mutex è in realtà un semaforo binario. Un SO può implementare i thread POSIX come thread kernel o thread utente. Di conseguenza, nel primo caso i mutex sarebbero implementati a livello kernel oppure con un'implementazione ibrida descritta nel Paragrafo 6.9.4; nel secondo caso sarebbero implementati a livello utente. In modo analogo, le variabili di condizione sono implementate a livello kernel, a livello utente o con un schema ibrido.

### 6.12.2 Sincronizzazione dei processi in Unix

Unix system V implementa i semafori a livello del kernel. Il nome di un semaforo è detto `key`. La `key` è associata a un array di semafori e l'accesso ai singoli semafori viene effettuato mediante indice. I processi condividono lo stesso semaforo utilizzando la stessa `key`. Un processo accede all'utilizzo di un semaforo effettuando una system call

*semget* passando key come parametro. Se esiste già un semaforo con la key specificata, il kernel consente l'accesso al kernel che ha eseguito la chiamata *semget*; in caso contrario, crea un nuovo semaforo, gli assegna la key e lo rende accessibile al processo.

Le operazioni *wait* e *signal* sul semaforo vengono effettuate utilizzando la chiamata di sistema *semop*, che si avvale di due parametri: una key, ovvero il nome del semaforo ed una lista di coppie (*subscript, op*) in cui *subscript* identifica un semaforo dell'array di semafori e *op* identifica l'operazione *wait* o *signal* da eseguire. L'intera lista di operazioni viene eseguita in modo atomico; o vengono eseguite tutte le operazioni ed il processo è libero di continuare l'esecuzione, o non viene effettuata nessuna operazione ed il processo si blocca. Un processo bloccato viene attivato solo quando tutte le operazioni specificate con la *semop* possono essere eseguite.

Le semantiche di *semop* possono essere usate per prevenire i deadlock. Si consideri il seguente esempio: i semafori *sem<sub>1</sub>* e *sem<sub>2</sub>* sono associati, rispettivamente, con le risorse *R<sub>1</sub>* e *R<sub>2</sub>*. Un processo esegue una *wait(sem<sub>i</sub>)* prima di utilizzare una risorsa *R<sub>i</sub>* ed una *signal(sem<sub>i</sub>)* al termine dell'utilizzo. Se entrambi i processi *P<sub>1</sub>* e *P<sub>2</sub>* richiedono entrambe le risorse simultaneamente, è possibile che *P<sub>1</sub>* ottenga l'accesso a *R<sub>1</sub>* ma si blocchi sulla *wait(sem<sub>2</sub>)* mentre il processo *P<sub>2</sub>* potrebbe ottenere l'accesso a *R<sub>2</sub>* ma bloccarsi su *wait(sem<sub>1</sub>)*. Questa è considerata una situazione di deadlock poiché entrambi i processi aspetterebbero l'un l'altro indefinitamente. Questo deadlock potrebbe essere evitato se i processi eseguissero entrambe le operazioni di *wait* utilizzando una singola *semop*, dal momento che a un processo potrebbero essere assegnate entrambe le risorse o nessuna di esse. Questa modalità di esecuzione ricorda l'approccio alla prevenzione dei deadlock (Paragrafo 8.5.1) in cui vengono allocate contemporaneamente tutte le risorse richieste.

Unix SVR4 fornisce una caratteristica interessante per rendere l'utilizzo dei semafori più attendibile. Il sistema tiene traccia di tutte le operazioni eseguite da un processo su ogni semaforo utilizzato ed effettua una *undo* (cancellazione) di queste operazioni quando il processo termina. Questa operazione consente di prevenire problemi in un'applicazione concorrente dovuti al non corretto funzionamento di qualche processo. Per esempio, se un processo *P<sub>i</sub>* esegue su un semaforo *sem<sub>i</sub>* più operazioni *wait* rispetto alle operazioni *signal* e poi termina, potrebbe causare attese indefinite di altri processi nell'applicazione. Effettuare un'operazione *undo* di tutte le operazioni *wait* e *signal* eseguite da *P<sub>i</sub>* potrebbe prevenire questi problemi. Per effettuare le operazioni di *undo* in maniera efficiente, il kernel mantiene un conteggio cumulativo dei cambiamenti dei valori di un semaforo in conseguenza delle operazioni eseguite da un processo su quel semaforo e le sottrae dal valore del semaforo quando il processo termina. Se un processo *P<sub>i</sub>* effettua più operazioni *wait* che operazioni *signal* su un semaforo *sem<sub>i</sub>*, il conteggio cumulativo per questo semaforo sarà negativo. Sottrarre questo conteggio annullerebbe l'effetto di *P<sub>i</sub>* su *sem<sub>i</sub>*. Il conteggio cumulativo del processo *P<sub>i</sub>* sarebbe 0 se avesse eseguito un ugual numero di operazioni *wait* e *signal* su *sem<sub>i</sub>*. In questo modo l'operazione *undo* non interferisce con il normale funzionamento dei processi che utilizzano i semafori.

Unix 4.4BSD inserisce un semaforo nelle aree di memoria condivise da un insieme di processi e fornisce una implementazione ibrida dei semafori secondo le linee guida discusse nel Paragrafo 6.9.4. In questo modo, evita l'esecuzione di chiamate di sistema nei casi in cui un'operazione *wait* non porti al blocco di un processo e un'operazione *signal* non porti all'attivazione di un processo, ottenendo una sincronizzazione più veloce.

### 6.12.3 Sincronizzazione dei processi in Linux

Linux fornisce implementazioni dei semafori nello stile Unix (Paragrafo 6.12.1) utilizzabili dagli utenti. Inoltre, fornisce due tipi di semafori utilizzabili dal kernel: un semaforo convenzionale ed un semaforo lettore-scrittore. Il semaforo convenzionale è implementato con uno schema a livello kernel che risulta più efficiente dello schema di livello kernel discusso nel Paragrafo 6.9.4. Utilizza una struttura dati che contiene il valore del semaforo, una flag che indica se qualche processo è bloccato su di esso e la lista di tali processi. Diversamente dallo schema del Paragrafo 6.9.4, non viene utilizzato un lock per evitare le race condition sul valore del semaforo; invece, le operazioni *wait* e *signal* utilizzano istruzioni trasparenti per decrementare ed incrementare il valore del semaforo. Queste operazioni bloccano la lista dei processi bloccati solo se si verificano che vi sono dei processi da aggiungere o rimuovere dalla lista - l'operazione *wait* blocca la lista solo se il processo che ha eseguito l'operazione *wait* deve essere bloccato, mentre

l'operazione *signal* blocca la lista solo se la flag del semaforo indica che la lista non è vuota.

Il semaforo lettore-scrittore fornisce funzionalità che possono essere utilizzate per implementare il problema dei lettori-scrittori del Paragrafo 6.9.3 all'interno del kernel in modo che più processi possono leggere una struttura dati del kernel, ma solo un processo può aggiornarla ogni volta. La sua implementazione non favorisce né i lettori né gli scrittori, ma permette ai processi di accedere alle proprie sezioni critiche in ordine FIFO, fatta eccezione per il fatto che due lettori consecutivi possono leggere concorrentemente. Questo obiettivo è ottenuto semplicemente mantenendo una lista di processi in attesa di effettuare un'operazione di lettura o scrittura, in ordine cronologico.

I kernel precedenti a Linux 2.6 implementavano la mutua esclusione in kernel space mediante system call. Tuttavia, come detto nel Paragrafo 6.9.4, un'operazione *wait* ha un basso tasso di fallimento; ovvero, un processo raramente viene bloccato su una chiamata *wait*, per cui molte chiamate di sistema risultano non necessarie. Il kernel 2.6 fornisce un mutex veloce in user space chiamato *futex*. Un *futex* è un intero in un'area di memoria condivisa su cui possono essere effettuate solo alcune operazioni. L'operazione *wait* su *futex* effettua una system call solo quando un processo deve essere bloccato su un *futex* e l'operazione *signal* su un *futex* effettua una system call solo quando un processo deve essere attivato. L'operazione *wait* inoltre fornisce un parametro attraverso il quale un processo può indicare per quanto tempo è disposto a essere bloccato su una *wait*. Quando questo tempo è trascorso, l'operazione *wait* fallisce e ritorna con un codice di errore al processo che ha effettuato la chiamata.

#### 6.12.4 Sincronizzazione dei processi in Solaris

La sincronizzazione dei processi nel sistema operativo Sun Solaris contiene tre caratteristiche interessanti: semafori lettori-scrittori e mutex adattivi, una struttura dati chiamata turnstile (tornello) e l'utilizzo del priority inversion protocol. Il semaforo lettore-scrittore è analogo al semaforo lettore-scrittore in Linux. Un mutex adattivo è utile in un SO multiprocessore, per cui se ne discuterà nel [Capitolo 10](#); qui ne viene fatta solo una panoramica.

Si ricordi dal Paragrafo 5.4.3 che il kernel Solaris fornisce il parallelismo mediante i thread kernel. Quando un thread  $T_i$  esegue un'operazione *wait* su un semaforo attualmente in uso da un altro processo  $T_j$ , il kernel può o bloccare  $T_i$  o lasciarlo attivo durante l'attesa (spinning). L'approccio basato sul blocco del processo coinvolge l'overhead dovuto al blocco del thread  $T_i$ , la schedulazione di un altro processo e l'attivazione del thread  $T_i$  nel momento in cui  $T_j$  rilascia il semaforo. Lo spinning, d'altra parte, provoca l'overhead dell'attesa attiva finché  $T_j$  rilascia il semaforo. Se  $T_j$  viene eseguito su un'altra CPU, può rilasciare il semaforo prima che  $T_i$  o  $T_j$  venga prelazionato, per cui è meglio lasciare  $T_i$  attivo. Se  $T_j$  non è in esecuzione,  $T_i$  può rimanere attivo per molto tempo, per cui è meglio risparmiare tempo di CPU bloccandolo. Il mutex adattivo utilizza questo metodo.

Il kernel Solaris utilizza una struttura dati chiamata *turnstile* (tornello) per memorizzare le informazioni riguardanti i thread bloccati su un mutex o su un semaforo lettore-scrittore. Questa informazione è utilizzata sia per la sincronizzazione sia per l'eredità della priorità. Per minimizzare il numero di tornelli necessari a ogni istante, il kernel di Solaris 7 collega un tornello a ogni nuovo thread che crea. Le seguenti azioni vengono eseguite quando un thread kernel deve essere bloccato su un mutex: se non ci sono thread già bloccati sul mutex, il kernel rimuove il tornello dal thread, lo associa al mutex ed inserisce l'id del thread nel tornello. Se un tornello è già associato al mutex, ovvero se qualche altro processo è già bloccato su di esso, il kernel rimuove il tornello del thread e lo restituisce al pool di tornelli liberi e inserisce l'id del thread nel tornello che è già associato al mutex. Quando un thread rilascia un mutex o un semaforo lettore-scrittore, il kernel ottiene le informazioni relative ai thread bloccati sul mutex o sul semaforo lettore-scrittore e decide quale thread attivare. A questo punto collega un tornello dal pool dei tornelli liberi con il thread attivato. Un tornello è restituito al pool dei tornelli liberi quando l'ultimo thread al suo interno viene riattivato.

Il kernel Solaris utilizza il priority inheritance protocol per ridurre i ritardi di sincronizzazione. Si consideri un thread  $T_i$  bloccato su un semaforo poiché il thread  $T_j$  si trova in una sezione critica implementata mediante un semaforo. Il thread  $T_i$  potrebbe accusare un lungo ritardo di sincronizzazione se  $T_j$  non viene schedulato per un lungo

periodo, cosa che si verificherebbe se  $T_j$  avesse un priorità inferiore a  $T_i$ . Per ridurre il ritardo di sincronizzazione per  $T_i$ , il kernel aumenta la priorità di  $T_j$  fino a quella di  $T_i$  finché  $T_j$  non esce dalla sezione critica. Se molti processi si bloccano sul semaforo utilizzato da  $T_j$ , la priorità di  $T_j$  dovrebbe essere aumentata fino a raggiungere quella del processo a più alta priorità bloccato sul semaforo. Ciò viene implementato recuperando le priorità dei processi bloccati dal tornello associato al semaforo.

### 6.12.5 Sincronizzazione dei processi in Windows

Windows è un sistema operativo object-oriented, per cui processi, file ed eventi sono rappresentati da oggetti. Il kernel fornisce un'interfaccia uniforme per la sincronizzazione dei thread su differenti tipi di oggetti come segue: un *dispatcher di oggetti* è uno speciale tipo di oggetto che può trovarsi o nello stato *signaled* o nello stato *nonsignaled*. Un oggetto dispatcher è contenuto in ogni oggetto sul quale è possibile eseguire la sincronizzazione, per esempio un oggetto che rappresenta un processo, un file, un evento, un mutex o un semaforo. Ogni thread che volesse sincronizzarsi con un oggetto verrebbe messo nello stato *wait* se l'oggetto dispatcher contenuto nell'oggetto si trovasse nello stato *nonsignaled*. La [Tabella 6.4](#) descrive la semantica relativa a vari tipi di oggetti, che determina quando lo stato di un oggetto cambia e quale dei thread in attesa su di esso viene attivato quando l'oggetto riceve un segnale.

| Object                     | Stato nonsignaled                                      | Stato signaled                                     | Azione conseguente a una signal               |
|----------------------------|--|--|---|
| Processo                   | Non terminato  | L'ultimo thread termina                            | Attivare tutti i thread                       |
| Thread                     | Non terminato  | Il thread termina                                  | Attivare tutti i thread                       |
| File                       | Richiesta di I/O pendente                              | I/O completato                                     | Attivare tutti i thread                       |
| Input da console           | Input non fornito                                      | Input fornito                                      | Attivare un thread                            |
| Modifica al file           | Nessuna modifica                                       | Modifica notificata                                | Attivare un thread                            |
| Notifica di un evento      | Non ancora impostato                                   | Impostazione dell'evento eseguita                  | Attivare tutti i thread                       |
| Evento di sincronizzazione | Reset  | Impostazione dell'evento eseguita                  | Attivare un thread e resettare l'evento       |
| Semaforo                   | Successo della wait                                    | Rilasciato   | Attivare un thread                            |
| Mutex                      | Successo della wait                                    | Rilasciato   | Attivare un thread                            |
| Variabile di condizione    | Prima e dopo una chiamata <i>wake</i> o <i>wakeall</i> | Funzione <i>wake</i> o <i>wakeall</i> eseguita     | Attivare uno o tutti i thread                 |
| Timer                      | Reinizializzazione                                     | Impostare scadenza arrivata o intervallo trascorso | Come notifica e sincronizzazione degli eventi |

**Tabella 6.4** Oggetti di Windows utilizzati per la sincronizzazione.

Un oggetto thread entra nello stato signaled quando il thread termina, mentre un oggetto processo entra nello stato signaled quando tutti i thread nel processo terminano. In entrambi i casi, tutti i thread in attesa sull'oggetto sono attivati. L'oggetto file entra nello stato signaled quando un'operazione di I/O sul file viene completata. Se qualche thread è in attesa su di esso, tutti sono attivati e il suo stato di sincronizzazione viene riportato a nonsignaled. Se nessun thread è in attesa su di esso, un thread che dovesse trovarsi in attesa su di esso in futuro, effettuerebbe l'operazione wait e lo stato di sincronizzazione dell'oggetto sarebbe impostato a nonsignaled. L'oggetto input da console ha un comportamento analogo, fatta eccezione per il fatto che solo un thread in attesa viene attivato quando riceve un segnale. L'oggetto modifica del file riceve un segnale quando il sistema rileva modifiche nel file. Per gli altri aspetti ha un comportamento uguale all'oggetto file.

I thread usano gli oggetti eventi, semafori, mutex e variabili di condizione per la sincronizzazione. Inviano segnali a questi oggetti eseguendo funzioni di libreria che puntano alle chiamate di sistema appropriate. A un oggetto viene inviato un segnale nel

momento della system call *set event*. Se si tratta di un evento di notifica, tutti i thread in attesa su di esso sono attivati. Se è un evento di sincronizzazione, solo un thread viene attivato e l'evento viene resettato. L'oggetto timer è anche progettato per essere utilizzato nelle modalità di notifica e sincronizzazione. Il kernel modifica lo stato dell'oggetto a *signaled* quando scade un tempo specificato o è trascorso un determinato intervallo. Le sue azioni di segnalazione sono simili a quelle degli eventi di notifica e sincronizzazione.

L'oggetto semaforo implementa un semaforo contatore, che può essere utilizzato per controllare un insieme di risorse. Il numero di risorse viene specificato come valore iniziale del semaforo. Un contatore nell'oggetto semaforo indica quante di queste risorse sono attualmente disponibili per essere utilizzate dai thread. L'oggetto semaforo è nello stato nonsignaled quando il contatore è a 0, per cui ogni processo che effettua una wait su di esso sarebbe messo nello stato *waiting*. Quando un thread rilascia una risorsa, il kernel incrementa il numero di risorse disponibili, cosa che imposta il semaforo nello stato signaled. Di conseguenza, un thread in attesa su di esso sarebbe attivato. Un thread può specificare un intervallo di tempo in una chiamata wait per indicare quanto tempo è disposto ad attendere per un oggetto. Viene attivato prima della fine di questo periodo se l'oggetto riceve un segnale; in caso contrario, la richiesta relativa alla wait sarebbe rimossa alla fine dell'intervallo e il processo riattivato. Un mutex viene implementato come semaforo binario. Ad un oggetto mutex viene inviato un segnale quando un processo esegue la funzione di rilascio; il kernel riattiva uno dei thread in attesa su di esso.

Il kernel Windows fornisce un varietà di lock di sincronizzazione: uno spinlock, un lock speciale chiamato *coda di spinlock* per configurazioni multiprocessori (Paragrafo 10.6.3), i fast mutex ed i push lock, i quali, come i futex di Linux, evitano l'esecuzione di chiamate di sistema a meno che un thread non debba attendere su un oggetto di sincronizzazione. Windows Vista fornisce un lock lettore-scrittore.

## Riepilogo

La sincronizzazione dei processi sta per la *sincronizzazione per l'accesso ai dati*, che è utilizzata per aggiornare dati condivisi in maniera mutuamente esclusiva e per *controllo di sincronizzazione*, che è usato per assicurare che i processi effettuino le loro azioni nell'ordine desiderato. I problemi classici di sincronizzazione dei processi come produttori-consumatori, lettori-scrittori ed i filosofi a cena rappresentano importanti classi di problemi di sincronizzazione di processi. In questo capitolo abbiamo discusso le problematiche fondamentali che sorgono nella sincronizzazione dei processi ed il supporto per la sincronizzazione dei processi forniti dal computer, dal kernel e dai linguaggi di programmazione. Inoltre abbiamo analizzato i problemi classici di sincronizzazione e dimostrato l'uso di vari mezzi di sincronizzazione messi a disposizione dai linguaggi di programmazione e dai sistemi operativi nella loro implementazione. È discussa e riportata anche una soluzione a un problema di sincronizzazione in linguaggio C e libreria pthread POSIX.

Una *race condition* è una situazione in cui le azioni di processi concorrenti può avere conseguenze inaspettate, quali valori non corretti di dati condivisi o interazioni sbagliate tra processi. Una race condition sussiste quando processi concorrenti aggiornano dati condivisi in modo non coordinato. Le race condition vengono evitate mediante l'uso della *mutua esclusione*, la quale assicura che solo un processo aggiorni i dati condivisi a ogni istante di tempo. Una *sezione critica* sul dato condiviso *d* è una sezione di codice che accede a *d* in maniera mutuamente esclusiva. Una race condition può anche verificarsi nel controllo della sincronizzazione, quando i processi non possono attendere le azioni l'uno dell'altro come previsto. Dunque evitare le race condition è un obiettivo primario nella sincronizzazione dei processi.

Il computer fornisce *istruzioni trasparenti*, che accedono alle locazioni di memoria in maniera mutuamente esclusiva. Un processo può usare un' istruzione trasparente su una variabile di lock per implementare una sezione critica. Tuttavia, questo approccio soffre delle *busy wait* poiché un processo che non può entrare nella sezione critica continua a ciclare finché può farlo; dunque il kernel fornisce uno strumento per bloccare tale processo finché non gli viene consentito l'accesso alla sezione critica. I compilatori dei linguaggi di programmazione implementano le primitive ed i costrutti di sincronizzazione dei processi utilizzando questi strumenti. Un *semaforo* è una

primitiva che facilita il blocco e l'attivazione dei processi senza race condition. Un *monitor* è un costrutto che possiede due caratteristiche: implementa le operazioni sui dati condivisi come sezioni critiche sui dati e fornisce istruzioni per il controllo della sincronizzazione.

I sistemi operativi forniscono caratteristiche per un'implementazione efficiente della sincronizzazione dei processi; per esempio, Linux fornisce semafori lettori-scrittori, Solaris fornisce il priority inheritance protocol per evitare alcuni problemi relativi alle attese attive e Windows fornisce gli oggetti dispatcher.

## Domande

- 6.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. Un'applicazione può contenere una race condition solo se il sistema su cui è in esecuzione l'applicazione possiede più di una CPU.
  - b. Il controllo della sincronizzazione è necessario quando i processi *generate* e *analyze* di [Figura 1.6\(b\)](#) condividono la variabile *sample*.
  - c. Un processo può subire una starvation all'entrata di una sezione critica se l'implementazione della sezione critica non soddisfa la condizione di attesa limitata.
  - d. Un processo può subire una starvation all'entrata di una sezione critica se l'implementazione della sezione critica non soddisfa la condizione di progresso.
  - e. Una busy wait è inevitabile a meno che venga eseguita una system call per bloccare un processo.
  - f. Attese attive indefinite si possono verificare in un SO che utilizza uno scheduling basato su priorità, ma non possono verificarsi in un SO che utilizza uno scheduling round-robin.
  - g. L'Algoritmo 6.1 può essere usato per implementare un sistema produttore-consumatore con buffer singolo se il processo  $P_1$  è un produttore ed il processo  $P_2$  è un consumatore.
  - h. Quando viene utilizzata una variabile di lock, un'istruzione indivisibile non è necessaria per implementare una sezione critica.
  - i. In un sistema produttore-consumatore che consiste di molti processi produttori, molti processi consumatori e molti buffer nel pool di buffer, è possibile che molti processi consumatori producano e molti processi consumatori consumino allo stesso tempo.
  - j. In un sistema lettori-scrittori che favorisce gli scrittori, alcuni processi lettori che vogliono leggere i dati condivisi possono bloccarsi anche quando altri processi lettori stanno leggendo i dati condivisi.
  - k. Un deadlock non può verificarsi nel problema dei filosofi a cena se un filosofo può mangiare solo con una forchetta.
  - l. Una sezione critica implementata usando i semafori soddisfa la proprietà dell'attesa limitata solo se l'operazione *signal* attiva i processi in ordine FIFO.
  - m. Una race condition può verificarsi sulle forchette se la soluzione del problema dei filosofi a cena di [Figura 6.19](#) fosse modificata in modo che l'azione "prendi le forchette una alla volta" fosse rimossa dal ciclo **while** e messa nel ciclo **while** successivo.
- 6.2. Un semaforo è inizializzato a 1. Vengono effettuate dodici operazioni wait e sette operazioni signal. Qual è il numero di processi in attesa su questo semaforo?
  - a. 12, b. 7, c. 4, d. 5
- 6.3. Un semaforo binario è inizializzato a 1. Cinque operazioni wait sono effettuate sul semaforo consecutivamente, seguite da otto operazioni signal. Successivamente vengono eseguite altre cinque operazioni wait. Qual è il numero di processi in attesa su questo semaforo?
  - a. 1, b. 2, c. 4, d. 5
- 6.4. Dieci processi condividono una sezione critica implementata utilizzando un semaforo contatore chiamato  $x$ . Nove di questi processi usano il codice  $wait(x)$ ;  $\{\text{critical section}\} signal(x)$ . Tuttavia, un processo erroneamente utilizza il codice  $signal(x)$ ;  $\{\text{critical section}\} signal(x)$ . Qual è il massimo numero di processi che possono trovarsi nella sezione critica allo stesso momento?
  - a. 1, b. 2, c. 10, d. 3

- 6.5. In un sistema lettori-scrittori, un'operazione di lettura consuma 3 unità di tempo ed una operazione di scrittura consuma 5 unità di tempo. Al tempo  $t_i - 1$  non sono presenti né lettori né scrittori nel sistema. Al tempo  $t_i$ , arriva un lettore e 5 lettori e 1 scrittore arrivano al tempo  $t_i + 1$ . Se non arrivano più lettori o scrittori, quando lo scrittore terminerà la scrittura?
- $t_i + 8$ ,
  - $t_i + 20$ ,
  - $t_i + 9$ ,
  - nessuno dei precedenti
- 6.6. Un processo produttore produce un nuovo elemento in 10 secondi ed un processo consumatore consuma un elemento in 20 secondi. In un sistema produttori-consumatori costituito da un singolo processo produttore, un singolo processo consumatore e un singolo buffer, entrambi i processi produttore e consumatore iniziano l'esecuzione al tempo 0. In quale momento il processo consumatore finirà di consumare 3 elementi?
- 20,
  - 60,
  - 70,
  - 90,
  - nessuno di a-d

## Problemi

- 6.1. Un programma concorrente effettua alcuni aggiornamenti di una variabile condivisa  $x$  all'interno di una sezione critica. La variabile  $x$  è anche usata nella seguente parte di codice che non è inserita all'interno di una sezione critica:

```

if  $x < c$ 
  then  $y = x$ ;
  else  $y = x + 10$ ;
print  $x, y$ ;

```

In questo programma, può verificarsi una race condition?

- 6.2. Due processi concorrenti condividono un dato condiviso  $somma$ , inizializzato a 0. Tuttavia, non utilizzano la mutua esclusione per accedere al suo valore. Ogni processo contiene un ciclo che viene eseguito 50 volte e contiene una singola istruzione  $somma = somma + 1$ . Se non vengono effettuate altre operazioni su  $somma$ , indicare il limite inferiore ed il limite superiore al valore di  $somma$  quando entrambi i processi terminano.
- 6.3. Analizzare gli Algoritmi 6.1 e 6.2 e commentare le proprietà della sezione critica che essi violano. Fornire degli esempi illustrando le violazioni.
- 6.4. Rispondere alle seguenti domande nel contesto dell'algoritmo di Dekker.
- L'algoritmo soddisfa la condizione di progresso?
  - Si può verificare una condizione di deadlock?
  - Si può verificare una condizione di livelock?
- 6.5. La condizione di attesa limitata è soddisfatta dall'algoritmo di Peterson?
- 6.6. Vengono effettuate le seguenti modifiche all'algoritmo di Peterson (vedi Algoritmo 6.4): le istruzioni  $flag[0] = true$  e  $flag[0] = false$  nel processo  $P_0$  vengono scambiate e lo stesso viene fatto nel processo  $P_1$ . Illustrare quali proprietà dell'implementazione delle sezioni critiche sono violate dal sistema risultante.
- 6.7. L'istruzione **while**  $flag[1] \text{ and } turn = 1$  nell'algoritmo di Peterson viene modificata a **while**  $flag[1] \text{ or } turn = 1$  nel processo  $P_0$  e la stessa modifica viene operata nel processo  $P_1$ . Quali proprietà dell'implementazione della sezione critica vengono violate nel sistema risultante?
- 6.8. Commentare l'effetto della cancellazione dell'istruzione **while**  $choosing[j] \text{ do } \{ nothing \}$ ; sul funzionamento dell'algoritmo del panettiere di Lamport.
- 6.9. La soluzione del problema produttori-consumatori mostrato in [Figura 6.37](#) usa le

funzioni del kernel *block* e *adivate* per la sincronizzazione dei processi. Può verificarsi una race condition. Illustrare come può verificarsi questa race condition. Il

Illustrare come può verificarsi questa race condition.

---

```

type           item = . . .;
var            buffer : item;
                  buffer_full : boolean;
                  producer_blocked : boolean;
                  consumer_blocked : boolean;
begin
  buffer_full := false;
  producer_blocked := false;
  consumer_blocked := false;
Parbegin
  repeat
    if buffer_full = false then
      { Producere in un buffer }
      buffer_full := true;
      if consumer_blocked = true then
        activate(consumatore);
        { Parte restante del ciclo }
    else
      producer_blocked := true;
      block(produttore);
      consumer_blocked := false;
  forever
Parend
repeat
  if buffer_full = true then
    { Consuma da un buffer }
    buffer_full := false;
    if producer_blocked = true then
      activate(produttore);
      { Parte restante del ciclo }
  else
    consumer_blocked := true;
    block(consumatore);
    producer_blocked := false;
  forever

```

Produttore
Consumatore

---

**Figura 6.37** Il problema dei produttori-consumatori con un errore di sincronizzazione dovuto a una race condition.

- 6.10. La soluzione al problema dei lettori-scrittori di [Figura 6.30](#) utilizza due semafori sebbene debba essere controllata una singola entità, ovvero il dato condiviso. Modificare questa soluzione in modo da utilizzare un singolo semaforo *rw\_permission* invece dei semafori *reading* e *writing*. (Suggerimento: eseguire una *wait(rw\_permission)* nel lettore solo se la lettura non è già in esecuzione.)
- 6.11. Modificare la soluzione al problema dei lettori-scrittori di [Figura 6.30](#) in modo da implementare un sistema lettoriscrittore che favorisca gli scrittori.
- 6.12. Implementare una sezione critica utilizzando le istruzioni Test-and-set o Swap del Paragrafo 6.5.2. Applicare i concetti introdotti nel Paragrafo 6.8.2 affinché venga rispettata la condizione di attesa limitata.
- 6.13. Una risorsa deve essere allocata ai processi che la richiedono in ordine FIFO. Ogni processo è implementato come segue:

```

repeat
  richiesta-risorsa(id_processo, id_resource);
  { Usa risorsa }
  rilascia-risorsa( id_process, id_resource);
  { Resto del ciclo }
forever

```

Implementare le procedure *richiesta-risorsa* e *rilascia-risorsa* usando i semafori.

- 6.14. Possono una o più delle seguenti caratteristiche eliminare le pecche della soluzione al problema dei filosofi a cena mostrata in [Figura 6.20](#)?
  - a. Se  $n$  filosofi sono attivi nel sistema, prevedere posti per almeno  $n + 1$  filosofi.
  - b. Assicurarsi che almeno un filosofo mancino ed uno destroso siano seduti al tavolo nello stesso momento.
- 6.15. In Figura 1.33, i produttori ed i consumatori eseguono sempre le istruzioni *buf\_pieno.signal* e *buf\_vuoto.signal*. Proporre ed implementare un metodo per ridurre il numero di istruzioni **signal** eseguite durante il funzionamento del sistema.
- 6.16. Implementare una soluzione al problema dei filosofi a cena utilizzando i monitor.

Minimizzare il numero di esecuzioni dell'istruzione *signal* nella soluzione ed osservarne l'effetto sulla complessità logica della soluzione.

- 6.17. Un cliente impedisce le seguenti istruzioni al manager di una banca: non accreditare nessun fondo sul mio conto se il bilancio del conto supera  $n$  e mantieni tutti i debiti finché il bilancio del conto non è sufficiente a pagare il debito. Progettare un monitor per implementare il conto bancario del cliente.
- 6.18. Il problema di sincronizzazione denominato il *barbiere addormentato* è descritto come segue: un barbiere ha una sola poltrona per il taglio in una piccola stanza ed una stanza grande con  $n$  sedie. Il barbiere e la poltrona per il taglio sono visibili dalla sala d'aspetto. Dopo aver servito un cliente, il barbiere controlla se ci sono clienti in attesa. In caso affermativo, ne fa accomodare uno e lo serve; altrimenti, si addormenta sulla poltrona per il taglio. Un cliente entra nella sala d'attesa solo se è disponibile almeno una sedia e o attende che il barbiere lo chiami se è impegnato, oppure lo sveglia se è addormentato. Identificare i vincoli di sincronizzazione tra i processi barbiere e cliente. Implementare i processi barbiere e cliente in modo che non si verifichino deadlock.
- 6.19. Si implementi un monitor per la simulazione di un gestore del clock per il controllo real-time di processi concorrenti. Il gestore del clock utilizza una variabile *clock* per memorizzare il tempo corrente. Il SO supporta un segnale chiamato *tempo\_trascorso* che viene generato ogni 2 ms. Il gestore del clock fornisce un'azione di gestione del segnale per *tempo\_trascorso* (Paragrafo 5.4.1) che aggiorna *clock* a ogni occorrenza del segnale. Questa azione è codificata come procedura del monitor. Una tipica richiesta al gestore del clock è "risvegliami alle 9.00 a.m." Il gestore del clock blocca i processi per gestire tali richieste e organizza la loro attivazione al tempo segnalato. Implementare questo monitor.
- 6.20. L'esecuzione di chiamate innestate a un monitor implica che una procedura in un monitor A richiami una procedura di un altro monitor, per esempio il monitor B. Durante l'esecuzione della chiamata innestata, la procedura del monitor A continua a mantenere la sua mutua esclusione. Mostrare che l'esecuzione di chiamate innestate a un monitor può portare a situazioni di deadlock.
- 6.21. Scrivere una breve nota riguardo l'implementazione dei monitor. La nota deve affrontare:
- come ottenere la mutua esclusione tra le procedure del monitor;
  - se le procedure del monitor devono essere implementate in maniera rientrante (Paragrafo 11.3.3.2).
- 6.22. Un insieme elevato di dati  $D$  viene utilizzato esclusivamente per rispondere a query ovvero, non vengono effettuati aggiornamenti su  $D$ , per cui le query possono essere eseguite in maniera concorrente. Data la dimensione di  $D$ , viene suddiviso in diverse porzioni  $D_1, D_2, \dots, D_n$  e a ogni istante solo una di queste parti, per esempio  $D_1$ , è caricata in memoria per gestire le query a essa relative. Se non ci sono query attive su  $D_1$  e se esistono query per altre porzioni dell'insieme di dati, per esempio  $D_2, D_3$  viene caricato in memoria e le query su di essa vengono eseguite in maniera concorrente. Quando  $D$  è diviso in due porzioni  $D_1$  e  $D_2$ , questo sistema viene chiamato sistema lettori-lettori. Implementare questo sistema, usando qualunque primitiva di sincronizzazione o struttura di controllo si desideri. Per evitare la starvation delle query, viene proposto di gestire un massimo di 10 query su ogni porzione di dati. Modificare il monitor per incorporare queste caratteristiche.
- 6.23. Un ponte su un'autostrada trafficata viene danneggiato da una inondazione. Deve essere istituita una strada a senso alternato sul ponte consentendo ai veicoli di transitare in direzioni opposte in modo alternato. Vengono formulate le seguenti regole per l'utilizzo del ponte:
- a ogni istante, il ponte viene utilizzato dai veicoli che transitano in una sola direzione;
  - se ci sono veicoli in attesa di transitare in entrambe le direzioni, essi si devono alternare nel passaggio sul ponte;
  - se non ci sono veicoli in attesa in una direzione, allora un qualsiasi numero di veicoli provenienti dalla direzione opposta può attraversare il ponte.

Sviluppare un sistema concorrente che implementi queste regole.

- 6.24. Quando ci sono veicoli in attesa in entrambe le direzioni, le regole del Problema 6.23(a) portano a un uso non ottimale del ponte. Dunque dovrebbe essere consentito il transito fino a 10 veicoli in una direzione anche se ci sono veicoli nella direzione

opposta. Implementare le regole modificate.

## Laboratorio 1: sincronizzazione

Si consideri il seguente problema: in un ristorante più persone siedono allo stesso tavolo e condividono B1 bottiglie di acqua e B2 bottiglie di vino. La politica del ristorante è che un operatore controlla se terminano le bottiglie e le sostituisce se vuote fino a quando le persone non decidono di lasciare il ristorante. Si fornisca una soluzione usando semafori e processi. Discutere la soluzione proposta in termini di possibili situazioni di starvation/deadlock e, nel caso, si propongano soluzioni ammissibili.

## Laboratorio 2: comunicazione tra processi

Un sistema di comunicazione tra processi mediante messaggi utilizza la convenzione *asymmetric naming* che verrà descritta nel Paragrafo 9.1.1, che fa uso delle seguenti regole: per inviare un messaggio, un mittente fornisce l'id del processo destinatario al quale deve essere consegnato. Per ricevere un messaggio, un processo fornisce semplicemente il nome di una variabile in cui il messaggio dovrebbe essere depositato; il sistema restituisce un messaggio inviato al processo richiedente da qualche *altro* processo.

Il sistema si compone di un monitor chiamato *Communication Manager* e di quattro processi. Il monitor fornisce le operazioni *send* e *receive*, che implementano lo scambio dei messaggi utilizzando un pool *globale* di 20 buffer di messaggi. Il funzionamento del sistema è il seguente.

1. Ogni processo ha un comportamento ciclico. Il suo funzionamento è governato da comandi contenuti in un file dei comandi utilizzato esclusivamente dal processo. A ogni iterazione, il processo legge un comando dal file e richiama un'appropriata funzione del monitor. Sono previsti tre comandi:
  - a. *send* <process\_id>, <message\_text>: il processo deve inviare un messaggio;
  - b. *receive* <variablejame>: il processo deve ricevere un messaggio;
  - c. *quit*: il processo deve terminare la propria esecuzione.
2. Quando un processo richiama l'operazione *send*, il monitor copia il testo del messaggio in un elemento vuoto del buffer. Se, al momento, il processo destinatario del messaggio è bloccato su un'operazione *receive*, il messaggio gli viene consegnato come descritto al punto 3 ed il processo viene riattivato. In ogni caso, il controllo viene restituito al processo che sta eseguendo l'operazione *send*. Se il buffer è pieno, il processo che sta eseguendo l'operazione *send* viene bloccato finché non si libera una posizione del buffer.
3. Quando un processo invoca un'operazione *receive*, i messaggi a esso indirizzati gli vengono recapitati secondo l'ordine FIFO. Il monitor cerca nel buffer il primo messaggio non consegnato al processo, copia il testo del messaggio nella variabile passata dal processo e libera l'elemento del buffer. Se un processo che sta eseguendo l'operazione *send* risulta bloccato come descritto al punto 2, non viene attivato. Il processo che sta eseguendo l'operazione *receive* viene bloccato se non ci sono messaggi da consegnargli. Viene riattivato quando un messaggio gli viene inviato.
4. Dopo aver eseguito un'operazione *send* o *receive*, il monitor scrive i dettagli dell'operazione eseguita in un file di log.
5. Il monitor rileva una situazione di deadlock, in cui alcuni processi sono bloccati indefinitamente. Scrive i dettagli della situazione di deadlock nel file di log e termina la propria esecuzione.
6. Il sistema di comunicazione tra processi tramite messaggi termina la sua esecuzione quando tutti i processi hanno completato la propria esecuzione.

Implementare il monitor *Communication\_Manager* e testarne il funzionamento con diversi insiemi di comandi che creino situazioni interessanti, incluse alcune situazioni di deadlock.

## Laboratorio 3: scheduler del disco

Uno *scheduler del disco* è quella parte del SO che decide l'ordine in cui le operazioni di I/O devono essere eseguite su un disco per ottenere un elevato throughput del disco (Paragrafo 14.7). I processi che intendono eseguire un'operazione sul disco utilizzano un monitor chiamato *Disk\_scheduler* ed il seguente pseudocodice:

```
var Scheduler_disco : Disco_Mon_tipo;
Parbegin
begin { Processo utente  $P_i$  }
  var indirizzo_blocco_disco : integer;
  repeat
    {leggi un comando dal file  $F_i$  }
    ( $P_i$ , Operazione_I/O,
     Scheduler_disco.Richiesta_I/O
     indirizzo_blocco_disco);
    { Effettua l'Operazione_I/O }
    Scheduler_disco.IO_completato ( $P_i$ );
    { Resto del ciclo }
  forever
end;
... { altri processi utente }
Parend;
```

Ogni processo ha un comportamento ciclico. Il suo funzionamento è governato dai comandi contenuti in un file dei comandi utilizzato esclusivamente dal processo.

Ogni comando serve per eseguire un'operazione di read o write su un blocco del disco. A ogni iterazione, un processo legge un comando dal suo file dei comandi ed invoca l'operazione del monitor *IO\_request* per passare i dettagli dell'operazione di I/O. *IO\_request* blocca il processo finché l'operazione di I/O che ha richiesto non viene schedulata. Quando il processo viene attivato, ritorna da *IO\_request* ed esegue la sua operazione di I/O. Dopo aver completato l'operazione di I/O, invoca l'operazione del monitor *IO\_complete* in modo tale che il monitor possa schedulare la prossima operazione di I/O. Il monitor scrive i dettagli delle sue azioni in un file di log ogni volta che le operazioni *IO\_request* o *IO\_complete* vengono invocate.

Implementare il tipo monitor *Disk\_Mon\_type*. Per semplicità, si assuma che le operazioni di I/O vengano schedulate in ordine FIFO e che il numero di processi non ecceda 10. (Suggerimento: annotare l'id di un processo insieme con i dettagli della sua operazione di I/O in una lista interna al monitor. Decidere quante variabili di condizione sono necessarie per bloccare ed attivare i processi.)

Modificare *Disk\_Mon\_type* in modo tale che le operazioni di I/O siano eseguite dal monitor stesso piuttosto che dai processi utente. (Suggerimento: l'operazione *I/O\_complete* non sarà più necessaria.)

## Note bibliografiche

Dijkstra (1965) discute il problema della mutua esclusione, descrive l'algoritmo di Dekker e presenta un algoritmo di mutua esclusione per  $n$  processi. Lamport (1974, 1979) descrive e dimostra l'algoritmo del panettiere. Ben Ari (1982) descrive l'evoluzione degli algoritmi di mutua esclusione e fornisce una dimostrazione dell'algoritmo di Dekker. Ben Ari (2006) discute la programmazione concorrente e distribuita. Peterson (1981), Lamport (1986, 1991) e Raynal (1986) sono ulteriori fonti sugli algoritmi di mutua esclusione.

Dijkstra (1965) propose i semafori. Hoare (1972) e Brinch Hansen (1972) discutono le regioni critiche e le regioni condizionali critiche, che sono costrutti di sincronizzazione che precedono i monitor. Brinch Hansen (1973) e Hoare (1974) descrivono il concetto di monitor. Buhr et al. (1995) descrivono diverse

implementazioni dei monitor. Richter (1999) descrive la sincronizzazione dei thread nei programmi C/C++ in ambiente Windows. Christopher e Thiruvathukal (2001) descrivono il concetto di monitor in Java, lo confrontano con i monitor di Brinch Hansen e Hoare e concludono che la sincronizzazione in Java non è ben sviluppata quanto i monitor di Brinch Hansen e Hoare.

Un costrutto o una primitiva di sincronizzazione sono completi se possono essere usati per implementare *tutti* i problemi di sincronizzazione dei processi. La completezza dei semafori viene discussa in Patil (1971), Lipton (1974) e Kosaraju (1975).

Brinch Hansen (1973, 1977) e Ben Ari (1982, 2006) discutono le metodologie per costruire programmi concorrenti. Owicki e Gries (1976) e Francez e Pneuli (1978) affrontano la metodologia per garantire la correttezza dei programmi concorrenti.

Vahalia (1996) e Stevens e Rago (2005) discutono la sincronizzazione dei processi in Unix, Beck et al. (2002), Bovet e Cesati (2005), e Love (2005), discutono la sincronizzazione in Linux, Mauro e McDougall (2006) discutono la sincronizzazione in Solaris, mentre Richter (1999) e Russinovich e Solomon (2005) discutono le caratteristiche di sincronizzazione in Windows.

1. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, Pearson Education, New York.
2. Ben Ari, M. (1982): *Principles of Concurrent Programming*, Prentice Hall, Englewood Cliffs, N.J.
3. Ben Ari, M. (2006): *Principles of Concurrent and Distributed Programming*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
5. Brinch Hansen, P. (1972): "Structured multiprogramming," *Communications of the ACM*, **15** (7), 574-578.
6. Brinch Hansen, P. (1973): *Operating System Principles*, Prentice Hall, Englewood Cliffs, N.J.
7. Brinch Hansen, P. (1975): "The programming language concurrent Pascal," *IEEE Transactions on Software Engineering*, **1** (2), 199-207.
8. Brinch Hansen, P. (1977): *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, N.J.
9. Buhr, M., M. Fortier, and M.H. Coffin (1995): "Monitor classification," *Computing Surveys*, **27** (1), 63-108.
10. Chandy, K.M., and J. Misra (1988): *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Mass.
11. Christopher, T.W., and G.K. Thiruvathukal (2001): *Multithreaded and Networked Programming*, Sun Microsystems.
12. Courtois, P.J., F. Heymans, and D.L. Parnas (1971): "Concurrent control with readers and writers," *Communications of the ACM*, **14** (10), 667-668.
13. Dijkstra, E.W. (1965): "Cooperating sequential processes," Technical Report EWD-123, Technological University, Eindhoven.
14. Eisenberg, M.A., and M.R. McGuire (1972): "Further comments on Dijkstra's concurrent programming control problem" *Communications of the ACM*, **15**(11), 999.
15. Francez, N., and A. Pneuli (1978): "A proof method for cyclic programs," *Acta Informatica*, **9**, 133-157.
16. Hoare, C.A.R. (1972): "Towards a theory of parallel programming," in *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrot (eds.), Academic Press, London, 1972.
17. Hoare, C.A.R (1974): "Monitors: an operating system structuring concept," *Communications of the ACM*, **17**(10), 549-557.
18. Kosaraju, S. (1973): "Limitations of Dijkstra's semaphore primitives and petri nets," *Operating Systems Review*, **7**, 4, 122-126.
19. Lamport, L. (1974): "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, **17**, 453-455.
20. Lamport, L. (1979): "A new approach to proving the correctness of multiprocess programs," *ACM Transactions on Programming Languages and Systems*, **1**, 84-97.
21. Lamport, L. (1986): "The mutual exclusion problem," *Communications of the ACM*, **33** (2), 313-348.
22. Lamport, L. (1991): "The mutual exclusion problem has been solved," *ACM*

- Transactions on Programming Languages and Systems*, **1**, 84-97.
- 23. Lipton, R. (1974): "On synchronization primitive systems," Ph.D. Thesis, Carnegie-Mellon University.
  - 24. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
  - 25. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
  - 26. Owicki, S., and D. Gries (1976): "Verifying properties of parallel programs: An axiomatic approach," *Communications of the ACM*, **19**, 279-285.
  - 27. Patil, S. (1971): "Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes," Technical Report, MIT.
  - 28. Peterson, G.L. (1981): "Myths about the mutual exclusion problem," *Information Processing Letters*, **12**, 3.
  - 29. Raynal, M. (1986): *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, Mass.
  - 30. Richter, J. (1999): *Programming Applications for Microsoft Windows*, 4th ed., Microsoft Press, Redmond, Wash.
  - 31. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
  - 32. Stevens, W.R., and S.A. Rago (2005): *Advanced Programming in the Unix Environment*, 2nd ed., Addison Wesley, Reading, Mass.,
  - 33. Vahalia, U. (1996): *Unix Internals - The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 7

## Scheduling

---

### Obiettivi di apprendimento

- Definizione di schedulazione dei processi
- Politiche di schedulazione preemptive e non preemptive
- Implementazione dello scheduling
- Scheduling real-time
- Scheduling nei sistemi operativi Unix, Solaris, Linux, Windows
- Analisi e valutazioni delle politiche di scheduling

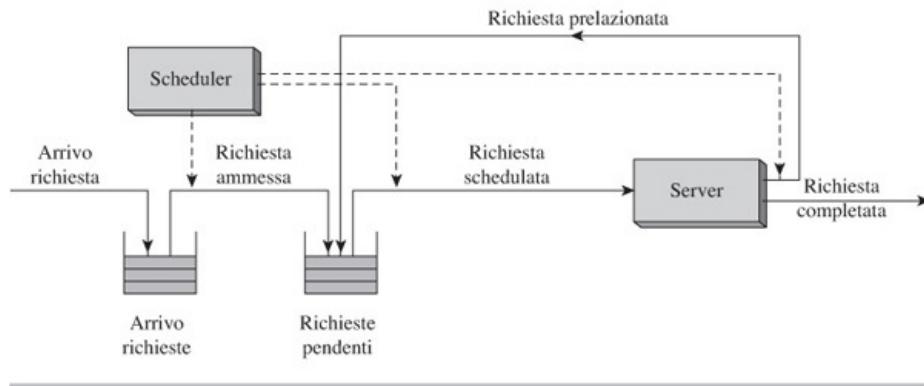
Una politica di scheduling decide a quale processo debba essere assegnata la CPU in un dato momento. Tale scelta, ovviamente, influenza sia le prestazioni del sistema che i servizi utente. Nel [Capitolo 3](#), abbiamo visto come *lo scheduling a priorità* fornisce buone prestazioni di sistema, e come *lo scheduling round-robin* con *time-slicing* fornisce buoni tempi di risposta per i processi. La politica di scheduling negli attuali sistemi operativi deve mirare a fornire la migliore combinazione tra servizio utente e prestazioni di sistema per meglio adattarsi al suo ambiente computazionale.

Una politica di scheduling impiega tre tecniche fondamentali per ottenere la combinazione tra servizio utente e prestazione di sistema desiderata: 1) *l'assegnazione delle priorità* ai processi può fornire buone prestazioni di sistema, come in un sistema multiprogrammato, o attribuire trattamenti privilegiati a funzioni importanti, come in un sistema real-time; 2) la *variazione della time slice* permette allo scheduler di adattare la time slice alla natura di un processo in modo tale da fornirgli un appropriato tempo di risposta, e da poter controllare il suo overhead; 3) il *riordino dei processi* può incrementare sia le prestazioni di sistema, misurate come produttività (throughput), che il servizio utente, misurato come tempi di completamento (turnaround time) o tempi di risposta dei processi. Nel seguito verrà discusso l'uso di queste tecniche e un insieme di euristiche di scheduling negli attuali sistemi operativi.

*L'analisi delle prestazioni* di una politica di scheduling è lo studio delle sue prestazioni. Può essere utilizzata per confrontare le prestazioni di due politiche di scheduling differenti oppure per determinare i valori di parametri chiave del sistema, come la dimensione di una coda dei processi. Verranno descritti diversi approcci all'analisi delle prestazioni delle politiche di scheduling.

### 7.1 Scheduling: terminologia e concetti

Lo scheduling, in generale, è l'attività di selezione della successiva richiesta che deve essere elaborata da un *server*. La [Figura 7.1](#) è un diagramma schematico dello scheduling. Lo scheduler considera attivamente una lista di richieste in attesa di essere elaborate e ne seleziona una per l'elaborazione da parte del server. Tale richiesta viene abbandonata dal server all'atto del completamento oppure quando lo scheduler la preleva e la riporta nella lista delle richieste in attesa. In entrambe le situazioni, lo scheduler seleziona la richiesta successiva da elaborare. Di volta in volta, lo scheduler ammette una nuova richiesta di elaborazione e la inserisce nella lista delle richieste in attesa. Le azioni dello scheduler sono mostrate dalle frecce tratteggiate in [Figura 7.1](#). Gli eventi relativi a una richiesta sono il suo *arrivo*, *ammissione*, *scheduling*, *prelazione* e *completamento*.



**Figura 7.1** Uno schema di scheduling.

In un sistema operativo, una *richiesta* è l'esecuzione di un job o di un processo, e il *server* è la CPU. Un job o un processo è detto in *arrivo* quando viene *sottomesso* dall'utente, ed è *ammesso* quando lo scheduler comincia a considerarlo nello scheduling. Un job o processo in attesa di essere schedulato può essere nella lista delle richieste in attesa oppure utilizzare la CPU o anche eseguire operazioni di I/O; infine, terminerà liberando le risorse. La decisione dello scheduler di quale richiesta ammettere è rilevante solo nei sistemi operativi con risorse limitate; per semplicità, nella maggior parte delle nostre discussioni assumiamo che una richiesta è ammessa automaticamente in coda.

Nel [Capitolo 3](#) abbiamo discusso di come l'uso delle priorità nello scheduler fornisce buone prestazioni di sistema, mentre l'uso dello scheduling round-robin fornisce un buon servizio in termini di risposta veloce. Gli attuali sistemi operativi utilizzano politiche di scheduling molto più complesse per raggiungere la migliore combinazione tra prestazione di sistema e servizio utente.

La [Tabella 7.1](#) elenca i termini e i concetti chiave relativi allo scheduling. Il *tempo di servizio* di un job o di un processo è la somma del tempo di CPU e di quello di I/O da esso richiesto per completare la sua esecuzione, e la *deadline*, che è specificata solo nei sistemi real-time (Paragrafo 3.7), è il tempo entro il quale la sua esecuzione dovrebbe essere completata. Sia il tempo di servizio che la deadline sono proprietà specifiche di un job o di un processo. Il *tempo di completamento* di un job o di un processo dipendono dai suoi tempi di arrivo e di servizio e dal tipo di servizio che riceve dal sistema operativo.

| Termine o concetto   | Definizione o descrizione   |
|--|---|
| <b>Relativamente alla richiesta</b>                                  |   |
| Tempo di arrivo  | Istante in cui un utente invia un job o un processo.  |
| Tempo di ammissione  | Istante in cui il sistema comincia a considerare un job o un processo per lo scheduling.  |
| Tempo di completamento   | Istante in cui un job o un processo è terminato.  |
| Deadline   | Istante entro il quale un job o un processo deve essere terminato per rispettare il requisito di risposta di un'applicazione real-time.                               |
| Tempo di servizio  | Il totale del tempo di CPU e quello di I/O richiesto da un job, processo o sottorichiesta per completare la sua operazione.   |
| Prelazione   | Deallocazione forzata della CPU da un job o da un processo.   |
| Priorità   | Una regola discriminante usata per selezionare un job o un processo quando molti job o processi attendono l'elaborazione.   |
| <b>Relativamente al servizio per l'utente: richieste individuali</b> |   |
| Superamento della deadline (deadline overrun)                        | La quantità di tempo per la quale il tempo di completamento di un job o un processo supera la sua deadline. Il deadline overrun può essere sia positivo sia negativo. |

|   |   |
|---|---|
| Condivisione equa   | Una condivisione specifica del tempo di CPU che dovrebbe essere dedicato all'esecuzione di un processo o di un gruppo di processi.  |
| Rapporto di risposta  | Il rapporto<br>$\frac{\text{tempo di attesa} + \text{tempo di servizio di un job o processo}}{\text{tempo di servizio del job o processo}}$   |
| Tempo di risposta ( $rt$ )  | Il tempo tra la sottomissione di una sottorichiesta per l'elaborazione e il tempo in cui il suo risultato diventa disponibile. Questo concetto è applicabile ai processi interattivi.   |
| Tempo impiegato per il completamento o tempo di turnaround ( $ta$ ) | Il tempo che intercorre tra la sottomissione di un job o processo e il suo completamento da parte del sistema. Questo concetto è significativo solo per job non interattivi o processi. |
| Turnaround pesato ( $w$ )   | Rapporto del tempo di turnaround di un job o processo e il suo tempo di servizio.   |
| <b>Relativamente al servizio per l'utente: servizio medio</b>       |   |
| Tempo di risposta medio ( $\bar{rt}$ )                              | La media dei tempi di risposta di tutte le sottorichieste elaborate dal sistema.  |
| Tempo medio di turnaround ( $\bar{ta}$ )                            | La media dei tempi di turnaround di tutti i job o processi elaborati dal sistema.   |
| <b>Relativamente alle prestazioni</b>                               |   |
| Durata della schedulazione  | Il tempo necessario per completare un insieme specifico di job o processi.  |
| Throughput  | Il numero medio di job, processi o sottorichieste completate da un sistema nell'unità di tempo.   |

**Tabella 7.1** Scheduling: termini e concetti.

Suddividiamo i concetti di scheduling in concetti relativi all'utente e concetti relativi al sistema per discernere ciò che riguarda il servizio utente da ciò che concerne le prestazioni di sistema.

### **Concetti di scheduling relativi all'utente**

In un ambiente interattivo, un utente interagisce con un processo durante un'operazione - l'utente fa una *sottorichiesta* a un processo e il processo risponde eseguendo azioni o calcolando risultati. Il *tempo di risposta* è il tempo intercorso tra la sottomissione di una sottorichiesta e il momento in cui la sua elaborazione è terminata; è una misura assoluta del servizio fornito a una sottorichiesta. Il *tempo di completamento* è un'analogia misura assoluta del servizio fornito a un job o a un processo. Il tempo di completamento differisce dal tempo di servizio di un job o di un processo perché include anche il tempo in cui il job o il processo non è in esecuzione nella CPU e non sta eseguendo operazioni di I/O. Tali misure sono state inizialmente introdotte al [Capitolo 3](#).

Vengono definite diverse altre misure, quali il *completamento pesato* che mette in relazione il tempo di completamento di un processo con il suo tempo di servizio. Per esempio, un completamento pesato di 5 indica che il completamento ricevuto da una richiesta è 5 volte il suo tempo di servizio. Il confronto dei completamenti pesati di diversi job o processi indica il servizio comparativo ricevuto. *Condivisione equa* è la condivisione del tempo di CPU che dovrebbe essere assegnato a un processo o a un gruppo di processi. Il *rapporto di risposta* (response ratio) di un job o di un processo è il rapporto (tempo di attesa + tempo di servizio)/tempo di servizio e descrive il ritardo nell'espletamento di un job o di un processo rispetto al suo tempo di servizio. L'impiego del rapporto di risposta è prevalente nel caso di una politica di scheduling per evitare la starvation dei processi (Paragrafo 7.2.3). Il *deadline overrun* è la differenza tra il tempo di completamento e la scadenza (deadline) di un job o di un processo in un'applicazione real-time. Un valore negativo per il deadline overrun indica che il job o il processo è stato completato prima della sua scadenza (deadline), mentre un valore positivo indica che la

scadenza (deadline) è stata superata. Il *tempo medio di risposta* (mean response time) e il *tempo medio di completamento* (mean turnaround time) sono misure del servizio medio fornito alle sottorichieste e ai processi o job rispettivamente.

### **Concetti di scheduling relativi al sistema**

Throughput e durata della schedulazione sono misure delle prestazioni del sistema. Il *throughput* indica il numero medio di richieste o sottorichieste completate nell'unità di tempo (Paragrafo 3.5). Fornisce una base per confrontare le prestazioni di due o più politiche di scheduling o per confrontare le prestazioni della stessa politica di scheduling in diversi periodi di tempo. La *durata della schedulazione* indica la quantità di tempo totale utilizzata dal server per completare un insieme di richieste.

Il throughput e la durata della schedulazione sono misure collegate fra loro. Consideriamo l'elaborazione di cinque richieste  $r_1, \dots, r_5$ . Siano  $min_a$  e  $max_c$  il minimo dei tempi di arrivo e il massimo dei tempi di completamento, rispettivamente. La lunghezza della schedulazione per queste cinque richieste è  $(max_c - min_a)$  e il throughput è  $5/(max_c - min_a)$ . Tipicamente non è possibile calcolare la durata della schedulazione e throughput in questo modo poiché un SO può ammettere ed elaborare anche altre richieste nell'intervallo di tempo tra  $min_a$  e  $max_c$ , per raggiungere buone prestazioni di sistema. Tuttavia, la durata della schedulazione è un importante termine di confronto della prestazione di politiche di scheduling quando l'overhead dello scheduling non è trascurabile. Il throughput è collegato al tempo medio di risposta (mean response time) e al tempo medio di completamento (mean turnaround time) in modo analogo.

#### **7.1.1 Tecniche fondamentali di scheduling**

Gli scheduler usano tre tecniche fondamentali di progettazione per fornire un buon servizio utente ovvero elevate prestazioni di sistema:

- *scheduling basato su priorità*: il processo in esecuzione dovrebbe essere il processo a più alta priorità che richiede l'uso della CPU. Questo è assicurato selezionando il processo in stato *Ready* a più alta priorità e prelazionandolo quando un processo con priorità più alta diventa *Ready*. Si ricordi dal Paragrafo 3.5.1 che un SO multiprogrammato assegna un'alta priorità ai processi di I/O-bound; questa assegnazione di priorità fornisce un elevato throughput del sistema;
- *riordino delle richieste*: il riordino implica che le richieste verranno espletate in un ordine diverso rispetto a quello di arrivo. Il riordino può essere utilizzato per incrementare il servizio utente; per esempio, espletare brevi richieste prima di quelle lunghe riduce il tempo di completamento medio delle richieste. Il riordino delle richieste è implicito nella *préemption*, che può essere utilizzata per migliorare il servizio utente, come in un sistema time-sharing, o per migliorare il throughput di sistema, come in un sistema multiprogrammato;
- *variazione della time slice*: quando si fa uso del time-slicing, dall'Equazione 3.2 del Paragrafo 3.6,  $\eta = \delta/(\delta + \sigma)$  dove  $\eta$  è l'efficienza della CPU,  $\delta$  è il time slice e  $\sigma$  è il costo di gestione (overhead) del SO per le decisioni di scheduling. Si ottengono tempi di risposta migliori quando vengono utilizzati valori inferiori per la time slice; tuttavia, esso riduce l'efficienza della CPU poiché deve essere sostenuto un costo rilevante per il cambio di contesto di un processo (context switch). Allo scopo di effettuare un bilancio tra efficienza della CPU e tempi di risposta, un SO potrebbe utilizzare valori diversi di  $\delta$  per differenti richieste – un valore piccolo per richieste I/O-bound e un valore grande per richieste CPU-bound – oppure potrebbe variare il valore di  $\delta$  per un processo quando il suo comportamento cambia da CPU-bound a I/O-bound, o da I/O-bound a CPU-bound.

Nei Paragrafi 7.2 e 7.3 verranno descritte le tecniche di scheduling basato su priorità e il riordino delle richieste nell'ambito delle classiche politiche di scheduling preemptive e nonpreemptive. Nei Paragrafi 7.4 e 7.5, illustreremo come gli scheduler nei moderni sistemi operativi combinano queste tre tecniche fondamentali per fornire una combinazione di buone prestazioni e buon servizio utente.

#### **7.1.2 Il ruolo della priorità**

La priorità è una regola discriminante che viene impiegata da uno scheduler quando più

richieste attendono attenzione dal server. La priorità di una richiesta può essere funzione di diversi parametri; ogni parametro rispecchia un attributo inerente la richiesta oppure un aspetto riguardante il suo espletamento. È detta *priorità dinamica* se alcuni dei suoi parametri cambiano durante l'elaborazione della richiesta; altrimenti, è detta *priorità statica*.

Alcuni riordini di processi potrebbero essere ottenuti tramite le priorità. Per esempio, i processi brevi verrebbero elaborati prima di quelli lunghi se la priorità fosse inversamente proporzionale al tempo di servizio di un processo, e i processi che hanno ricevuto minor tempo di CPU verrebbero elaborati prima se la priorità fosse inversamente proporzionale al tempo di CPU utilizzato da un processo. Comunque, possono essere necessarie funzioni di priorità complesse per ottenere alcuni tipi di riordino dei processi come quelli ottenuti tramite il time-slicing; tale uso incrementerebbe il costo di gestione (overhead) dello scheduling. In tali situazioni, gli scheduler utilizzano algoritmi per determinare l'ordine in cui le richieste vengono elaborate.

Se due o più richieste hanno la stessa priorità, quale richiesta dovrebbe essere schedulata per prima? Uno schema tipico prevede l'uso dello scheduling round-robin tra tali richieste. Tale modalità, in cui i processi con la stessa priorità condividono la CPU tra di loro se nessuno dei processi a più alta priorità è pronto, fornisce un servizio utente migliore rispetto al privilegiare una delle richieste rispetto alle altre con la stessa priorità.

Lo scheduling basato su priorità ha lo svantaggio che una richiesta a bassa priorità potrebbe non essere mai espletata fintanto che arrivano richieste con priorità più alta. Questa situazione è detta *starvation*. Questo potrebbe essere evitato incrementando la priorità di una richiesta che non è stata schedulata per un certo lasso di tempo. In questo modo, la priorità di una richiesta a bassa priorità si incrementerebbe nell'attesa di essere schedulata finché la sua priorità non supera la priorità di tutte le altre richieste in attesa. Questa tecnica è detta *aging* delle richieste.

## 7.2 Politiche di scheduling nonpreemptive

Nello *scheduling nonpreemptive*, un server elabora sempre una richiesta schedulata fino a completamento. In tal modo, lo scheduling è eseguito solo quando l'elaborazione di una richiesta schedulata precedentemente è completata e quindi la prelazione di una richiesta come mostrato in [Figura 7.1](#) non si verifica mai. Lo scheduling di tipo nonpreemptive è interessante perché molto semplice – lo scheduler non deve distinguere tra una richiesta non elaborata e una parzialmente elaborata.

Poiché una richiesta non è mai prelazionata, lo scheduler ha soltanto la funzione di riordinare le richieste per migliorare il servizio utente o la prestazione di sistema. Discutiamo le tre politiche di scheduling nonpreemptive in questo paragrafo.

- Scheduling first-come, first-served (FCFS)
- Scheduling shortest job first (SJF)
- Scheduling highest response ratio next (HRN)

Illustriamo il funzionamento e le prestazioni delle varie politiche di scheduling con l'aiuto dei cinque processi mostrati in [Tabella 7.2](#). Per semplicità assumiamo che questi processi non eseguano operazioni di I/O.

| Processo            | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---------------------|-------|-------|-------|-------|-------|
| Tempo di ammissione | 0     | 2     | 3     | 4     | 8     |
| Tempo di servizio   | 3     | 3     | 5     | 2     | 3     |

**Tabella 7.2** Processo di scheduling.

### 7.2.1 Scheduling FCFS

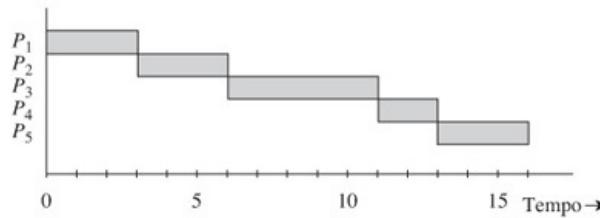
Le richieste sono schedolate nell'ordine in cui giungono al sistema. La lista delle richieste in attesa è organizzata come una coda e lo scheduler seleziona sempre la prima richiesta nella lista. Un esempio di scheduling FCFS è un sistema di elaborazione batch in cui i job sono ordinati in base al loro istante di arrivo (o arbitrariamente, se essi

arrivano esattamente nello stesso istante) e i risultati di un job sono restituiti all'utente immediatamente al termine del job. L'esempio successivo illustra il funzionamento di uno scheduler FCFS.

### Esempio 7.1 - Scheduling FCFS

La Figura 7.2 illustra le decisioni prese in base alla politica di scheduling FCFS per i processi della Tabella 7.2. Il processo  $P_1$  è schedulato all'istante 0. La lista dei processi in attesa contiene  $P_2$  e  $P_3$  quando  $P_1$  termina a 3 secondi, così  $P_2$  viene schedulato. La colonna *Completato* mostra l'id del processo completato, il suo tempo di completamento ( $ta$ ) e il completamento pesato ( $w$ ). I valori medi di  $ta$  e  $w$  ( $\bar{ta}$  e  $\bar{w}$ ) sono mostrati sotto la tabella. Il grafico temporale di Figura 7.2 mostra l'ordine di esecuzione dei processi.

| Tempo | Processo completato |           |          | Processi nel sistema<br>(in ordine FCFS) | Processo<br>schedulato |
|-------|---------------------|-----------|----------|--|------------------------|
|       | <i>id</i>           | <i>ta</i> | <i>w</i> |  |                        |
| 0     | —                   | —         | —        | $P_1$                                    | $P_1$                  |
| 3     | $P_1$               | 3         | 1.00     | $P_2, P_3$                               | $P_2$                  |
| 6     | $P_2$               | 4         | 1.33     | $P_3, P_4$                               | $P_3$                  |
| 11    | $P_3$               | 8         | 1.60     | $P_4, P_5$                               | $P_4$                  |
| 13    | $P_4$               | 9         | 4.50     | $P_5$                                    | $P_5$                  |
| 16    | $P_5$               | 8         | 2.67     | —  | —                      |



**Figura 7.2** Scheduling che utilizza la politica FCFS.

Dall'Esempio 7.1 si vede che si verifica una variazione considerevole nei completamenti pesati forniti dal scheduling FCFS. Questa variazione è maggiore nel caso in cui i processi soggetti a grandi tempi di completamento sono brevi - per esempio, il completamento pesato di  $P_4$  sarebbe stato maggiore se la sua richiesta di esecuzione fosse stata 1 secondo o 0.5 secondi.

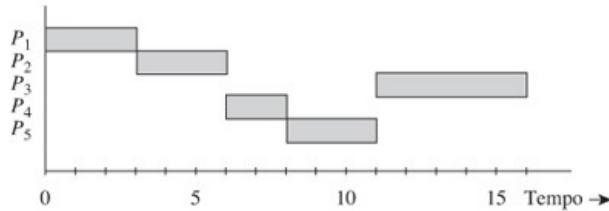
### 7.2.2 Scheduling Shortest Job First (SJF)

Lo scheduler SJF schedula sempre la richiesta con il minimo tempo di servizio. Così, una richiesta rimane in attesa finché non siano state elaborate tutte le richieste più brevi.

#### Esempio 7.2 - Scheduling Shortest Job First (SJF)

La Figura 7.3 illustra le decisioni prese in base alla politica di scheduling SJF per i processi della Tabella 7.2, e l'elaborazione dei processi. All'istante 0,  $P_1$  è il solo processo nel sistema, quindi è schedulato. Termina in 3 secondi. A questo punto, i processi  $P_2$  e  $P_3$  fanno parte del sistema, e  $P_2$  è più breve di  $P_3$ . Così  $P_2$  è schedulato, e così via.

| Tempo | Processo completato   |           |          | Processi nel sistema<br>(in ordine FCFS)          | Processo<br>schedulato |
|-------|-----------------------|-----------|----------|---|------------------------|
|       | <i>id</i>             | <i>ta</i> | <i>w</i> |   |                        |
| 0     | —                     | —         | —        | { <i>P</i> <sub>1</sub> }                         | <i>P</i> <sub>1</sub>  |
| 3     | <i>P</i> <sub>1</sub> | 3         | 1.00     | { <i>P</i> <sub>2</sub> , <i>P</i> <sub>3</sub> } | <i>P</i> <sub>2</sub>  |
| 6     | <i>P</i> <sub>2</sub> | 4         | 1.33     | { <i>P</i> <sub>3</sub> , <i>P</i> <sub>4</sub> } | <i>P</i> <sub>4</sub>  |
| 8     | <i>P</i> <sub>4</sub> | 4         | 2.00     | { <i>P</i> <sub>3</sub> , <i>P</i> <sub>5</sub> } | <i>P</i> <sub>5</sub>  |
| 11    | <i>P</i> <sub>5</sub> | 3         | 1.00     | { <i>P</i> <sub>3</sub> }                         | <i>P</i> <sub>3</sub>  |
| 16    | <i>P</i> <sub>3</sub> | 13        | 2.60     | {}  | —                      |



**Figura 7.3** Scheduling che utilizza la politica shortest job first (SJF).

Il tempo di completamento medio e il completamento pesato medio sono migliori rispetto allo scheduling FCFS poiché le richieste brevi tendono a ricevere tempi di completamento e completamento pesato più brevi. Questa caratteristica degrada il servizio che riceve richieste lunghe; tuttavia, i loro completamenti pesati non si incrementano di molto poiché i tempi di servizio sono lunghi. Il throughput è maggiore rispetto allo scheduling FCFS nei primi 10 secondi di schedulazione a causa dei processi brevi che sono in fase di elaborazione; comunque, è identico alla fine della schedulazione poiché sono stati elaborati processi uguali.

L'utilizzo in pratica della politica SJF mostra diverse difficoltà. I tempi di servizio dei processi non sono noti al sistema operativo *a priori*, di conseguenza il sistema operativo si aspetta che gli utenti forniscano stime dei tempi di servizio dei processi. Tale scheduling presenta un comportamento non corretto nel caso in cui gli utenti o possiedono sufficiente esperienza nello stimare i tempi di servizio, o manipolano il sistema per ottenere un servizio migliore dando stime dei tempi di servizio più basse per i loro processi. La politica SJF offre un servizio scarso a processi lunghi, perché un flusso costante di processi brevi che arrivano nel sistema può generare la starvation di un processo lungo.

### 7.2.3 Scheduling Highest Response Ratio Next (HRN)

La politica HRN calcola i rapporti di risposta di tutti i processi nel sistema secondo l'Equazione 7.1 e seleziona il processo con il più elevato rapporto di risposta.

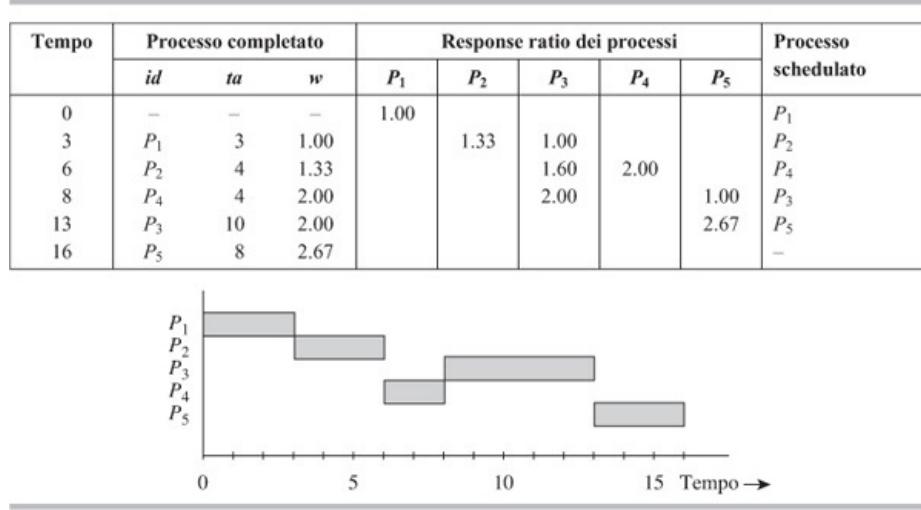
$$\text{Rapporto di risposta} = \frac{\text{tempo di attesa} + \text{tempo di servizio del processo}}{\text{tempo di servizio del processo}} \quad (7.1)$$

Il rapporto di risposta del processo giunto per ultimo è 1. Esso si incrementa secondo il rapporto (1/tempo di servizio) mentre è in attesa di essere eseguito. Il rapporto di risposta di un processo breve si incrementa più rapidamente di quello di un processo lungo, quindi i processi più brevi sono privilegiati per quanto riguarda lo scheduling. Comunque, il rapporto di risposta di un processo lungo può eventualmente diventare sufficientemente grande da consentire che il processo sia schedulato. Questa caratteristica provoca un effetto simile alla tecnica di *aging* discussa precedentemente nel Paragrafo 7.1.2, cosicché i processi lunghi non vanno incontro a starvation. L'esempio successivo illustra questa proprietà.

#### Esempio 7.3 - Scheduling Highest Response Ratio Next (HRN)

Il funzionamento della politica di scheduling HRN per i cinque processi riportati in **Tabella 7.2** è schematizzato in **Figura 7.4**. Prima che il processo *P*<sub>1</sub> termini, arrivano i processi *P*<sub>2</sub> e *P*<sub>3</sub>. *P*<sub>2</sub> ha un rapporto di risposta più alto di *P*<sub>3</sub>, per cui è il prossimo schedulato. Quando termina, *P*<sub>3</sub> ha un rapporto di risposta più alto di prima;

comunque,  $P_4$ , che è arrivato dopo  $P_3$ , ha anch'esso un rapporto di risposta più alto perché è un processo più breve, dunque  $P_4$  è schedulato. Quando  $P_4$  termina,  $P_3$  ha un rapporto di risposta più alto del processo più breve  $P_5$  poiché ha già trascorso un certo tempo in attesa, mentre  $P_5$  è appena arrivato. Di conseguenza ora viene schedulato  $P_3$ . Ciò comporta un completamento pesato minore per  $P_3$  rispetto allo scheduling SJF (Figura 7.3). Così, dopo una lunga attesa, un processo lungo viene schedulato prima di uno più breve.



**Figura 7.4** Funzionamento della politica highest response ratio next (HRN).

### 7.3 Politiche di scheduling preemptive

Nello *scheduling con diritto di prelazione* (preemptive), il server può essere commutato all'elaborazione di una nuova richiesta prima di aver completato quella corrente. La richiesta interrotta (preempted) è riportata nella lista delle richieste in attesa (Figura 7.1). La sua elaborazione viene ripresa quando sarà nuovamente schedulata. Quindi, una richiesta potrebbe essere schedulata più volte prima del suo completamento. Questa caratteristica causa un maggior carico (overhead) per lo scheduling rispetto a quello senza diritto di prelazione (nonpreemptive). Abbiamo discusso già in precedenza dello scheduling con diritto di prelazione (preemptive) nei sistemi operativi multi-programmati e time-sharing, nel Capitolo 3.

Trattiamo, in questo paragrafo, tre politiche di scheduling con diritto di prelazione (preemptive).

- Scheduling round-robin con time-slicing (RR)
- Scheduling least completed next (LCN)
- Scheduling shortest time to go (STG)

La politica di scheduling RR condivide la CPU tra le richieste ammesse elaborandole a turno. Le altre due politiche prendono in considerazione il tempo di CPU richiesto da una richiesta o da essa utilizzato per prendere le loro decisioni di scheduling.

#### 7.3.1 Scheduling Round-Robin con Time-Slicing (RR)

La politica RR si prefigge lo scopo di fornire buoni tempi di risposta a tutte le richieste. Il quanto di tempo (time slice), indicato con  $\delta$ , è la massima quantità di tempo di CPU che una richiesta può utilizzare quando è schedulata. Una richiesta è sospesa al termine di un quanto di tempo (time slice). Per semplificare, il kernel fa in modo che venga generato un timer interrupt quando il quanto di tempo è scaduto.

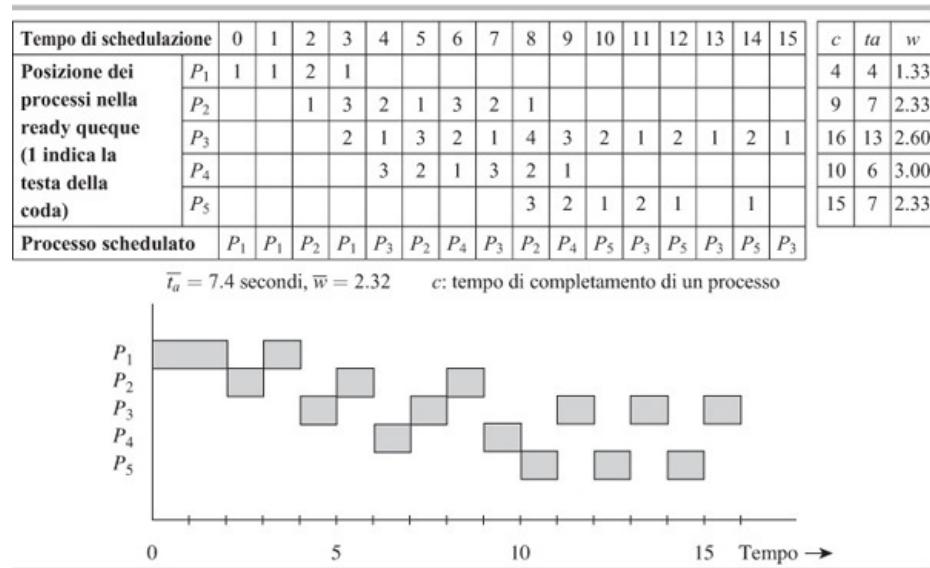
La politica RR fornisce servizi confrontabili a tutti i processi CPU-bound. Questa caratteristica si riflette in valori approssimativamente uguali dei loro completamenti pesati. Il valore attuale del completamento pesato di un processo dipende dal numero di processi presenti nel sistema. I completamenti pesati forniti ai processi che eseguono operazioni di I/O dovrebbero dipendere dalla durata delle operazioni di I/O. La politica

RR non contribuisce alle misure delle prestazioni del sistema come il throughput poiché non attribuisce un trattamento privilegiato ai processi brevi. L'esempio seguente illustra le prestazioni dello scheduling RR.

#### Esempio 7.4 - Scheduling Round-Robin (RR)

Uno scheduler round-robin mantiene una coda di processi in stato *Ready* e seleziona semplicemente il primo processo in coda. Il processo in esecuzione è sospeso (preempted) quando il quanto di tempo termina e viene posto in fondo alla coda. Si assume che un nuovo processo che è ammesso nel sistema nello stesso istante in cui un processo è sospeso (preempted) sarà inserito nella coda prima del processo sospeso.

La [Figura 7.5](#) riassume il funzionamento dello scheduler RR con  $\delta = 1$  secondo per i cinque processi della [Tabella 7.2](#). Lo scheduler prende decisioni di scheduling ogni secondo. L'istante in cui viene presa una decisione è mostrato nella prima riga della tabella nella metà superiore della [Figura 7.5](#). Le prime cinque righe mostrano le posizioni dei cinque processi nella coda dei processi pronti (*Ready*). Una casella bianca indica che il processo non è nel sistema in quel determinato istante. L'ultima riga mostra il processo selezionato dallo scheduler; ossia il processo che occupa la prima posizione nella coda dei processi pronti. Consideriamo la situazione a 2 secondi: la coda dello scheduling contiene  $P_2$  seguito da  $P_1$ . Di conseguenza,  $P_2$  viene schedulato. Il processo  $P_3$  arriva al terzo secondo, e viene inserito nella coda.  $P_2$  è sospeso (preempted) al terzo secondo e viene inserito nella coda. Quindi, la coda ha il processo  $P_1$  seguito da  $P_3$  e  $P_2$ , dunque  $P_1$  viene schedulato.



**Figura 7.5** Scheduling che utilizza la politica round-robin con time-slicing (RR).

I tempi di completamento e di completamento pesato dei processi sono mostrati nella parte destra della tabella. La colonna  $c$  mostra i tempi di completamento. I tempi di completamento e i completamenti pesati sono inferiori a quelli ottenuti con politiche nonpreemptive, discusse nel [Paragrafo 7.2](#) perché il tempo di CPU è condiviso tra molti processi a causa del time-slicing. Si può osservare che i processi  $P_2$ ,  $P_3$  e  $P_4$ , che arrivano all'incirca allo stesso tempo, ottengono approssimativamente gli stessi tempi di completamento pesato.  $P_4$  riceve il peggior completamento pesato perché, durante la maggior parte della sua vita, è uno dei tre processi presenti nel sistema.  $P_1$  riceve il miglior completamento pesato perché nessun altro processo è presente nel sistema durante la parte iniziale della sua esecuzione. Quindi i tempi di completamento pesato dependono dal carico nel sistema.

Come discusso nel [Capitolo 3](#), se un sistema contiene  $n$  processi, ogni sottorichiesta di un processo impiega esattamente  $\delta$  secondi, e il carico (overhead) per la decisione di scheduling è  $\sigma$ , il tempo di risposta ( $rt$ ) per una sottorichiesta è  $n \times (\sigma + \delta)$ . Comunque, la relazione tra  $\delta$  e  $rt$  è più complessa di questa. Primo, alcuni processi saranno bloccati per I/O o per attendere azioni dell'utente, così il tempo di risposta sarà controllato dal

numero di processi attivi piuttosto che da  $n$ . Secondo, se una richiesta necessita di un tempo di CPU maggiore di  $\delta$  secondi, dovrà essere schedulato più di una volta prima che possa produrre una risposta. Di conseguenza per valori piccoli di  $\delta$ ,  $rt$  per una richiesta può essere maggiore per valori più piccoli di  $\delta$ . L'esempio seguente illustra tale aspetto.

#### Esempio 7.5 - Variazione del tempo di risposta nello scheduling RR

Un SO contiene 10 processi identici iniziati allo stesso istante. Ogni processo riceve 15 identiche sottorichieste, e ogni sottorichiesta utilizza 20 ms di tempo di CPU. Una sottorichiesta è seguita da un'operazione di I/O che utilizza 10 ms. Il sistema usa 2 ms per lo scheduling della CPU. Per  $\delta \geq 20$  ms, la prima sottorichiesta del primo processo riceve un tempo di risposta di 22 ms e la prima sottorichiesta dell'ultimo processo riceve un tempo di risposta di 220 ms. Di conseguenza, il tempo di risposta medio è 121 ms. Una successiva sottorichiesta di un qualsiasi processo riceve un tempo di risposta di  $10 \times (2 + 20) - 10$  ms = 210 ms perché il processo impiega 10 ms nell'attesa di un I/O prima di ricevere la prossima sottorichiesta. Per  $\delta = 10$  ms, una sottorichiesta sarebbe sospesa (preempted) dopo 10 ms. Una volta schedulata di nuovo, sarebbe eseguita per 10 ms e produrrebbe i risultati. Quindi, il tempo di risposta per il primo processo è  $10 \times (2 + 10) + (2 + 10) = 132$  ms, e quello per l'ultimo processo è  $10 \times (2 + 10) + 10 \times (2 + 10) = 240$  ms. Una successiva sottorichiesta riceve un tempo di risposta di  $10 \times (2 + 10) + 10 \times (2 + 10) - 10 = 230$  ms. La Figura 7.6 riassume le prestazioni del sistema per valori differenti di  $\delta$ . Come atteso, la durata della schedulazione e il carico (overhead) sono maggiori per valori inferiori di  $\delta$ . Il grafico in Figura 7.6 illustra la variazione del tempo di risposta medio al secondo e successive sottorichieste per differenti valori di  $\delta$ . Si noti che il tempo di risposta è maggiore quando  $\delta$  è 5 ms piuttosto che quando vale 10 ms.

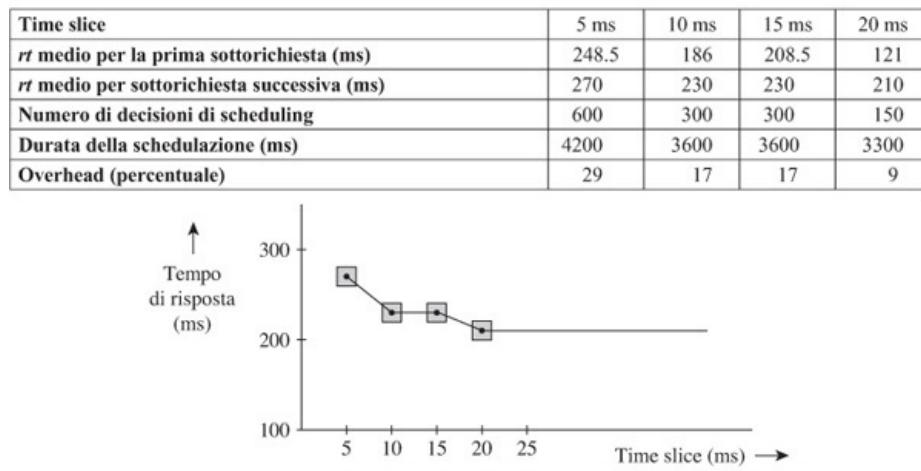


Figura 7.6 Prestazioni dello scheduling RR per differenti valori di  $\delta$ .

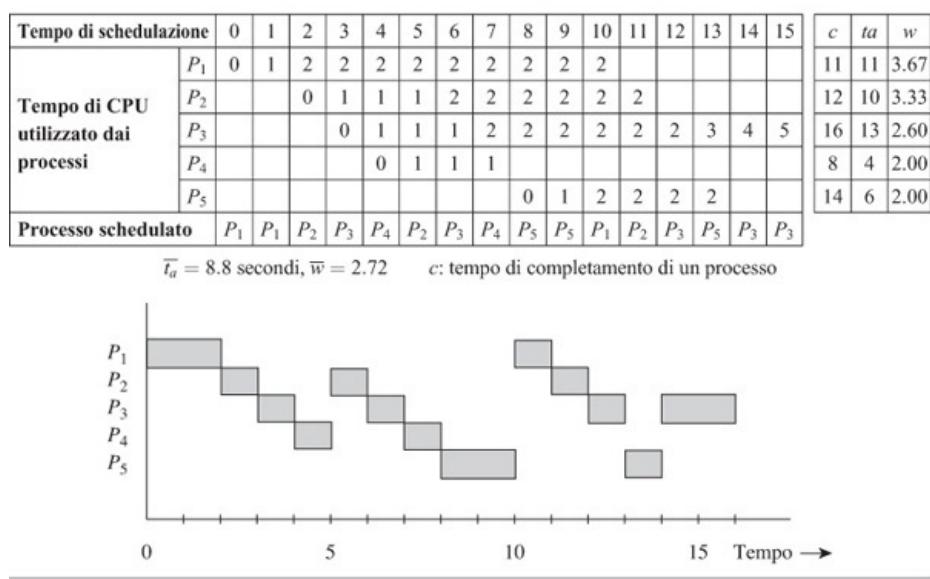
#### 7.3.2 Scheduling Least Completed Next (LCN)

La politica LCN seleziona il processo che ha utilizzato il minimo tempo di CPU. Così, la natura di un processo, sia CPU-bound che I/O-bound e la sua richiesta di tempo di CPU non influenzano la sua evoluzione nel sistema. Secondo la politica LCN, tutti i processi operano approssimativamente eguali progressi in termini di tempo di CPU utilizzato, cioè questa politica garantisce che processi brevi finiranno prima di processi lunghi. In ultimo, comunque, questa politica ha il noto svantaggio di generare la starvation dei processi lunghi. Inoltre, trascura i processi esistenti quando arrivano nuovi processi nel sistema. Così anche i processi non troppo lunghi soffrono di starvation o di tempi lunghi di completamento.

#### Esempio 7.6 - Scheduling Least Completed Next (LCN)

L'implementazione della politica di scheduling LCN per i cinque processi riportati in Tabella 7.2 è riassunta in Figura 7.7. Le righe centrali della tabella, nella metà superiore della figura, mostrano la quantità di tempo di CPU già utilizzato da un processo. Lo scheduler analizza questa informazione e seleziona il processo che ha

utilizzato la minor quantità di tempo di CPU. In caso di uguaglianza (tie), seleziona il processo che non è stato elaborato per il periodo di tempo più lungo. I tempi di completamento e di completamento pesato dei processi sono mostrati nella metà di destra della tabella.



**Figura 7.7** Scheduling utilizzando la politica least completed next (LCN).

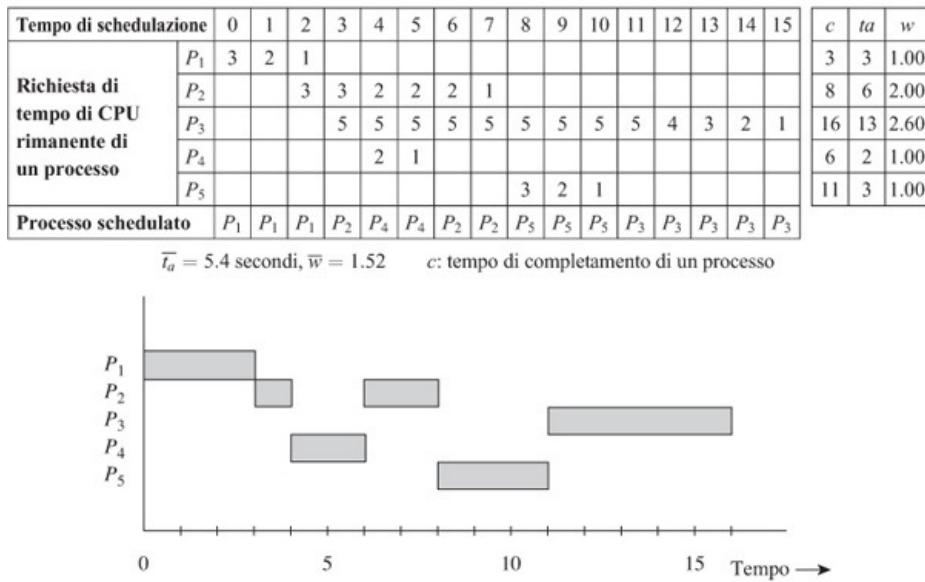
Si può vedere che l'elaborazione di  $P_1$ ,  $P_2$  e  $P_3$  è ritardata perché arrivano nuovi processi e ottengono servizi di CPU prima che questi processi possano fare ulteriori progressi. La politica LCN fornisce tempi di completamento e completamento pesato più ridotti di quelli forniti dalla politica RR (Esempio 7.4) e dalla politica STG (da discutere successivamente) perché privilegia i nuovi processi rispetto a quelli esistenti nel sistema finché i nuovi processi non li raggiungono in termini di utilizzo di CPU; per esempio, privilegia  $P_5$  su  $P_1$ ,  $P_2$  e  $P_3$ .

### 7.3.3 Scheduling Shortest Time to Go (STG)

La politica "shortest time to go" seleziona un processo i cui requisiti in termini di tempo di CPU rimanente sono i minimi del sistema. È una versione preemptive della politica shortest job first (SJF) del Paragrafo 7.2, privilegia i processi più brevi rispetto a quelli lunghi e fornisce un buon throughput. Inoltre, la politica STG favorisce un processo che sta per terminare rispetto a processi brevi che arrivano nel sistema. Questa caratteristica aiuta a migliorare i tempi di completamento e i completamenti pesati dei processi. Poiché è analoga alla politica SJF, i processi lunghi possono andare incontro a starvation.

#### Esempio 7.7 - Scheduling Shortest Time to Go (STG)

La Figura 7.8 riassume le prestazioni della politica di scheduling STG per i cinque processi mostrati in Tabella 7.2. L'elemento usato da questa politica di scheduling è il tempo di CPU necessario per il completamento di ciascun processo. In caso di uguaglianza (tie), seleziona il processo che non è stato elaborato per il periodo di tempo più lungo. L'esecuzione di  $P_3$  è ritardata perché  $P_2$ ,  $P_4$  e  $P_5$  richiedono minor tempo di CPU di esso.



**Figura 7.8** Scheduling utilizzando la politica shortest time to go (STG).

## 7.4 Scheduling in pratica

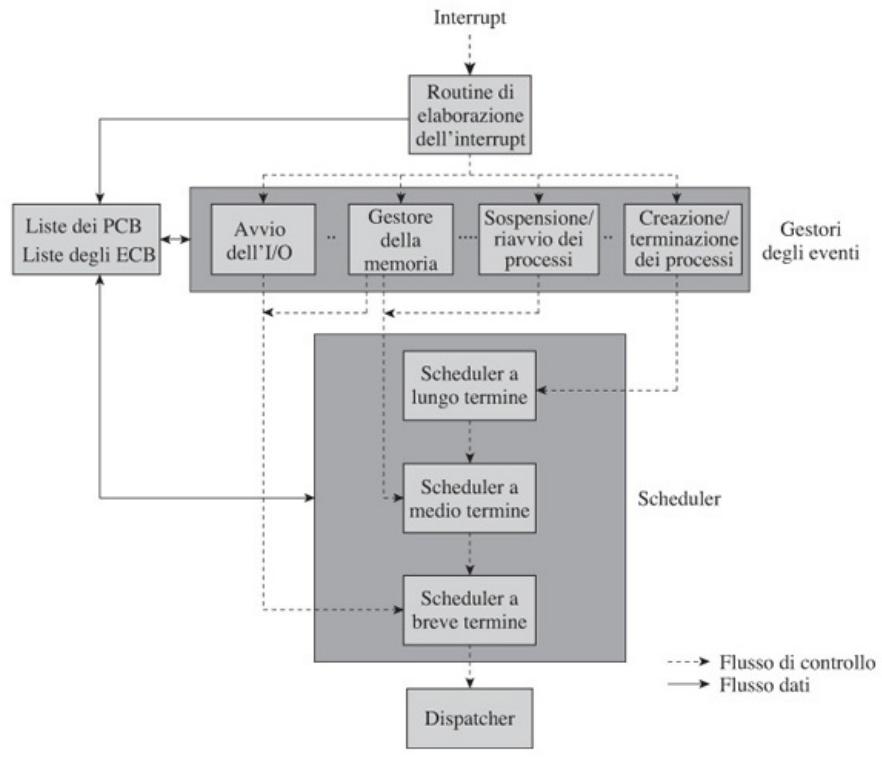
Per fornire una giusta combinazione di prestazione di sistema e servizio utente, un sistema operativo deve adattare il suo funzionamento alla natura e al numero delle richieste utente e alla disponibilità delle risorse. Un unico scheduler che utilizza una politica di scheduling classica non può rispondere a tutte queste necessità efficacemente. Di conseguenza, un moderno SO impiega *diversi* scheduler - fino a tre scheduler, come vedremo più avanti - e alcuni di questi scheduler possono utilizzare una *combinazione* di diverse politiche di scheduling.

### 7.4.1 Scheduler a lungo, medio e breve termine

Questi scheduler eseguono le seguenti funzioni:

- *scheduler a lungo-termine*: decide quando ammettere allo scheduling un processo creato di recente, in base alla sua natura (se CPU-bound o I/O-bound) e in base alla disponibilità di risorse come le strutture dati del kernel e lo spazio su disco per lo swapping;
- *scheduler a medio-termine*: decide quando scaricare un processo dalla memoria e quando ricaricarlo, affinché un numero sufficiente di processi *Ready* esista in memoria;
- *scheduler a breve-termine*: decide quale processo *Ready* è il prossimo elaborato dalla CPU e per quanto.

Lo *scheduler a breve termine* è quello che attualmente seleziona un processo per l'elaborazione. Di conseguenza è anche detto *scheduler di processo*, o semplicemente *scheduler*. La **Figura 7.9** mostra una panoramica dello scheduling e delle relative azioni. Come discusso nei Paragrafi 2.3 e 5.2.2, il funzionamento del kernel è interrupt-driven. Ogni evento che richiede l'attenzione del kernel causa un interrupt. La routine di elaborazione dell'interrupt esegue una funzione di salvataggio del contesto e invoca un gestore di eventi. Il gestore di eventi (event handler) analizza l'evento e cambia lo stato del processo, se qualcuno è coinvolto, per poi invocare lo scheduler a lungo termine, medio termine o breve termine, a seconda delle necessità. Per esempio, il gestore di eventi che crea un nuovo processo invoca lo scheduler a lungo termine, i gestori di eventi per la sospensione e il ripristino di processi (Paragrafo 5.2.1.1) invocano lo scheduler a medio termine e il gestore della memoria può invocare lo scheduler a medio termine se è esaurita la memoria. La maggior parte dei gestori di eventi invocano direttamente lo scheduler a breve termine.



**Figura 7.9** Gestione degli eventi e schedulazione.

### **Scheduling a lungo termine**

Lo scheduler a lungo termine può ritardare l'ammissione di una richiesta per due motivi: non riesce ad allocare risorse sufficienti quali strutture dati del kernel o dispositivi di I/O per una richiesta in arrivo, oppure potrebbe accadere che l'ammissione di una richiesta influenzi in qualche modo la prestazione di sistema; per esempio, se il sistema attualmente contiene un gran numero di richieste di tipo CPU-bound, lo scheduler può rimandare l'ammissione di una nuova richiesta di tipo CPU-bound, mentre potrebbe ammettere subito una nuova richiesta di tipo I/O-bound.

Lo scheduling a lungo termine veniva utilizzato negli anni 1960 e 1970 per lo scheduling di job perché i sistemi informatici avevano risorse limitate, quindi, lo scheduling a lungo-termine era necessario per decidere *se* un processo poteva essere attivato all'istante corrente. Continua a essere importante in sistemi operativi con risorse limitate. Inoltre, viene utilizzato in sistemi in cui le richieste hanno una scadenza, o un insieme di richieste sono ripetute con periodicità nota, per decidere *quando* un processo dovrebbe essere attivato per rispettare i requisiti di risposta delle applicazioni. In altri sistemi operativi, lo scheduling a lungo-termine non è rilevante.

### **Scheduling a medio termine**

Lo scheduling a medio termine effettua il mapping di un gran numero di richieste che sono state ammesse nel sistema in un numero inferiore di richieste che possono risiedere in memoria a ogni istante. Di conseguenza, pone la sua attenzione nel rendere disponibili un sufficiente numero di processi *Ready* allo scheduler a breve termine sospendendo o riattivando i processi. Lo scheduler a medio termine decide quando scaricare un processo dalla memoria e quando ricaricarlo in memoria, cambia lo stato del processo opportunamente, e inserisce il suo blocco di controllo di processo (PCB) nella lista di PCB. Le operazioni di carico (swapping-in) e scarico (swapping-out) sono eseguite dal gestore della memoria (memory manager).

Un processo può essere sospeso quando un utente richiede una sospensione, quando il sistema esaurisce la memoria libera, o quando accade che la CPU non possa essere allocata al processo nel breve. Nei sistemi time-sharing, i processi in stato *bloccato* (blocked) o *ready* sono candidati alla sospensione (Figura 5.5). La decisione di riattivare un processo è più complessa: lo scheduler a medio-termine considera la posizione

occupata da un processo nella lista di scheduling, stima quando probabilmente è selezionata come successiva, e la carica in quel momento.

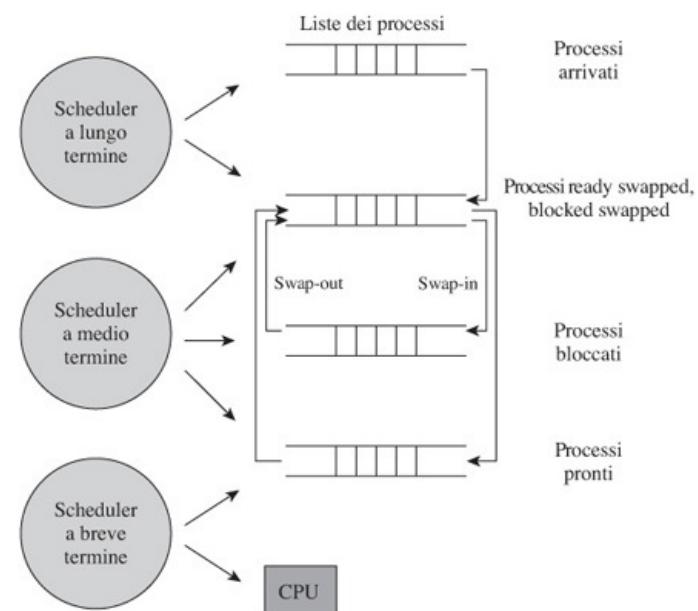
### Scheduling a breve termine

Lo scheduling a breve termine riguarda l'uso effettivo della CPU. Esso seleziona un processo da una lista di processi *ready* e lo manda al meccanismo di dispatching. Può, inoltre, decidere quanto tempo è consentito al processo l'utilizzo della CPU e istruisce il kernel affinché produca un timer interrupt di conseguenza.

L'Esempio 7.8 illustra lo scheduling a lungo, medio e breve termine in un SO time-sharing.

#### Esempio 7.8 - Time-sharing

La Figura 7.10 illustra lo scheduling in un sistema operativo time-sharing. Lo scheduler a lungo-termine ammette un processo quando le risorse del kernel come i blocchi di controllo (control block), lo spazio su disco per lo swap, e altre risorse quali i dispositivi di I/O - o reali o virtuali - possono essere allocati. Il kernel copia il codice del processo nello spazio di swap e aggiunge il processo alla lista dei processi scaricati (swapped-out).

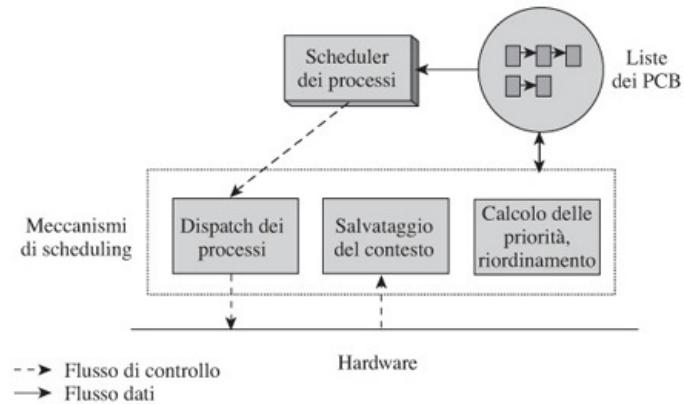


**Figura 7.10** Scheduling a lungo, medio e breve termine in un sistema time-sharing.

Lo scheduler a medio termine controlla lo swapping dei processi e decide quando spostare i processi tra le liste *ready swapped* e *ready* e tra le liste *blocked swapped* e *blocked* (Figura 5.5). Ogni volta che la CPU è libera, lo scheduler a breve termine seleziona un processo dalla lista *ready* per l'esecuzione. Il meccanismo di dispatching inizia o ripristina il funzionamento del processo selezionato sulla CPU. Un processo può passare tra gli scheduler a medio e a breve termine più volte come risultato di swapping.

## 7.4.2 Strutture dati e meccanismi di scheduling

La Figura 7.11 è un diagramma schematico dello scheduler di processo. Usa diverse liste di PCB la cui organizzazione e utilizzo dipendono dalla politica di scheduling. Lo scheduler dei processi seleziona un processo e passa il suo id al meccanismo di dispatching dei processi. Il meccanismo di dispatching dei processi carica i contenuti dei due campi PCB - il program status word (PSW) e i registri general-purpose (GPRs) - nella CPU per l'operazione di ripristino del processo selezionato. Così, il meccanismo di dispatching si interfaccia con lo scheduler da un lato e con l'hardware dall'altro lato.



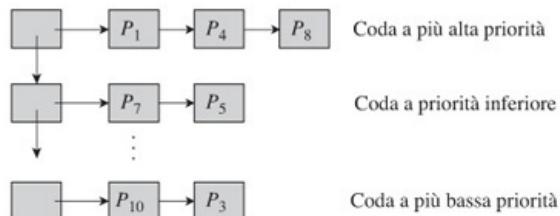
**Figura 7.11** Uno schema dello scheduler dei processi.

Il meccanismo di salvataggio del contesto è una parte della routine di elaborazione degli interrupt. Quando si verifica un interrupt, viene invocato il meccanismo di salvataggio del contesto per salvare il PSW e il GPRs del processo interrotto. Il calcolo della priorità e il meccanismo di riordino ricalcola la priorità delle richieste e riordina le liste PCB per rispecchiare le nuove priorità. Questo meccanismo è invocato esplicitamente dallo scheduler quando necessario oppure periodicamente. Le sue azioni precise dependono dalla politica di scheduling applicata.

Una domanda valida per tutti gli scheduler: che cosa dovrebbe fare lo scheduler se non ci sono processi *ready*? In tal caso, non c'è "lavoro" per la CPU; comunque, la CPU deve essere pronta a gestire un qualsiasi interrupt che uno dei processi *bloccato* (*blocked*) può attivare. A tal fine, tipicamente, un kernel esegue un *idle loop*, cioè un loop senza fine che non contiene istruzioni. Quando un interrupt determina una transizione da *blocked* → *ready* per un processo, è necessario ancora eseguire lo scheduling e quel processo deve essere schedulato; l'esecuzione di un idle loop impiega energia. Nel Paragrafo 7.4.9 parleremo di soluzioni alternative che risparmiano energia quando non ci sono processi *ready* nel sistema.

### 7.4.3 Scheduling basato su priorità

La [Figura 7.12](#) mostra una soluzione efficiente per i dati di scheduling relativamente allo scheduling basato su priorità. Viene mantenuta una lista separata di processi *ready* per ogni valore di priorità; questa lista è organizzata come una coda di PCB, in cui un PCB punta al PCB del successivo processo in coda. La testa della coda contiene due puntatori: uno punta al PCB del primo processo nella coda, e l'altro punta alla testa della coda per la priorità inferiore successiva. Lo scheduler scandisce le teste in ordine di priorità decrescente e seleziona il primo processo nella prima coda non vuota che trova. In questo modo, il carico (overhead) dello scheduling dipende dal numero di priorità distinte, piuttosto che dal numero di processi *ready*.



**Figura 7.12** Code dei processi ready nello scheduling basato su priorità.

Lo scheduling basato su priorità può condurre alla starvation dei processi a bassa priorità. Come discusso nel Paragrafo 7.1.2, la tecnica dell'aging dei processi, che incrementa la priorità di un processo *ready* se esso non viene selezionato per un certo periodo di tempo, può essere utilizzata per superare la starvation. In questo schema, le

priorità dei processi sarebbero *dinamiche*, così il PCB di un processo dovrebbe essere spostato tra diverse code dei processi pronti come mostrato in [Figura 7.12](#).

La starvation nello scheduling basato su priorità può condurre anche a una situazione chiamata *inversione di priorità*. Consideriamo un processo ad alta priorità che necessita di una risorsa attualmente allocata a un processo a bassa priorità. Se il processo a bassa priorità va incontro a starvation, non può utilizzare la risorsa e la rilascia. Di conseguenza, il processo ad alta priorità rimane bloccato (blocked) indefinitamente. Questa situazione si risolve attraverso il *priority inheritance protocol*, che temporaneamente aumenta la priorità del processo a bassa priorità, che tiene la risorsa, al valore di priorità del processo ad alta priorità che necessita della risorsa. Il processo, tenendo la risorsa, può ora ottenere la CPU, utilizzare la risorsa e rilasciarla. Quando il processo rilascia la risorsa, il kernel riporta la sua priorità al precedente valore.

#### 7.4.4 Scheduling Round-Robin con Time-Slicing

Lo scheduling Round-Robin può essere implementato attraverso un'unica lista di PCB di *processi ready*. Questa lista è organizzata come una coda. Lo scheduler rimuove sempre il primo PCB dalla coda e seleziona il processo a cui si riferisce. Se la time slice si esaurisce, il PCB del processo è posto alla fine della coda. Se un processo comincia un'operazione di I/O, il suo PCB è aggiunto alla fine della coda quando la sua operazione di I/O termina. Così il PCB di un processo *ready* si sposta verso la testa della coda finché il processo viene selezionato.

#### 7.4.5 Scheduling multilivello

La politica di scheduling *multilivello* combina lo scheduling basato su priorità e quello round-robin per fornire una buona combinazione tra prestazione del sistema e tempi di risposta. Uno scheduler multilivello utilizza varie code di processi pronti. Una priorità e una time slice sono associati a ogni coda, e lo scheduling round-robin con time-slicing è eseguito all'interno di essa. La coda ad alto livello di priorità ha una time slice piccola associata a essa, che assicura buoni tempi di risposta per i processi in questa coda, mentre la coda a un livello di priorità basso ha una time slice grande, che assicura un carico ridotto per lo switching di processo. Un processo in testa a una coda è selezionato solo se le code per tutti i livelli di priorità più alta sono vuote. Lo scheduling è con diritto di prelazione (preemptive), quindi un processo è prelazionato quando un nuovo processo è aggiunto a una coda a un livello di priorità maggiore. Come nello scheduling round-robin con time-slicing, quando un processo fa una richiesta di I/O, o è scaricato (swapped out), e il suo PCB viene rimosso dalla coda dei processi pronti. Quando l'operazione di I/O termina, il processo è ricaricato in memoria (swapped in), e il suo PCB è aggiunto alla fine di quella coda dei pronti dove era collocato precedentemente.

Per avere un beneficio dalle caratteristiche dello scheduling multilivello, il kernel pone i processi ad alta interazione nella coda a livello di priorità più alto. La breve time slice associata a questa coda è adeguata per questi processi, quindi ricevono buoni tempi di risposta (Equazione 3.1). I processi moderatamente interattivi sono posti nella coda a un livello di priorità medio, e ricevono time slice maggiori. I processi non interattivi sono posti in una coda a uno dei livelli di priorità più bassi. Questi processi ricevono un time slice grande, che riduce il carico (overhead) di scheduling.

##### Esempio 7.9 - Scheduling multilivello

La [Figura 7.12](#) illustra le code dei processi ready in uno scheduler multilivello. I processi  $P_7$  e  $P_5$  hanno un time slice maggiore rispetto ai processi  $P_1$ ,  $P_4$  e  $P_8$ .

Comunque, sono eseguiti solo quando  $P_1$ ,  $P_4$  e  $P_8$  sono bloccati. I processi  $P_{10}$  e  $P_3$ , invece, sono eseguiti solo quando tutti gli altri processi nel sistema sono bloccati. In tale modo, i due processi andrebbero incontro a starvation se questa situazione non fosse rara.

La politica di scheduling multilivello utilizza priorità statiche. Di conseguenza, eredita le fondamentali lacune dello scheduling basato su priorità impiegato nei sistemi multiprogrammati: un processo è classificato *a priori* in processo di tipo CPU-bound o in I/O-bound per l'assegnazione della priorità. Se fosse erroneamente classificato, un processo I/O-bound potrebbe ricevere una bassa priorità, che andrebbe a scapito sia del servizio utente che della prestazione di sistema, oppure un processo CPU-bound potrebbe ricevere un'elevata priorità, che potrebbe andare a scapito della prestazione

del sistema.

A causa dell'uso di priorità statiche, la politica di scheduling multilivello non può gestire anche un cambiamento nel comportamento computazionale o di I/O di un processo, non può prevenire la starvation dei processi a bassi livelli di priorità (vedi Esempio 7.9), e non può impiegare il protocollo di ereditarietà delle priorità per superare l'inversione delle priorità (Paragrafo 7.4.3). Tutti questi problemi sono risolti dalla politica di scheduling multilivello adattivo.

### ***Scheduling multilevello adattivo***

Nello scheduling multilevello *adattivo*, anche detto *scheduling multilivello con feedback*, lo scheduler varia la priorità di un processo cosicché il processo riceve una time slice compatibile con la sua richiesta di tempo di CPU. Lo scheduler determina il "corretto" livello di priorità per un processo osservando il suo recente uso di CPU e di I/O e sposta il processo a questo livello. In questo modo, un processo che è I/O-bound durante una fase e CPU-bound durante un'altra fase riceverà sempre un'appropriata priorità e time slice. Questa caratteristica elimina i problemi dello scheduling multilivello descritti precedentemente.

CTSS, un SO time-sharing per gli IBM 7094 negli anni '60, è un esempio ben noto di scheduling multilevello adattivo. Il sistema utilizzava una struttura di priorità basata su otto livelli, con i livelli numerati da 0 a 7, con 0 il livello di priorità più alto e 7 il livello di priorità più basso. Il livello numero  $n$  aveva una time slice di  $0.5 \times 2^n$  secondi di CPU associati a esso. All'inizio, ogni processo utente era posto al livello 2 o 3 in base alla sua richiesta di memoria e poi avanzava o retrocedeva nella struttura delle priorità secondo le seguenti regole: se un processo utilizzava completamente la time slice al suo livello corrente di priorità (ovvero non iniziava un'operazione di I/O), veniva retrocesso al successivo livello con numerazione maggiore; invece, se un processo spendeva più di un minuto in stato *ready* al suo livello corrente di priorità senza ottenere alcun servizio di CPU, veniva avanzato al successivo livello con numerazione inferiore. Inoltre, qualsiasi processo che eseguiva I/O sul terminale utente avanzava al livello 2. Successivamente, doveva essere riportato al "giusto" livello di priorità tramite possibili diminuzioni.

### **7.4.6 Scheduling fair share**

Una criticità comune a tutte le politiche di scheduling discusse, è quella per cui tutte cercano di fornire servizi equi ai processi, piuttosto che agli utenti o alle loro applicazioni. Se le applicazioni creano un diverso numero di processi, un'applicazione che impiega più processi è probabile che riceva maggiore attenzione dalla CPU rispetto a una che utilizza meno processi.

La nozione di *fair share* risponde a questa esigenza. Una fair share è la frazione del tempo di CPU che dovrebbe essere dedicata a un gruppo di processi che appartengono allo stesso utente o alla stessa applicazione, assicurando un equo utilizzo della CPU da parte degli utenti o delle applicazioni. La percentuale attuale del tempo di CPU ricevuta da un gruppo di processi può differire dalla fair share del gruppo se tutti i processi in qualche gruppo sono inattivi. Per esempio, consideriamo cinque gruppi di processi,  $G_1 - G_5$ , ognuno richiedente un 20 percento di tempo di CPU. Se tutti i processi in  $G_1$  fossero *blocked*, i processi di ognuno degli altri gruppi dovrebbero avere il 25 percento del tempo di CPU disponibile, così il tempo di CPU non andrebbe perso. Cosa dovrebbe fare lo scheduler quando i processi di  $G_1$  diventano attivi dopo un certo tempo? Dovrebbe attribuirgli solo il 20 per cento del tempo di CPU, perché quella è la loro fair share del tempo di CPU, oppure dovrebbe attribuirgli tutto il tempo di CPU disponibile finché il loro attuale consumo di CPU dall'inizio diventi il 20 per cento? Lo scheduling a lotteria, che descriveremo nel seguito, e le politiche di scheduling utilizzate nei sistemi operativi Unix e Solaris (Paragrafo 7.6) differiscono nel modo in cui viene gestita tale situazione.

Lo *scheduling a lotteria* è una nuova tecnica proposta per la condivisione di una risorsa in maniera probabilisticamente equa. I "biglietti" della lotteria sono distribuiti a tutti i processi che condividono una risorsa in maniera tale che un processo ne riceve un numero proporzionale all'utilizzo della risorsa. Per esempio, consideriamo che a un processo vengono attribuiti cinque biglietti oltre i 100 biglietti iniziali se la sua parte (fair share) della risorsa fosse il 5 per cento. Quando la risorsa deve essere allocata, viene realizzata una lotteria tra i biglietti posseduti dai processi che necessitano attivamente della risorsa. Il processo che possiede il biglietto vincente ottiene allora l'allocazione della risorsa. Quanta parte delle risorse allocate per il processo dipende

quanto sia richiesta la risorsa. Lo scheduling a lotteria può essere utilizzato per lo scheduling fair share della CPU: i biglietti possono essere rilasciati alle applicazioni (o utenti) in base alla loro fair share del tempo di CPU. Un'applicazione può condividere i suoi biglietti tra i suoi processi in qualsiasi modo desideri. Per allocare una time slice di CPU, lo scheduler realizza una lotteria a cui partecipano solo i biglietti dei processi *ready*. Quando la time slice è di pochi millisecondi, questo metodo di scheduling fornisce equità anche su frazioni di secondi nel caso tutti i gruppi di processi fossero attivi.

#### 7.4.7 Prelazionabilità del kernel

La prelazionabilità del kernel gioca un ruolo fondamentale nell'efficacia di uno scheduler. Un kernel non prelazionabile può gestire un evento senza ulteriori interruzioni, così i gestori degli eventi (event handler) hanno un accesso mutuamente esclusivo alle strutture dati del kernel senza dover utilizzare tecniche di sincronizzazione per l'accesso ai dati. Comunque, se gli event handler presentano lunghi tempi di esecuzione, la non prelazionabilità causerebbe anche un'elevata latenza del kernel, poiché il kernel non potrebbe rispondere prontamente agli interrupt. Questa latenza, che potrebbe essere circa 100 ms in elaboratori con CPU lente, determina un significativo degrado dei tempi di risposta e un rallentamento del funzionamento del SO. Quando lo scheduling di un processo ad alta priorità è ritardato perché il kernel sta gestendo un evento legato a un processo a bassa priorità, si verifica una situazione analoga all'inversione di priorità. Rendere il kernel prelazionabile dovrebbe risolvere il problema. Ora, lo scheduling sarebbe eseguito più spesso, così un processo ad alta priorità che viene attivato da un interrupt sarebbe eseguito prima.

#### 7.4.8 Scheduling: euristiche

Gli scheduler nei moderni sistemi operativi utilizzano molte euristiche per ridurre il loro carico (overhead) e fornire un buon servizio utente. Queste euristiche impiegano due tecniche principali:

- uso del quanto di tempo (time quantum);
- variazione della priorità di processo.

Un *quanto di tempo* è il limite di tempo di CPU che un processo può utilizzare su un intervallo di tempo. Viene impiegato come segue: a ogni processo è assegnata una priorità e un quanto di tempo. Un processo viene selezionato in base alla sua priorità, nel caso non abbia esaurito il suo quanto di tempo. Mentre opera, la quantità di tempo di CPU usata è sottratta dal suo quanto di tempo. Dopo che un processo ha esaurito il suo quanto di tempo, non può essere considerato per lo scheduling finché il kernel non gli concede un altro quanto di tempo, il che accade solo quando tutti i processi attivi hanno esaurito i loro quanti. In questo modo, il quanto di tempo di un processo controllerebbe la condivisione del tempo di CPU utilizzato; quindi potrebbe essere utilizzato per implementare lo scheduling fair share.

La priorità di processo potrebbe essere variata per raggiungere diversi obiettivi. La priorità di un processo potrebbe essere incrementata durante l'esecuzione di una chiamata di sistema (system call), cosicché completerà rapidamente l'esecuzione della chiamata, rilascerà qualsiasi risorsa del kernel allocata allo scopo, e lascerà il kernel. Questa tecnica migliorerebbe la risposta per gli altri processi che sono in attesa per le risorse del kernel tenute dal processo che esegue la chiamata di sistema. L'ereditarietà della priorità potrebbe essere implementata elevando la priorità di un processo che possiede una risorsa al valore della più alta priorità di un processo in attesa di quella risorsa.

La priorità di processo può essere modificata per caratterizzare più accuratamente la natura di un processo. Quando il kernel inizia un nuovo processo, non ha modo di sapere se il processo è I/O-bound oppure CPU-bound, così assegna una priorità di default al processo. Mentre il processo è in esecuzione, il kernel modifica la sua priorità in base al suo comportamento utilizzando un'euristica del seguente tipo: quando il processo viene attivato dopo un periodo di blocking, la sua priorità può essere elevata in dipendenza della causa del blocking. Per esempio, se è stato bloccato a causa di un'operazione di I/O, la sua priorità dovrebbe essere aumentata per fornirgli un migliore tempo di risposta. Se fosse bloccato per un input da tastiera, attende che l'utente risponda per un tempo lungo, così la sua priorità subisce un ulteriore incremento. Se un processo ha esaurito la

time slice completamente, la sua priorità può essere ridotta perché è più CPU-bound di quanto era stato considerato precedentemente.

#### 7.4.9 Gestione del consumo di energia

Quando nessun processo *ready* è presente, il kernel pone la CPU in un *ciclo inattivo o idle* (Paragrafo 7.4.2). Questa soluzione consuma energia eseguendo istruzioni inutili. Nei sistemi in cui l'energia è una risorsa da non sprecare, come i sistemi embedded e mobili, è essenziale evitare questo spreco di energia.

Per rispondere a questo requisito, gli elaboratori forniscono modalità speciali per la CPU. In una di queste modalità, la CPU non esegue istruzioni, risparmiando energia; e può, tuttavia, accettare gli interrupt, consentendo in tal modo di riprendere la normale operatività quando necessario. Useremo genericamente il termine *sleep mode* della CPU per tali modalità. Alcuni elaboratori forniscono diversi sleep mode. Nel "light" sleep mode, la CPU semplicemente ferma l'esecuzione delle istruzioni. Nell'"heavy" sleep mode, la CPU non solo ferma l'esecuzione delle istruzioni, ma effettua ulteriori passi per ridurre il consumo di energia, per esempio, rallentando il clock e disconnettendo la CPU dal bus di sistema. Idealmente, il kernel dovrebbe mettere la CPU nella massima modalità di sleep possibile quando il sistema non ha processi nello stato *ready*. Comunque, una CPU impiega un tempo più lungo per "risvegliarsi" (wake up) da una modalità heavy piuttosto che da una light, quindi il kernel deve fare un compromesso. Si comincia mettendo la CPU in modalità light sleep mode. Se nessun processo diventa *ready* per un tempo abbastanza lungo, si mette la CPU in una modalità sleep più heavy, e così via. In questo modo, si raggiunge un compromesso tra la necessità di risparmio energetico e quella di reattività del sistema.

I sistemi operativi come Unix e Windows applicano la gestione dell'energia a tutti i dispositivi. Tipicamente, un dispositivo è posto in uno stato di consumo a bassa energia se nel suo stato attuale di consumo di energia è stato dormiente per qualche tempo. Vengono fornite, inoltre, agli utenti utility attraverso le quali possono configurare lo schema di gestione del consumo di energia utilizzato dal SO.

### 7.5 Scheduling real-time

Lo scheduling real-time deve gestire due particolari vincoli di scheduling contemporaneamente nel tentativo di rispettare le scadenze (o deadline) delle applicazioni: i processi di un'applicazione real-time possono essere processi interattivi, quindi la deadline di un'applicazione dovrebbe essere tradotta in opportune deadline per i processi, e possono essere periodici, tale che istanze differenti di un processo possono arrivare a intervalli fissati e tutte devono rispettare le loro deadline. L'Esempio 7.10 illustra questi vincoli; nel seguito di questo paragrafo, discutiamo le tecniche usate per gestirli.

#### Esempio 7.10 - Dipendenze e periodi nelle applicazioni real-time

Consideriamo una forma ristretta dell'applicazione real-time di logging dei dati dell'Esempio 5.1, in cui il *buffer* può ospitare un unico campione di dati. Poiché i campioni arrivano in numero di 500 al secondo, il requisito di risposta dell'applicazione è 1.99 ms. Di conseguenza, i processi *copia\_campione* e *registra\_campione* devono operare l'uno dopo l'altro e completare la loro operazione entro i 1.99 ms. Se il processo *registra\_campione* richiede 1.5 ms per la sua operazione, il processo *copia\_campione* ha una scadenza (deadline) di 0.49 ms dopo l'arrivo di un messaggio. Poiché un nuovo campione arriva ogni 2 ms, ognuno dei processi ha un periodo di 2 ms.

#### 7.5.1 Precedenze e schedulabilità di un processo

I processi di un'applicazione real-time interagiscono tra loro per assicurare che le loro azioni vengono eseguite nel giusto ordine (Paragrafo 6.1). Supponiamo per semplicità che tale interazione si verifichi solo all'inizio o alla fine di un processo. Ciò causa dipendenze tra i processi, che devono essere considerate nella determinazione delle scadenze e nello scheduling. Usiamo un *grafo di precedenza tra processi* (process precedence graph - PPG) per descrivere tali dipendenze tra processi.

Si dice che il processo  $P_i$  precede il processo  $P_j$  se l'esecuzione di  $P_i$  deve essere

completata prima che  $P_j$  possa iniziare la sua esecuzione. La notazione  $P_i \rightarrow P_j$  indica che il processo  $P_i$  precede direttamente il processo  $P_j$ . La relazione di precedenza è transitiva; per esempio,  $P_i \rightarrow P_j$  e  $P_j \rightarrow P_k$  implica che  $P_i$  precede  $P_k$ . La notazione  $P_i \xrightarrow{*} P_k$  è usata per indicare che il processo  $P_i$  precede direttamente o indirettamente  $P_k$ . Un *grafo di precedenza tra processi* è un grafo  $G \equiv (N, E)$  tale che  $P_i \in N$  rappresenta un processo, e un arco  $(P_i, P_j) \in E$  implica  $P_i \rightarrow P_j$ . Così, un percorso  $P_i, \dots, P_k$  in PPG implica  $P_i \xrightarrow{*} P_k$ . Un processo  $P_k$  è un discendente di  $P_i$  se  $P_i \xrightarrow{*} P_k$ . Nel Paragrafo 3.7 abbiamo definito *sistema real-time hard* quello che rispetta il requisito di risposta di un'applicazione real-time in maniera garantita, anche quando sono richieste azioni di fault tolerance. Questa condizione implica che il tempo richiesto dal SO per completare l'elaborazione di tutti i processi dell'applicazione non supera il requisito di risposta dell'applicazione. Dall'altra parte, un *sistema real-time soft* rispetta il requisito di risposta di un'applicazione solo in modo probabilistico, e non necessariamente tutte le volte. La nozione di *schedulabilità* aiuta a differenziare queste situazioni.

**Definizione 7.1 Schedulabilità** Una sequenza di decisioni di scheduling che consentono ai processi di un'applicazione di agire secondo le precedenze e rispettare il requisito di risposta dell'applicazione.

Lo *scheduling real-time* pone l'attenzione sull'implementazione della schedulabilità per un'applicazione, se ne esiste una. Consideriamo un'applicazione che effettua l'aggiornamento delle informazioni sulle partenze aeree su display a intervalli di 15 secondi. Consiste dei seguenti processi indipendenti, tra cui il processo  $P_5$  che gestisce una situazione eccezionale che accade raramente.

| Processo          | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|-------------------|-------|-------|-------|-------|-------|
| Tempo di servizio | 3     | 3     | 2     | 4     | 5     |

Una sequenza schedulabile per completare tutti i cinque processi in 15 secondi non esiste, quindi potrebbe verificarsi un superamento della scadenza (deadline overrun). In ogni caso, quando non è attivo il processo  $P_5$  sono possibili diversi schedule. Lo scheduler in un sistema real-time soft può utilizzare uno di questi.

La [Tabella 7.3](#) riassume tre approcci principali allo scheduling real-time. Nel seguito discutiamo le caratteristiche e proprietà di questi approcci allo scheduling.

| Approccio                     | Descrizione   |
|-------------------------------|---|
| Scheduling statico            | Uno schedule viene preparato <i>prima</i> che l'esecuzione dell'applicazione real-time cominci, tenendo conto delle interazioni tra processi, le periodicità, i vincoli sulle risorse e le scadenze.                      |
| Scheduling basato su priorità | L'applicazione real-time viene analizzata per attribuire appropriate <i>priorità</i> ai suoi processi. Durante l'esecuzione dell'applicazione viene adottato lo scheduling convenzionale basato su priorità.              |
| Scheduling dinamico           | Lo scheduling è eseguito quando <i>proviene</i> una richiesta di creazione di un processo. La creazione di un processo avviene solo se il requisito di risposta del processo può essere soddisfatto in maniera garantita. |

**Tabella 7.3** Approcci allo scheduling real-time.

### **Scheduling statico**

Come indica il nome, viene preparato un ordine di schedulazione prima che il sistema venga posto in esecuzione. Lo scheduler considera le precedenze tra processi, le periodicità, i vincoli sulle risorse e le possibilità di sovrapposizione tra le operazioni di I/O in alcuni processi con elaborazioni in altri processi. Questo scheduling è rappresentato in forma tabellare, in cui le righe indicano quando l'operazione di processi

differenti dovrebbe cominciare. Nessuna decisione di scheduling viene presa durante l'operatività del sistema. Il SO real-time semplicemente consulta la tabella e attiva i processi indicati. Lo scheduling statico ha un overhead trascurabile durante l'operatività del sistema. Comunque, è inflessibile e non può gestire problematiche come la tolleranza ai guasti (fault tolerance).

La dimensione della tabella di scheduling dipenderà dai periodi dei processi. Se tutti i processi hanno lo stesso periodo, o se i processi non sono periodici, la tabella di scheduling avrà tante righe pari al numero dei processi nell'applicazione. Questa schedulazione viene utilizzata ripetutamente durante l'operatività del sistema. Se le periodicità dei processi sono diverse, la lunghezza della schedulazione, che è necessario rappresentare nella tabella di scheduling, sarà il minimo comune multiplo delle periodicità di tutti i processi dell'applicazione.

### **Scheduling basato su priorità**

Un analista di sistema fa due considerazioni nell'assegnare le priorità ai processi: criticità e periodicità dei processi. Un processo con un periodo inferiore deve completare la sua operazione prima di un processo con un periodo maggiore, per cui deve avere una priorità più alta. Questo approccio ha i benefici e le controindicazioni normalmente associate all'uso delle priorità. Fornisce funzionalità di degrado graduale perché l'esecuzione di funzioni critiche dovrebbe continuare anche quando si verificano problemi. Comunque, incorre in overhead di scheduling durante il funzionamento.

### **Scheduling dinamico**

Nei sistemi che usano l'approccio a scheduling dinamico, lo scheduling è eseguito durante le operazioni di sistema. I sistemi multimediali, come il video on demand, utilizzano un approccio a scheduling dinamico in cui una decisione di scheduling viene presa quando arriva un processo. Una richiesta di inizio processo contiene informazioni come il requisito di risorse per il processo, il tempo di servizio, la scadenza o una specifica della qualità di servizio. Nel ricevere una tale richiesta, lo scheduler controlla se è possibile assegnare le risorse di cui necessita il processo, rispettare la scadenza o fornirgli la qualità di servizio richiesta. Crea il processo solo se questi controlli vengono superati.

Un altro approccio allo scheduling dinamico è ammettere processi all'esecuzione in maniera ottimistica. Secondo questo approccio, non si garantisce che i requisiti, quali la scadenza o la qualità di servizio, vengano rispettati. I sistemi soft real-time spesso seguono questo approccio.

## **7.5.2 Scheduling della deadline**

Per ogni processo possono essere specificati due tipi di deadline ( dette anche scadenze): la *scadenza d'inizio* (starting deadline), cioè il minimo istante entro cui le operazioni del processo devono cominciare, e la *scadenza di fine* (completion deadline), cioè il tempo entro il quale le operazioni del processo devono terminare. Consideriamo nel seguito solo scadenze di fine.

### **Stima della scadenza**

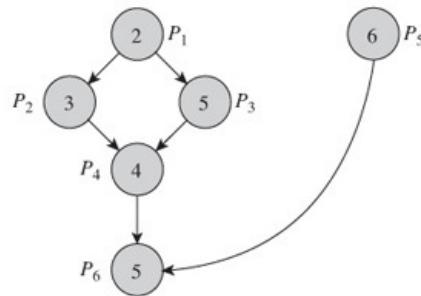
Un analista di sistema esegue un'analisi approfondita di un'applicazione real-time e dei suoi requisiti di risposta. Le scadenze per i processi singoli vengono determinate considerando le precedenze del processo e lavorando sulle richieste di risposta dell'applicazione. Di conseguenza, la scadenza di fine  $D_i$  (completion deadline) di un processo  $P_i$  è:

$$D_i = D_{\text{applicazione}} - \sum_{k \in \text{discendenti}(i)} x_k \quad (7.2)$$

dove  $D_{\text{applicazione}}$  è la scadenza dell'applicazione,  $x_k$  è il tempo di servizio del processo  $P_k$ , e  $\text{discendenti}(i)$  è l'insieme dei discendenti di  $P_i$  nel PPG, cioè l'insieme di tutti i processi che giacciono sullo stesso percorso tra  $P_i$  e il nodo di uscita del PPG. Pertanto, se la scadenza per un processo  $P_i$  è rispettata, tutti i processi che direttamente o indirettamente dipendono da  $P_i$  possono terminare entro la scadenza globale dell'applicazione. Questo metodo è illustrato nell'Esempio 7.11.

### Esempio 7.11 - Determinazione delle scadenze di processo

La [Figura 7.13](#) mostra il PPG di un'applicazione real-time contenente 6 processi. Ogni cerchio è un nodo del grafo e rappresenta un processo. Il numero nel cerchio indica il tempo di servizio di un processo. Un arco nel PPG rappresenta un vincolo di precedenza. Così, un processo  $P_2$  può essere iniziato solo dopo che il processo  $P_1$  termina, il processo  $P_4$  può essere iniziato solo dopo che il processo  $P_2$  e il  $P_3$  terminano, ecc. Assumiamo che i processi non eseguano operazioni di I/O e che siano elaborati senza diritto di prelazione (nonpreemptive). Il totale dei tempi di servizio dei processi è 25 secondi. Se l'applicazione deve produrre una risposta in 25 secondi, le scadenze dei processi sarebbero le seguenti:



**Figura 7.13** Il grafo di precedenza dei processi (process precedence graph - PPG) per un sistema real-time.

| Processo | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
|----------|-------|-------|-------|-------|-------|-------|
| Deadline | 8     | 16    | 16    | 20    | 20    | 25    |

Un metodo pratico di stima delle scadenze dovrà prevedere diversi altri vincoli. Per esempio, i processi possono eseguire I/O. Se un'operazione di I/O di un processo può essere sovrapposta all'esecuzione di qualche processo indipendente, la scadenza dei suoi predecessori (e antenati) nel PPG può essere incrementata della durata della sovrapposizione di I/O. (I processi indipendenti sono definiti formalmente nel Paragrafo 6.1.) Per esempio, i processi  $P_2$  e  $P_3$  in [Figura 7.13](#) sono indipendenti l'uno dall'altro; se il tempo di servizio di  $P_2$  comprende 1 secondo di tempo di I/O, la scadenza di  $P_1$  può diventare 9 secondi invece di 8 secondi se l'operazione di I/O di  $P_2$  può sovrapporsi all'elaborazione di  $P_3$ . Comunque, l'esecuzione sovrapposta di processi deve considerare la disponibilità delle risorse. Di conseguenza la determinazione delle scadenze è più complessa di quanto riportato qui.

#### **Scheduling Earliest Deadline First (EDF)**

Come suggerisce il nome, questa politica seleziona sempre il processo con la scadenza più breve. Consideriamo un insieme di processi real-time che non eseguono operazioni di I/O. Se  $seq$  è la sequenza in cui i processi sono elaborati secondo una politica di scheduling su deadline e  $pos(P_i)$  è la posizione del processo  $P_i$  in  $seq$ , non si verifica un superamento della scadenza per il processo  $P_i$  solo se la somma dei suoi tempi di servizio e dei tempi di servizio di tutti i processi che lo precedono nella  $seq$  non supera la sua scadenza, cioè,

$$\sum_{k: pos(P_k) \leq pos(P_i)} x_k \leq D_i \quad (7.3)$$

dove  $x_k$  è il tempo di servizio del processo  $P_k$ , e  $D_i$  è la scadenza del processo  $P_i$ . Se questa condizione non è soddisfatta, si avrà un superamento della scadenza per il processo  $P_i$ .

Quando esiste una sequenza schedulabile, si dimostra che la Condizione 7.3 vale per tutti i processi; cioè non si verificherà per alcun processo un superamento della

scadenza. La [Tabella 7.4](#) illustra l'operazione della politica EDF per le scadenze dell'Esempio 7.11. La notazione  $P_4 : 20$  nella colonna processi nel sistema indica che il processo  $P_4$  ha la scadenza 20. I processi  $P_2$ ,  $P_3$  e  $P_5$ ,  $P_6$  hanno le stesse scadenze, quindi sono possibili tre schedulazioni oltre quella mostrata in [Tabella 7.4](#) con lo scheduling EDF. Nessuno di essi incorrerebbe nel superamento di scadenza.

| Tempo | Processo completato | Deadline overrun | Processi nel sistema  | Processo schedulato |
|-------|---------------------|------------------|---|---------------------|
| 0     | –                   | 0                | $P_1 : 8, P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$ | $P_1$               |
| 2     | $P_1$               | 0                | $P_2 : 16, P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$          | $P_2$               |
| 5     | $P_2$               | 0                | $P_3 : 16, P_4 : 20, P_5 : 20, P_6 : 25$                    | $P_3$               |
| 10    | $P_3$               | 0                | $P_4 : 20, P_5 : 20, P_6 : 25$                              | $P_4$               |
| 14    | $P_4$               | 0                | $P_5 : 20, P_6 : 25$  | $P_5$               |
| 20    | $P_5$               | 0                | $P_6 : 25$  | $P_6$               |
| 25    | $P_2$               | 0                | –   | –                   |

**Tabella 7.4** Funzionamento dello scheduling Earliest Deadline First (EDF).

I principali vantaggi dello scheduling EDF sono la sua semplicità e la natura non preemptive, che riduce il costo di scheduling. Lo scheduling EDF è una buona politica per lo scheduling statico poiché l'esistenza di una sequenza schedulabile, che può essere controllata *a priori*, assicura che non accadano superamenti di scadenze. È anche una buona politica di scheduling dinamico per l'utilizzo in sistemi real-time soft; comunque, il numero di processi che non rispettano la loro scadenza non è prevedibile. Il prossimo esempio illustra questo aspetto dello scheduling EDF.

#### Esempio 7.12 - I problemi dello scheduling EDF

Consideriamo il PPG della [Figura 7.13](#) con l'arco  $(P_5, P_6)$  rimosso. Esso contiene due applicazioni indipendenti, una che contiene i processi  $P_1 - P_4$  e  $P_6$ , mentre l'altra che contiene solo  $P_5$ . Se tutti i processi devono essere completati entro 19 secondi, non esiste una sequenza schedulabile. Ora le scadenze dei processi determinate usando l'Equazione 7.2 sono le seguenti:

| Processo | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ | $P_6$ |
|----------|-------|-------|-------|-------|-------|-------|
| Deadline | 2     | 10    | 10    | 14    | 19    | 19    |

Lo scheduling EDF può selezionare i processi o nella sequenza  $P_1, P_2, P_3, P_4, P_5, P_6$ , che è la stessa di [Tabella 7.4](#), o nella sequenza  $P_1, P_2, P_3, P_4, P_6, P_5$ . I processi  $P_5$  e  $P_6$  non rispettano le loro scadenze nella prima sequenza, mentre solo il processo  $P_5$  non rispetta la sua scadenza nella seconda sequenza. Non possiamo prevedere quale sequenza sarà scelta da un'implementazione dello scheduling EDF, quindi il numero di processi che non rispettano le loro scadenze non è prevedibile.

#### 7.5.3 Scheduling rate monotonic

Quando i processi di un'applicazione sono periodici, l'esistenza di una sequenza schedulabile può essere determinata in un modo interessante. Consideriamo tre processi indipendenti che non eseguono operazioni di I/O:

| Processo               | $P_1$ | $P_2$ | $P_3$ |
|------------------------|-------|-------|-------|
| Periodo (ms)           | 10    | 15    | 30    |
| Tempo di servizio (ms) | 3     | 5     | 9     |

Il processo  $P_1$  si ripete ogni 10 ms e necessita di 3 ms di tempo di CPU. Quindi la frazione del tempo di CPU che usa è  $3/10$ , cioè 0.30. Le frazioni del tempo di CPU usato da  $P_2$  e  $P_3$  sono rispettivamente  $5/15$  e  $9/30$ , cioè 0.33 e 0.30. Si sommano fino a 0.93, così se l'overhead di CPU del funzionamento del SO è trascurabile, è possibile elaborare

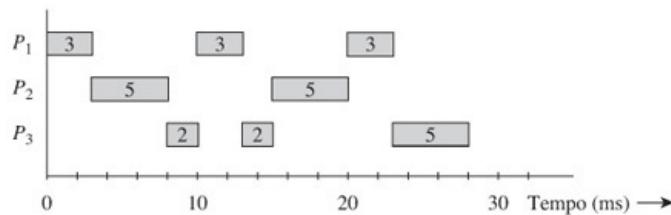
questi tre processi. In generale, un insieme di processi periodici  $P_1, \dots, P_n$  che non eseguono operazioni di I/O può essere elaborato da un sistema real-time hard che ha un trascurabile carico se:

$$\sum_{i=1 \dots n} \frac{x_i}{T_i} \leq 1 \quad (7.4)$$

dove  $T_i$  è il periodo di  $P_i$  e  $x_i$  è il suo tempo di servizio.

Dobbiamo ancora schedulare questi processi affinché possano tutti agire senza mancare le loro scadenze. La politica di scheduling *rate monotonic* (RM) esegue tale operazione come segue: determina il *rate* al quale un processo si deve ripetere, cioè il numero di ripetizioni per secondo, e considera il rate stesso come priorità del processo. Ora impiega una tecnica di scheduling basato su priorità per eseguire lo scheduling. Pertanto, un processo con un periodo inferiore ha una priorità più alta, cosa che dovrebbe consentirgli di terminare presto la sua operazione.

Nell'esempio precedente, le priorità dei processi  $P_1, P_2$  e  $P_3$  sarebbero 1/0.010, 1/0.015 e 1/0.025, cioè 100, 67 e 45 rispettivamente. La Figura 7.14 mostra come questi processi dovrebbero operare. Il processo  $P_1$  dovrebbe essere selezionato per primo. Si dovrebbe eseguire una volta e diventare dormiente dopo 3 ms, perché  $x_1 = 3$  ms. Ora dovrebbe essere selezionato  $P_2$  e dovrebbe terminare dopo 5 ms. Poi dovrebbe essere selezionato  $P_3$ , ma dovrebbe essere sospeso (preempted) dopo 2 ms perché  $P_1$  diventa *ready* per la seconda volta, e così via. Come mostrato in Figura 7.14, il processo  $P_3$  dovrebbe essere completato in 28 ms. Entro questo tempo,  $P_1$  è stato eseguito tre volte e  $P_2$  è stato eseguito due volte.



**Figura 7.14** Funzionamento di processi real-time utilizzando il rate monotonic scheduling.

Con lo scheduling rate monotonic non è garantita la ricerca di una sequenza schedulabile in tutte le situazioni. Per esempio, se il processo  $P_3$  avesse un periodo di 27 ms, la sua priorità sarebbe differente; tuttavia, le priorità relative dei processi dovrebbero essere immutate, quindi  $P_3$  sarebbe completato in 28 ms come prima, subendo quindi un superamento di scadenza pari a 1 ms. Una sequenza schedulabile si dovrebbe ottenere se  $P_3$  fosse stato selezionato a 20 ms e  $P_1$  a 25 ms; comunque, non è possibile con lo scheduling RM perché i processi sono selezionati in una modalità basata su priorità. Liu e Layland (1973) hanno mostrato che lo scheduling RM non può essere capace di evitare i superamenti di scadenze se la frazione totale del tempo di CPU usato dai processi secondo l'Equazione 7.4 supera  $m(2^{1/m} - 1)$ , dove  $m$  è il numero di processi. Questa espressione ha limite inferiore pari a 0.69, il che implica che se un'applicazione possiede un ampio numero di processi, lo scheduling RM non può raggiungere più del 69 per cento dell'utilizzo di CPU per rispettare le scadenze dei processi.

Liu e Layland riportano, inoltre, un *algoritmo di scheduling deadline-driven* che attribuisce dinamicamente le priorità ai processi basate sulle deadline correnti - ad un processo con una deadline più breve viene assegnata una priorità più alta di un processo con una deadline più lunga. Ciò può evitare il superamento della scadenza anche quando la frazione della Equazione 7.4 ha valore 1 e può raggiungere il 100 per cento dell'uso di CPU. Comunque, la sua prestazione pratica sarebbe inferiore a causa del carico dovuto all'assegnamento dinamico delle priorità. Ricordiamo che lo scheduling EDF può evitare il superamento della scadenza se esiste una sequenza schedulabile. Di conseguenza, anch'esso può raggiungere il 100 per cento dell'uso di CPU. Se impiegato staticamente, sarebbe sottoposto a un lieve carico durante l'operazione.

## 7.6 Casi di studio

### 7.6.1 Scheduling in Unix

Unix è un sistema operativo time-sharing puro che usa una politica di scheduling multilevello adattivo in cui le priorità di processo sono variate per assicurare buone prestazioni di sistema e per fornire inoltre un buon servizio utente. I processi sono allocati con priorità numeriche, dove un valore numerico più alto implica una priorità effettiva inferiore. In Unix 4.3 BSD, le priorità variano nell'intervallo da 0 a 127. I processi in modalità utente hanno priorità tra 50 e 127, mentre quelli in modalità kernel hanno priorità tra 0 e 49. Quando un processo è bloccato in una chiamata di sistema, la sua priorità è modificata assumendo un valore nell'intervallo 0-49, a seconda della causa del blocco. Quando diventa di nuovo attivo, esegue il resto della chiamata di sistema con tale priorità. Questa soluzione assicura che un processo venga selezionato il più presto possibile, completi il task che stava eseguendo in modalità kernel e rilasci le risorse del kernel. Quando esce dalla modalità kernel, la sua priorità ritorna al suo valore precedente, che era nell'intervallo 50-127.

Unix usa la seguente formula per variare la priorità di un processo:

$$\begin{aligned} \text{priorità processo} = & \text{ priorità base per processi utente} \\ & + f(\text{tempo CPU usato recentemente}) + \text{valore nice} \end{aligned} \quad (7.5)$$

che è implementata come segue: lo scheduler conserva il tempo di CPU usato da un processo nella sua tabella di processo. Questo campo è inizializzato a 0. Il clock genera un interrupt 60 volte al secondo, e il gestore del clock incrementa il contatore nel campo relativo all'uso della CPU del processo in esecuzione. Lo scheduler ricalcola le priorità del processo ogni secondo in un loop. Per ogni processo, divide il valore nel campo relativo all'uso della CPU per 2, lo memorizza, e lo utilizza anche come valore di  $f$ . Ricordiamo che un valore numerico elevato implica una priorità effettiva inferiore; quindi il secondo fattore nell'Equazione 7.5 riduce la priorità di un processo. La divisione per 2 assicura che l'effetto del tempo di CPU usato da un processo *decade*; cioè si riduce in un arco di tempo, per evitare il problema della starvation riscontrato nella politica least completed next (LCN) (Paragrafo 7.3.2).

Un processo può variare la sua priorità attraverso l'ultimo fattore nell'Equazione 7.5. La chiamata di sistema "*nice(<priority value>)*;" imposta il *valore nice* di un processo utente. Prende un valore zero o positivo come argomento. In tal modo, un processo può solo decrementare la sua effettiva priorità per essere nice agli altri processi. Tipicamente viene fatto quando entra in una fase CPU-bound.

#### Esempio 7.13 - Scheduling dei processi in Unix

La [Tabella 7.5](#) riassume il funzionamento della politica di scheduling Unix per i processi in [Tabella 7.2](#). Si assume che il processo  $P_3$  sia un processo I/O bound che inizia un'operazione di I/O che dura 0.5 secondi dopo l'uso della CPU per 0.1 secondi, e nessuno degli altri processi esegue I/O. Il campo  $T$  indica il tempo di CPU utilizzato da un processo e il campo  $P$  contiene la sua priorità. Lo scheduler aggiorna il campo  $T$  di un processo 60 volte al secondo e ricalcola le priorità di processo una volta ogni secondo. La time slice è 1 secondo, e la priorità di base dei processi utente è 60. La prima linea della [Tabella 7.5](#) mostra che al secondo 0, solo  $P_1$  è presente nel sistema. Il suo campo  $T$  contiene 0, di conseguenza la sua priorità è 60. Sono mostrate due linee per il secondo 1. La prima linea mostra i campi  $T$  dei processi per il secondo 1, mentre la seconda linea mostra i campi  $P$  e  $T$  dopo le azioni di calcolo della priorità al secondo 1. Alla remine della time slice, il contenuto del campo  $T$  di  $P_1$  è 60. L'operazione di dividere il tempo di CPU per 2 riduce il suo valore a 30, e quindi la priorità di  $P_1$  diventa 90. Al secondo 2, la priorità effettiva di  $P_1$  è inferiore rispetto a quella di  $P_2$  perché i loro campi  $T$  contengono 45 e 0, rispettivamente, e quindi è selezionato  $P_2$ . Analogamente è selezionato  $P_3$  al secondo 2.

| Tempo | $P_1$ |    | $P_2$ |    | $P_3$ |   | $P_4$ |    | $P_5$ |   | Processo schedulato |
|-------|-------|----|-------|----|-------|---|-------|----|-------|---|---------------------|
|       | P     | T  | P     | T  | P     | T | P     | T  | P     | T |                     |
| 0.0   | 60    | 0  |       |    |       |   |       |    |       |   | $P_1$               |
| 1.0   |       | 60 |       |    |       |   |       |    |       |   | $P_1$               |
|       |       | 90 | 30    |    |       |   |       |    |       |   |                     |
| 2.0   |       | 90 | 0     |    |       |   |       |    |       |   | $P_2$               |
|       | 105   | 45 | 60    | 0  |       |   |       |    |       |   |                     |
| 3.0   |       | 45 | 60    | 0  |       |   |       |    |       |   | $P_3$               |
|       | 82    | 22 | 90    | 30 | 60    | 0 |       |    |       |   |                     |
| 3.1   | 82    | 22 | 90    | 30 | 60    | 6 |       |    |       |   | $P_1$               |
| 4.0   |       | 76 | 30    | 6  |       |   |       |    |       |   | $P_3$               |
|       | 98    | 38 | 75    | 15 | 63    | 3 |       |    |       |   |                     |
| 4.1   | 98    | 38 | 75    | 15 | 63    | 9 |       |    |       |   | $P_2$               |
| 5.0   |       | 38 | 69    | 9  |       | 0 |       |    |       |   | $P_4$               |
|       | 79    | 19 | 94    | 34 | 64    | 4 | 60    | 0  |       |   |                     |
| 6.0   | 19    |    | 34    |    | 4     |   | 60    |    |       |   | $P_3$               |
|       | 69    | 9  | 77    | 17 | 62    | 2 | 90    | 30 |       |   |                     |

**Tabella 7.5** Funzionamento della politica di Scheduling Unix quando i processi eseguono operazioni di I/O.

Poiché  $P_3$  usa la CPU solo per 0.1 secondi prima di iniziare un'operazione di I/O, ha una priorità più alta di  $P_2$  quando viene eseguito lo scheduling al secondo 4; di conseguenza, esso è selezionato prima del processo  $P_2$ . È schedulato ancora al secondo 6. Questa caratteristica corregge in qualche modo l'inclinazione verso i processi I/O-bound mostrata dallo scheduling round-robin puro.

### Scheduling fair share

Per assicurare un'equa divisione del tempo di CPU tra i gruppi di processi, gli scheduler Unix aggiungono il termine  $f$  (tempo di CPU usato dai processi nel gruppo) all'Equazione 7.5. In tal modo, le priorità di tutti i processi di un gruppo si riducono quando qualcuno di essi utilizza tempo di CPU. Questa caratteristica assicura che i processi di un gruppo ricevano un trattamento privilegiato se nessuno di loro ha utilizzato per molto tempo la CPU recentemente. L'effetto del nuovo fattore decade anche durante il tempo.

### Esempio 7.14 - Scheduling fair share in Unix

La [Tabella 7.6](#) descrive lo scheduling fair share dei processi di [Tabella 7.2](#). I campi  $P$ ,  $C$  e  $G$  contengono la priorità di processo, il tempo di CPU utilizzato da un processo e il tempo di CPU utilizzato da un gruppo di processi, rispettivamente. Abbiamo due gruppi di processi. Il primo gruppo contiene i processi  $P_1$ ,  $P_2$ ,  $P_4$  e  $P_5$ , mentre il secondo gruppo contiene soltanto il processo  $P_3$ .

Tempo di completamento medio ( $\bar{t}_a$ ) = 9.2 secondi  
 Completamento pesato medio ( $\bar{w}$ ) = 3.15

| T* | P <sub>1</sub> |    |    | P <sub>2</sub> |    |    | P <sub>3</sub> |    |    | P <sub>4</sub> |    |    | P <sub>5</sub> |    |    | PS <sup>†</sup> |
|----|----------------|----|----|----------------|----|----|----------------|----|----|----------------|----|----|----------------|----|----|-----------------|
|    | P              | C  | G  | P              | C  | G  | P              | C  | G  | P              | C  | G  | P              | C  | G  |                 |
| 0  | 60             | 0  | 0  |                |    |    |                |    |    |                |    |    |                |    |    | P <sub>1</sub>  |
| 1  | 120            | 30 | 30 |                |    |    |                |    |    |                |    |    |                |    |    | P <sub>1</sub>  |
| 2  | 150            | 45 | 45 | 105            | 0  | 45 |                |    |    |                |    |    |                |    |    | P <sub>2</sub>  |
| 3  | 134            | 22 | 52 | 142            | 30 | 52 | 60             | 0  | 0  |                |    |    |                |    |    | P <sub>3</sub>  |
| 4  | 97             | 11 | 26 | 101            | 15 | 26 | 120            | 30 | 30 | 86             | 0  | 26 |                |    |    | P <sub>4</sub>  |
| 5  | 108            | 5  | 43 | 110            | 7  | 43 | 90             | 15 | 15 | 133            | 30 | 43 |                |    |    | P <sub>3</sub>  |
| 6  | 83             | 2  | 21 | 84             | 3  | 21 | 134            | 37 | 37 | 96             | 15 | 21 |                |    |    | P <sub>1</sub>  |
| 7  |                |    |    | 101            | 1  | 40 | 96             | 18 | 18 | 107            | 7  | 40 |                |    |    | P <sub>3</sub>  |
| 8  |                |    |    | 80             | 0  | 20 | 138            | 39 | 39 | 83             | 3  | 20 | 80             | 0  | 20 | P <sub>5</sub>  |
| 9  |                |    |    | 100            | 0  | 40 | 98             | 19 | 19 | 101            | 1  | 40 | 130            | 30 | 40 | P <sub>3</sub>  |
| 10 |                |    |    | 80             | 0  | 20 | 138            | 39 | 39 | 80             | 0  | 20 | 95             | 15 | 20 | P <sub>2</sub>  |
| 11 |                |    |    | 130            | 30 | 40 | 98             | 19 | 19 | 100            | 0  | 40 | 107            | 7  | 40 | P <sub>3</sub>  |
| 12 |                |    |    | 95             | 15 | 20 |                |    |    | 80             | 0  | 20 | 83             | 3  | 20 | P <sub>4</sub>  |
| 13 |                |    |    | 107            | 7  | 40 |                |    |    |                |    |    | 101            | 1  | 40 | P <sub>5</sub>  |
| 14 |                |    |    | 113            | 3  | 50 |                |    |    |                |    |    | 110            | 0  | 50 | P <sub>5</sub>  |
| 15 |                |    |    | 116            | 1  | 55 |                |    |    |                |    |    |                |    |    | P <sub>2</sub>  |
| 16 |                |    |    |                |    |    |                |    |    |                |    |    |                |    |    |                 |

\* T = Tempo

<sup>†</sup> PS = Processo schedulato

**Tabella 7.6** Esempio di scheduling fair share in Unix.

Al secondo 2, il processo  $P_2$  è appena arrivato. La sua priorità effettiva è bassa perché il processo  $P_1$ , che fa parte dello stesso gruppo, è stato eseguito per 2 secondi. In ogni caso  $P_3$  non possiede una bassa priorità quando arriva perché il tempo di CPU già utilizzato dal suo gruppo è pari a 0. Come previsto, il processo  $P_3$  riceve un trattamento privilegiato rispetto ad altri processi. Infatti, riceve una time slice a momenti alternati. I processi  $P_2$ ,  $P_4$  e  $P_5$  risentono dell'appartenenza allo stesso gruppo di processi. Questi fatti si ripercuotono nei tempi di completamento e nei completamenti pesati dei processi, che sono, come segue:

| Processo               | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|------------------------|-------|-------|-------|-------|-------|
| Tempo finale           | 7     | 16    | 12    | 13    | 15    |
| Tempo di completamento | 7     | 14    | 9     | 9     | 7     |
| Completamento pesato   | 2.33  | 4.67  | 1.80  | 4.50  | 2.33  |

### 7.6.2 Scheduling in Solaris

La piattaforma Solaris supporta quattro classi di processi - i processi time-sharing, i processi interattivi, i processi di sistema e i processi real-time. Una time slice è detta *quanto di tempo* nella terminologia di Solaris. I processi time-sharing e interattivi hanno priorità tra 0 e 59, dove un numero maggiore implica una priorità più alta. I processi di sistema hanno priorità tra 60 e 99; non viene assegnata a essi una time slice. I processi real-time hanno priorità tra 100 e 159 e sono selezionati tramite una politica round-robin all'interno di un livello di priorità. I thread usati per elaborare gli interrupt hanno priorità tra 160 e 169.

Lo scheduling dei processi time-sharing e interattivi è gestito da una tabella di dispatch. Per ogni livello di priorità, la tabella specifica come dovrebbe mutare la priorità di un processo in base alla sua natura, CPU-bound o I/O-bound, e per evitare, inoltre, la starvation. L'uso della tabella, piuttosto che di una regola di calcolo della priorità come per Unix, fornisce all'amministratore di sistema possibilità di tuning a grana fine. Le entrate della tabella di dispatch per ogni livello di priorità contengono i seguenti valori:

- $ts\_quantum$  il quanto di tempo per processi appartenenti a questo livello di priorità;
- $ts\_tqexp$  la nuova priorità di un processo che usa il suo quanto di

|                   |   |
|-------------------|---|
|                   | tempo interamente;  |
| <i>ts_slpret</i>  | la nuova priorità di un processo che si blocca prima di aver utilizzato il suo quanto di tempo interamente; |
| <i>ts_maxwait</i> | tempo massimo di attesa di un processo senza essere selezionato;  |
| <i>ts_lwait</i>   | la nuova priorità di un processo che non è stato selezionato entro il tempo <i>ts_maxwait</i> .             |

Un processo che si blocca prima che il suo quanto di tempo scada si suppone che sia un processo I/O-bound, la cui priorità è posta a *ts\_slpret*, che è una priorità più alta della sua priorità attuale. Analogamente, un processo che usa il suo quanto di tempo interamente si assume che sia un processo CPU-bound, pertanto *ts\_tqexp* è una priorità più bassa, *ts\_maxwait* è utilizzata per evitare starvation, di conseguenza *ts\_lwait* è una priorità più alta. Oltre questi cambiamenti di priorità effettuati dal kernel, un processo può cambiare la sua priorità attraverso la chiamata di sistema *nice* con un numero nell'intervallo da -19 a 19 come parametro.

Solaris 9 supporta, inoltre, una classe di scheduling fair share. Un gruppo di processi prende il nome di progetto e a esso viene assegnata una piccola quantità di tempo di CPU. La fair share di un progetto in qualsiasi istante dipende dalle condivisioni degli altri progetti che sono attivi in maniera concorrente; è il quoziente delle condivisioni del progetto e della somma delle condivisioni di tutti quei progetti che hanno almeno un processo attivo. Nei sistemi multiprocessore, le condivisioni sono definite indipendentemente per ciascuna CPU. Solaris 10 introduce il concetto di *zone* al di sopra dei progetti. Le condivisioni di CPU vengono assegnate sia alle zone sia ai progetti allo scopo di fornire uno scheduling a due-livelli.

### 7.6.3 Scheduling in Linux

Linux supporta sia applicazioni real-time che non-real-time. A tale scopo, possiede due classi di processi. I processi real-time possiedono priorità statiche tra 0 e 100, dove 0 è la priorità più alta. I processi real-time possono essere schedulati in due modi: FIFO o round-robin nell'ambito di ogni livello di priorità. Il kernel associa una flag a ogni processo per indicare come dovrebbe essere schedulato.

I processi non-real-time hanno priorità più basse di quelli real-time; le loro priorità sono dinamiche e hanno valori numerici tra -20 e 19, dove -20 rappresenta la priorità più alta. Effettivamente, il kernel ha (100 + 40) livelli di priorità. Per cominciare, ogni processo non-real-time ha priorità 0. La priorità può essere variata dal processo stesso attraverso le chiamate di sistema *nice* o *setpriority*. Tuttavia, sono necessari privilegi speciali per aumentare la priorità con la *nice*, quindi i processi usano tipicamente questa chiamata per diminuire le loro priorità quando desiderano essere "simpatici" con altri processi. In aggiunta a tale modalità di variazione della priorità, il kernel varia la priorità di un processo per rispecchiare la sua natura I/O-bound o CPU-bound. Per implementarla, il kernel conserva le informazioni relativamente alla quantità di tempo di CPU che il processo ha usato recentemente, per quanto tempo è rimasto in stato di *blocked*, e aggiunge un bonus tra 5 e -5 al valore *nice* del processo. Così, un processo altamente interattivo avrebbe una priorità effettiva data da *nice* -5, mentre un processo CPU-bound avrebbe una priorità effettiva data da *nice* +5.

A causa della struttura delle priorità multilivello, il kernel Linux organizza i suoi dati di scheduling come mostrato in [Figura 7.12](#) del Paragrafo 7.4.3. Per limitare il carico di scheduling, Linux usa uno scheduler schematico analogo a quello di [Figura 5.9](#). Così, lo scheduling non viene eseguito dopo ogni gestione di un evento. Viene eseguito quando il processo attualmente in esecuzione deve bloccarsi a causa di una chiamata di sistema, o quando la flag *need\_resched* è stata settata da un'azione di gestione di un evento. Questo accade durante la gestione della scadenza del time slice, o durante la gestione di un evento che attiva un processo la cui priorità è più alta di quella del processo correntemente in esecuzione.

I processi non-real-time vengono schedulati usando la nozione di time slice, che nel sistema operativo Linux è un quanto di tempo che un processo può usare durante un certo periodo di tempo secondo la sua priorità (Paragrafo 7.4.8). Un processo che esaurisce la sua time slice riceverà una nuova time slice solo dopo che tutti i processi abbiano esaurito le loro time slice. Linux utilizza le time slice nell'intervallo compreso tra 10 e 200 ms. Per assicurare che un processo a più alta priorità riceverà maggiore CPU di

un processo a priorità inferiore, Linux assegna una time slice maggiore al processo a priorità più alta. Questa assegnazione di time slice non influenza i tempi di risposta perché un processo ad alta priorità sarebbe di natura interattiva, di conseguenza eseguirebbe un'operazione di I/O prima di utilizzare abbastanza tempo di CPU.

Lo scheduler Linux usa due liste di processi, un'*active list* e una *exhausted list*. Entrambe le liste sono ordinate per priorità dei processi e utilizzano le strutture dati descritte precedentemente. Lo scheduler seleziona un processo dall'active list, che utilizza tempo della sua time slice. Quando la sua time slice è esaurita, viene messo nella exhausted list. Gli scheduler nel kernel Linux 2.5 e nei kernel precedenti eseguivano un ciclo di ricalcolo della priorità quando l'active list diventava vuota. Il ciclo calcolava una nuova time slice per ciascun processo in base alla loro priorità dinamica. Alla fine del ciclo, tutti i processi venivano trasferiti all'active list e veniva ripresa la normale attività di scheduling.

Il kernel Linux 2.6 usa un nuovo scheduler che ha minor overhead e scalabile rispetto al numero di processi e di CPU. Lo scheduler suddivide il carico del ricalcolo delle priorità in tutta l'operazione dello scheduler, piuttosto che concentrarla nel ciclo di ricalcolo. Questo avviene ricalcolando la priorità di un processo quando il processo esaurisce la sua time slice e viene spostato alla exhausted list. Quando l'active list diventa vuota, lo scheduler scambia semplicemente l'active list con la exhausted list.

La scalabilità dello scheduler è assicurata in due modi. Lo scheduler ha un bit flag per indicare se la lista dei processi per un livello di priorità è vuota. Quando è invocato, lo scheduler valuta i flag delle liste di processo in ordine decrescente di priorità, e seleziona il primo processo nella prima lista di processo non vuota che trova. Questa procedura ha un carico (overhead) di scheduling che non dipende dal numero di processi ready; ma dipende solo dal numero di livelli di scheduling, di conseguenza è limitato da una costante. Questo scheduling è detto scheduling  $O(1)$ , cioè di ordine 1. Gli scheduler nei precedenti kernel Linux usavano un lock di sincronizzazione sull'active list dei processi per evitare race condition quando erano supportate più CPU. Il kernel Linux 2.6 gestisce le active list legate a ogni CPU; questo elimina il lock di sincronizzazione e i ritardi a esso associati. Questa soluzione assicura inoltre che un processo lavora sulla stessa CPU ogni volta che è schedulato; aiuta ad assicurare migliori hit ratio di cache.

## 7.6.4 Scheduling in Windows

Lo scheduling di Windows ha lo scopo di fornire buoni tempi di risposta ai thread real-time e interattivi. Lo scheduling è priority-driven e con diritto di prelazione (preemptive). Lo scheduling nell'ambito di un livello di priorità viene eseguito tramite una politica round-robin con time-slicing. Una time slice è detta *quanto* nella terminologia Windows. Le priorità di thread non-real-time sono variate dinamicamente per favorire i thread interattivi. Questo aspetto è analogo allo scheduling multilivello adattivo (Paragrafo 7.4.5).

Ai thread real-time sono attribuite priorità più alte rispetto ad altri thread, i.e. hanno priorità nell'intervallo tra 16-31, mentre gli altri thread hanno priorità nell'intervallo tra 1-15. Le priorità dei thread non real-time possono variare durante la loro vita, di conseguenza questa classe di thread è anche detta *classe a priorità variabile*. La priorità effettiva di un thread in questa classe in ogni momento è una combinazione di tre fattori - la priorità base del processo al quale il thread appartiene; la priorità base del thread, che si trova nell'intervallo da -2 a 2, e una componente dinamica assegnata dal kernel per favorire i thread interattivi.

Il kernel varia la componente dinamica della priorità del thread come segue: se il thread esaurisce la sua time slice quando è schedulato, la sua priorità è ridotta di 1. Quando un thread *waiting*, cioè, *blocked*, viene attivato, gli viene attribuito un incremento di priorità basato sulla natura dell'evento su cui era bloccato. Se era stato bloccato su un input da tastiera, la sua priorità è incrementata di 6. Per negare un ingiusto vantaggio a un thread I/O-bound, il tempo rimanente del suo quanto corrente è ridotto di un clock tick ogni volta che effettua una richiesta di I/O. Per evitare la starvation, la priorità di un thread *ready* che non ha ricevuto tempo di CPU per più di 4 secondi è aumentata a 15 e il suo quanto è incrementato a due volte il suo valore normale. Quando questo quanto termina, la sua priorità e il quanto tornano ai loro vecchi valori.

Lo scheduler utilizza una struttura dati che assomiglia a quella mostrata in [Figura 7.12](#), tranne per due dettagli che forniscono efficienza. Poiché i valori di priorità si

trovano nell'intervallo 0-31 (con priorità 0 riservata a un thread di sistema) viene usato un array di 32 puntatori per puntare alle code dei thread ready ai diversi livelli di priorità. Per indicare se un thread ready esiste a tutti i livelli di priorità viene utilizzato un vettore a 32 bit flag. Questa soluzione consente allo scheduler di localizzare velocemente il primo thread nella coda non vuota a più alta priorità. Quando nessun thread del sistema o thread utente è in stato *ready*, lo scheduler seleziona uno speciale *thread idle* per la CPU che continuamente esegue un *idle loop* finché viene selezionato un thread. Nel loop, attiva funzioni nel livello di astrazione dell'- hardware (HAL) a istanti appropriati per eseguire la gestione dell'energia. In un sistema multiprocessore, lo scheduler che opera su di una CPU può schedulare un thread su un'altra CPU che è *idle* (Paragrafo 10.6.3). Per facilitare questo scheduling, l'*idle loop* esamina anche le strutture dati di scheduling per controllare se un thread è stato schedulato sulla CPU che sta eseguendo l'*idle loop* e, se questo è il caso, commuta la CPU al thread schedulato.

Per risparmiare energia quando l'elaboratore è *idle*, Windows prevede un numero di stati di sistema in cui l'elaboratore opera in modo tale che utilizza poca energia. Nello stato *ibernato*, gli stati delle applicazioni in esecuzione sono memorizzati su disco e il sistema è spento. Quando il sistema viene attivato, gli stati delle applicazioni vengono ripristinati da disco prima che l'operazione possa riprendere. L'uso del disco per memorizzare gli stati dell'applicazione porta a un lento ripristino; tuttavia, fornisce affidabilità perché il funzionamento dell'elaboratore è immune da perdita o esaurimento di energia mentre l'elaboratore è *ibernato*. Nello stato *standby*, gli stati delle applicazioni in esecuzione sono salvate in memoria, e l'elaboratore passa in una modalità operativa a basso consumo. Il ripristino che fa uso degli stati delle applicazioni memorizzati in memoria è più veloce. Tuttavia, le informazioni sullo stato verrebbero perse se l'energia venisse a mancare o fosse esaurita mentre il sistema è in stato *standby*, quindi il funzionamento dell'elaboratore non è affidabile. Di conseguenza, Windows Vista ha introdotto un nuovo stato ibrido detto stato di *sleep* in cui gli stati dell'applicazione sono memorizzati sia in memoria che su disco. Il funzionamento del sistema è ripristinato come nello stato *standby* se gli stati dell'applicazione sono disponibili in memoria; altrimenti, è ripristinato come nello stato *ibernato* usando gli stati dell'applicazione memorizzati su disco.

## 7.7 Analisi delle prestazioni delle politiche di scheduling

L'analisi delle prestazioni di una politica di scheduling ha come obiettivo lo studio delle sue prestazioni, usando misure quali il tempo di risposta di un processo, l'efficienza d'uso della CPU e il throughput del sistema. L'analisi delle prestazioni può essere utilizzata per confrontare le prestazioni di politiche di scheduling alternative, e per determinare "buoni" valori di parametri chiave di sistema come la time slice, il numero di utenti attivi e la dimensione della lista di processi *ready*.

La prestazione di una politica di scheduling è sensibile alla natura delle richieste dirette a essa, e quindi l'analisi delle prestazioni dovrebbe essere condotta nell'ambiente in cui la politica è stata introdotta. L'insieme delle richieste dirette alla politica di scheduling è detto *workload*. Il primo passo nell'analisi delle prestazioni di una politica è caratterizzare accuratamente il suo tipico *workload*. Nel seguito discuteremo alcuni problemi relativi a tale passo.

Come detto nel Paragrafo 7.2, nel contesto della politica SJF, le stime utente dei tempi di servizio non sono affidabili perché gli utenti non hanno l'esperienza per fornire buone stime del tempo di servizio oppure perché utenti esperti possono fornire stime ingannevoli per ottenere un trattamento di favore dal sistema. Alcuni utenti possono anche ricorrere a modifiche nelle loro richieste per ottenere un servizio migliore; per esempio, un utente che sa che si sta usando la politica SJF, può dividere un programma che richiede un elevato tempo di esecuzione in diversi programmi con tempi brevi di servizio. Tutti questi fattori alterano il *workload*. Di conseguenza, la caratterizzazione di un tipico *workload* dovrebbe essere sviluppata senza coinvolgere gli utenti.

Possono essere usati tre approcci per l'analisi delle prestazioni di politiche di scheduling:

- implementazione di una politica di scheduling in un SO;
- simulazione;
- modellazione matematica.

Sia la simulazione che la modellazione matematica evitano la necessità di implementare

la politica di scheduling in un SO, evitando così il costo, la complessità e i ritardi legati all'implementazione della politica. Tuttavia, per ottenere lo stesso risultato di un'implementazione, questi approcci richiedono una caratterizzazione molto dettagliata delle richieste del workload, che generalmente non è fattibile in pratica. Di conseguenza, gli aspetti delle prestazioni come il carico di scheduling o il servizio per richieste individuali sono studiati meglio attraverso l'implementazione, mentre la simulazione e la modellazione matematica ben si adattano allo studio delle prestazioni di una politica di scheduling e per la determinazione di "buoni" valori di parametri di sistema come la time slice, la numero di utenti o la dimensione della lista dei processi *ready*.

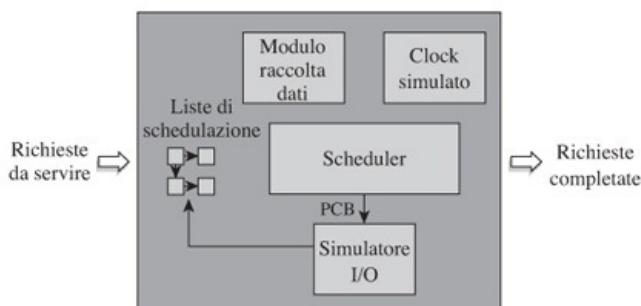
### 7.7.1 Analisi della prestazione attraverso implementazione

La politica di scheduling da valutare è implementata in un sistema operativo reale che è usato nell'ambiente operativo di riferimento. Il SO riceve richieste utente reali; le elabora utilizzando la politica di scheduling e colleziona dati per analisi statistiche sulla prestazione della politica. Questo approccio all'analisi delle prestazioni è invasivo, perché un SO reale deve essere modificato e ricostruito per ogni politica di scheduling da analizzare. Tutto ciò può essere evitato usando una macchina virtuale, che consente a un kernel ospite di essere modificato senza influenzare il funzionamento del kernel ospitante; tuttavia, l'overhead introdotto dall'uso della macchina virtuale causerebbe imprecisioni nella misurazione delle prestazioni.

### 7.7.2 Simulazione

La simulazione è ottenuta codificando la politica di scheduling e le funzioni importanti di SO come un programma, detto *simulator*, e usando un tipico workload come input. Il workload è una registrazione di alcuni workload reali diretti al SO durante uno periodo campione. L'analisi può essere ripetuta con molti workload per eliminare l'effetto delle variazioni attraverso i workload.

La Figura 7.15 mostra lo schema di un simulatore. Il simulatore lavora nel seguente modo. Mantiene le strutture dati che sono utilizzate dalla politica di scheduling simulata, in cui pone le informazioni riguardanti le richieste utente come arrivano nel sistema, vengono ammesse e vengono elaborate. Mantiene, inoltre, un clock per tenere traccia del *tempo simulato*. Istante per istante, simula l'azione di scheduling e seleziona una richiesta per l'elaborazione. Fa una stima di quanto tempo la richiesta userà la CPU prima che si verifichi un evento, come l'inizio di un'operazione di I/O o il completamento di una richiesta. Poi, incrementa il clock di simulazione di quanto tempo la richiesta avrebbe utilizzato la CPU prima del verificarsi dell'evento ed elimina la richiesta dalla coda di scheduling. Poi esegue una volta ancora lo scheduling e così via. Può comprendere altri moduli, come un modulo simulatore di I/O che dovrebbe prevedere quando l'operazione di I/O, iniziata da una richiesta, dovrebbe terminare. Quando il clock di simulazione mostra questo tempo, aggiunge la richiesta alla coda di scheduling. Il modulo di raccolta dati colleziona dati utili per l'analisi di prestazioni. Il livello di dettaglio gestito in un simulatore determina il costo della simulazione e la qualità dei risultati.



**Figura 7.15** Simulazione di una politica di scheduling.

### 7.7.3 Modellazione matematica

Una *modellazione matematica* consiste di due componenti: un modello per il server e un modello per il workload da elaborare. Il modello fornisce un insieme di espressioni matematiche per importanti caratteristiche delle prestazioni quali il tempo di servizio delle richieste e l'overhead. Queste espressioni consentono di determinare l'influenza dei vari parameteri sulle prestazioni di sistema. Il modello del workload differisce dai workload utilizzati nelle simulazioni in quanto non è una registrazione di workload reali in uno specifico periodo di tempo. È, invece, una *distribuzione* statistica che rappresenta il workload; cioè, è una funzione che genera richieste fittizie che hanno le stesse proprietà statistiche del reale workload entro *ogni* periodo di tempo osservato.

### **Teoria delle code**

L'uso diffuso di modelli matematici per l'analisi delle prestazioni di vari sistemi ha determinato lo sviluppo di una branca separata della matematica conosciuta come *teoria delle code*. L'analisi delle prestazioni che fa uso della teoria delle code è detta analisi delle code. La prima applicazione molto nota di analisi delle code è dovuta a Erlang (1909) nella valutazione delle prestazioni di una centralina telefonica con il numero delle linee troncate come parametro di controllo.

Il modello fondamentale della teoria delle code di un sistema è identico al semplice modello di scheduler discusso all'inizio di questo capitolo (Figura 7.1), ed è noto come *modello single-server*. L'analisi delle code è utilizzata per lo sviluppo di espressioni matematiche per l'efficienza del server, la lunghezza media della coda e il tempo di attesa medio. Una richiesta che arriva all'istante  $a_i$  con tempo di servizio  $x_i$  è completata all'istante  $c_i$ . Il tempo trascorso (detto elapsed time) ( $c_i - a_i$ ) dipende da due fattori: i tempi di arrivo e i tempi di servizio delle richieste che sono o in esecuzione oppure presenti nella coda di scheduling in qualche istante compreso nell'intervallo ( $c_i - a_i$ ) e dalla politica di scheduling utilizzata dal server. E ragionevole assumere che i tempi di arrivo e di elaborazione delle richieste che arrivano al sistema non siano già noti; cioè, queste caratteristiche non sono deterministiche in natura.

Sebbene le caratteristiche di ciascuna richiesta non siano note, abitualmente si ritengono conformi ad alcune distribuzioni statistiche. Un ambiente computazionale è così caratterizzato da due parametri: una distribuzione statistica per i tempi di arrivo delle richieste e una distribuzione statistica per i tempi di servizio. Diamo una breve introduzione alle distribuzioni statistiche e al loro uso nella modellazione matematica, usando la seguente notazione:

$\alpha$  tasso medio di arrivo (richieste per secondo);  
 $\omega$  tasso medio di esecuzione (richieste per secondo);  
 $\rho = \alpha/\omega$ .

$\rho$  è detto *fattore di utilizzo* del server. Quando  $\rho > 1$ , il carico diretto al sistema supera la sua capacità. In questo caso, il numero di richieste nel sistema si incrementa indefinitamente. La valutazione delle prestazioni di un sistema tale è di scarsa rilevanza pratica poiché i tempi di completamento possono essere arbitrariamente alti. Quando  $\rho < 1$ , la capacità del sistema supera il carico totale diretto a esso. Tuttavia, questo è vero solo come media a lungo termine; non potrebbe valere in un intervallo arbitrario di tempo. Di conseguenza, il server può essere idle di tanto in tanto, e possono esserci poche richieste nella coda in alcuni istanti.

In pratica, la maggior parte dei sistemi soddisfa  $\rho < 1$ . Anche quando si considera uno slow server,  $\rho$  non supera 1 perché la maggior parte dei sistemi sono auto-regolatori in natura: il numero di utenti è finito e il tasso di arrivo delle richieste diminuisce quando la lunghezza della coda è grande poiché la maggior parte delle richieste utente sono bloccate nella coda!

Un sistema raggiunge un *stato di equilibrio* (steady state) quando tutti i transienti nel sistema, indotti a causa del suo brusco avvio al tempo  $t = 0$ , muoiono. Nello steady state, i valori della lunghezza media della coda, tempo di attesa medio, tempi di completamento medio, ecc., rispecchiano le prestazioni della politica di scheduling. Per ottenere questi valori, cominciamo ad assumere determinate distribuzioni per l'arrivo e l'elaborazione delle richieste nel sistema.

### **Tempi di arrivo**

Il tempo intercorso tra l'arrivo di due richieste consecutive è detto *tempo tra gli arrivi*. Poiché  $\alpha$  è il tasso di arrivo, il tempo medio dei tempi di arrivo è  $1/\alpha$ . Una distribuzione

statistica che abbia questo tempo medio tra gli arrivi (mean inter-arrival time) e che si adatti ragionevolmente bene ai dati empirici può essere utilizzata per la caratterizzazione del workload. Gli arrivi delle richieste nel sistema possono essere considerati come eventi casuali totalmente indipendenti l'uno dall'altro. Sono state fatte due assunzioni che conducono alla distribuzione di Poisson degli arrivi. Primo, si suppone che il numero degli arrivi nell'intervallo da  $t$  a  $t + dt$  dipenda solo dal valore di  $dt$  e non dalla storia precedente del sistema nell'intervallo  $(0, t)$ . Secondo, per valori piccoli di  $dt$ , la probabilità di più di un arrivo nell'intervallo da  $t$  a  $(t + dt)$  si assume che sia trascurabile. La prima assunzione è conosciuta come la *proprietà di assenza di memoria* della distribuzione dei tempi di arrivo. Una funzione di distribuzione esponenziale che fornisce la probabilità di un arrivo nell'intervallo da 0 a  $t$  per ogni  $t$  ha la forma:

$$F(t) = 1 - e^{-\alpha \cdot t}$$

Questa distribuzione ha tempo medio tra i tempi di arrivo  $1/\alpha$  poiché  $\int_0^\infty t \cdot dF(t) = 1/\alpha$ . Si verifica che la distribuzione esponenziale adatta i tempi tra gli arrivi ai dati empirici in modo quasi corretto (nella pratica si verifica che una distribuzione iper-esponenziale con la stessa media  $1/\alpha$  sia un'approssimazione migliore per i dati sperimentali (Coffman e Wood, 1966).

### Tempi di servizio

La funzione  $S(t)$  fornisce la probabilità che il tempo di servizio di una richiesta sia minore o uguale a  $t$ .

$$S(t) = 1 - e^{-\omega \cdot t}$$

Come nel caso dei tempi di arrivo, facciamo due assunzioni che ci conducono alla distribuzione di Poisson dei tempi di servizio. Di conseguenza, la probabilità che una richiesta che abbia già utilizzato  $t$  unità di tempo di servizio termini nei  $dt$  secondi successivi dipende solo dal valore di  $dt$  e non da  $t$ . Nello scheduling preemptive, una richiesta viene ogni volta schedulata per l'esecuzione dopo un'interruzione. La proprietà di assenza di memoria dei tempi di servizio implica che un algoritmo di scheduling non può fare alcuna previsione basata sulla storia passata di una richiesta nel sistema. In tal modo, qualsiasi politica di scheduling preemptive che richiede la conoscenza del comportamento futuro delle richieste deve dipendere dalle stime dei tempi di servizio fornite dal programmatore. Le prestazioni dello scheduling dipenderanno allora in maniera critica dagli input dell'utente e possono essere manipolate dagli utenti. In pratica, un sistema deve forzare per raggiungere l'effetto opposto, cioè le prestazioni del sistema dovrebbero essere immuni dalle specifiche dell'utente (o misspecification) del tempo di servizio di una richiesta. Questo requisito punta verso lo scheduling round-robin con time-slicing come politica pratica di scheduling.

### Analisi delle prestazioni

La relazione tra  $L$ , la lunghezza della coda media e  $W$ , il tempo di attesa medio per una richiesta prima che il suo servizio inizi, è data dalla formula di Little

$$L = \alpha \times W \quad (7.6)$$

Questa relazione deriva dal fatto che mentre una richiesta attende in coda,  $\alpha \times W$  nuove richieste arrivano in coda.

Quando arriva una nuova richiesta, viene aggiunta alla coda delle richieste. In uno scheduling nonpreemptive, la nuova richiesta viene considerata solo dopo che il server abbia completato la richiesta attualmente servita. Sia  $W_0$  il tempo atteso per completare la richiesta corrente. Naturalmente,  $W_0$  è indipendente dalla politica di

$$W_0 = \frac{\alpha}{2} \cdot \int_0^\infty t^2 dF(t) \text{ scheduling. } t^2 dF(t), \text{ e ha valore } \frac{\alpha}{\omega^2} \text{ per la distribuzione}$$

esponenziale  $F(t) = 1 - e^{-\alpha \cdot t}$ .  $W$ , il tempo di attesa medio per una richiesta, quando si usa una specifica politica di scheduling, è calcolato a partire da  $W_0$  e dalle caratteristiche della politica di scheduling. Descriviamo nel seguito come vengono ricavati i tempi di attesa medi per le politiche FCFS e SJF. Tali informazioni sono più complesse da ricavare per le politiche HRN e round-robin e possono essere trovate in Brinch Hansen (1973). La

**Tabella 7.7** sintetizza i tempi di attesa medi di una richiesta il cui tempo di servizio sia  $t$  con l'utilizzo di differenti politiche di scheduling.

| Politica di scheduling | Tempo di attesa medio per una richiesta con tempo di servizio = $t$   |
|------------------------|---|
| FCFS                   | $\frac{W_0}{1 - \rho}$  |
| SJF                    | $\frac{W_0}{1 - \rho_t}$ , dove $\rho_t = \int_0^t \alpha \cdot y \cdot dS(y)$  |
| HRN                    | Per piccoli $t$ : $W_0 + \frac{\rho^2}{1 - \rho} \times \frac{t}{2}$<br>Per grandi $t$ : $\frac{W_0}{(1 - \rho) \left(1 - \rho + \frac{2 \cdot W_0}{t}\right)}$                                 |
| Round-robin            | $\frac{n}{\omega(1 - P_0)} - \frac{1}{\alpha}$ , dove $P_0 = \frac{1}{\sum_{j=0}^n \frac{n!}{(n-j)!} \times (\alpha)^j}$<br>( $P_0$ è la probabilità che nessun terminale aspetti una risposta) |

Nota:  $W_0 = \frac{\alpha}{2} \cdot \int_0^\infty t^2 dF(t)$ . Per una distribuzione esponenziale  $F(t) = 1 - e^{-\alpha \cdot t}$ , è  $\frac{\alpha}{\omega^2}$ .

**Tabella 7.7** Riepilogo dell'analisi delle prestazioni.

$W$ , il tempo di attesa per una richiesta  $r'$ , è la quantità di tempo che  $r'$  passa nella coda prima che venga servito. Di conseguenza, nello scheduling FCFS

$$W = W_0 + \sum_i x_i$$

dove la richiesta  $i$  è prima della richiesta  $r'$  nella coda di scheduling. Poiché il sistema è in steady state, possiamo sostituire il termine  $\sum_i$  con  $n \times \frac{1}{\omega}$ , dove  $n$  è il numero di richieste prima di  $r'$  e  $\frac{1}{\omega}$  il tempo di servizio medio. Poiché  $n$  è la lunghezza della coda media,  $n = \alpha W$  dalla formula di Little. Quindi:

$$\begin{aligned} W &= W_0 + \alpha \times W \times \frac{1}{\omega} \\ &= W_0 + \rho \times W \end{aligned}$$

Perciò,  $W = \frac{W_0}{1 - \rho}$ . Così, il tempo di attesa medio nello scheduling FCFS cresce bruscamente per valori grandi di  $\rho$ .

Nello scheduling SJF, le richieste i cui tempi di servizio sono minori di  $x_{r'}$ , dove  $x_{r'}$  è il tempo di servizio di  $r'$ , sono servite prima della richiesta  $r'$ . Di conseguenza il tempo di attesa per la richiesta  $r'$  è:

$$\begin{aligned} W &= W_0 + \sum_i x_i, \text{ dove } x_i < x_{r'} \\ &= \frac{W_0}{1 - \rho'_{r'}}, \text{ dove } \rho'_{r'} = \int_0^{r'} \alpha \cdot y \cdot dS(y) \end{aligned}$$

### Pianificazione della capacità

L'analisi delle prestazioni può essere utilizzata per la pianificazione della capacità. Per esempio, le formule mostrate in **Tabella 7.7** possono essere usate per determinare i valori di importanti parametri come la misura dell'ultimo dei processi *ready* usato dal kernel.

Come esempio, consideriamo un SO in cui il tasso medio di arrivo di richieste è 5 richieste per secondo, e il tempo medio di risposta per richieste è 3 secondi. La

lunghezza media della coda è calcolata dalla formula di Little (Equazione 7.6) come  $5 \times 3 = 15$ . Notare che le code supereranno questa lunghezza di volta in volta. L'esempio seguente fornisce un criterio di decisione riguardo alla capacità della coda dei processi ready.

### Esempio 7.15 - Pianificazione della capacità con l'uso dell'analisi delle code

Un kernel consente che fino a  $n$  ingressi nella coda siano richieste *ready*. Se la coda è piena quando una nuova richiesta arriva, la richiesta viene rifiutata e lascia il SO. Sia  $\rho_i$  la probabilità che la coda ready contenga  $i$  processi a ogni istante, si può mostrare che risulta:

$$p_i = \frac{\rho^i \times (1 - \rho)}{1 - \rho^{n+1}} \quad (7.7)$$

Per  $\rho = 0.5$  e  $n = 3$ ,  $p_0 = \frac{8}{15}$ ,  $p_1 = \frac{4}{15}$ ,  $p_2 = \frac{2}{15}$ , e  $p_3 = \frac{1}{15}$ . Quindi il 6.7 percento di richieste viene perso. Si dovrebbe usare un valore maggiore di  $n$  per ridurre il numero di richieste perse.

## Riepilogo

Lo scheduler di un SO decide quale deve essere il prossimo processo a essere elaborato dalla CPU e per quanto tempo deve essere elaborato. Le sue decisioni influenzano sia il servizio utente che le prestazioni del sistema. In questo capitolo, abbiamo discusso tre tecniche degli scheduler di processo: *scheduling basato su priorità*, *riordino delle richieste* e *variazione della time slice*; abbiamo studiato come gli scheduler le usano per fornire un'auspicata combinazione tra servizio utente e prestazioni di sistema. Inoltre, abbiamo studiato lo scheduling real-time.

Una politica di scheduling nonpreemptive esegue lo scheduling solo quando il processo che la CPU sta elaborando termina; la politica focalizza solamente sul riordino delle richieste per migliorare il tempo di completamento medio dei processi. La politica *shortest job first* (SJF) ha il problema della starvation, poiché alcuni processi possono essere ritardati indefinitamente. La politica *highest response ratio next* (HRN) non ha questo problema perché il rapporto di risposta di un processo tende ad aumentare all'aumentare dell'attesa della CPU.

Le politiche di scheduling preemptive prelazionano un processo quando è considerato opportuno prendere una nuova decisione di scheduling. La politica *round-robin* (RR) elabora tutti i processi a turno, limitando la quantità del tempo di CPU usato da ciascun processo al valore della time slice. La politica *least completed next* (LCN) seleziona il processo che ha ricevuto la minima quantità di servizio, mentre la politica *shortest time to go* (STG) seleziona il processo che è più vicino al completamento. Quest'ultima politica è anche nota come Shortest Remaining Time First (SRTF).

In pratica, un sistema operativo usa un criterio che coinvolge i tre scheduler. Lo *scheduler a lungo termine* decide quando un processo deve essere ammesso all'elaborazione. Lo *scheduler a medio termine* decide quando un processo deve essere scaricato (swapped out) su disco e quando ricaricato in memoria. Lo *scheduler a breve termine* seleziona uno dei processi che è presente in memoria. La politica di scheduling *multilivello adattivo* assegna differenti valori di time slice ai processi con diverse priorità e varia la priorità di un processo in base al suo comportamento recente per fornire una combinazione di buon tempo di risposta e basso overhead di scheduling. La politica di scheduling *fair share* assicura che i processi di un'applicazione collettivamente non superino uno specifico share di tempo di CPU.

Lo scheduling real-time concentra l'attenzione sul rispetto dei vincoli di tempo delle applicazioni. Lo *scheduling deadline* considera le deadline dei processi mentre effettua decisioni di scheduling. Lo *scheduling rate monotonic* assegna le priorità ai processi in base ai loro periodi ed effettua uno scheduling basato su priorità.

I moderni sistemi operativi hanno a che fare con diversi workload, quindi gli scheduler dividono i processi in differenti classi come quella real-time e non-real-time, e usano un'appropriata politica di scheduling per ogni classe.

L'analisi delle prestazioni viene usata sia per studiare che per correggere la prestazione delle politiche di scheduling senza implementarle nel SO. Essa usa una caratterizzazione matematica del tipico workload in un sistema per determinare il throughput del sistema o valori dei parametri chiave dello scheduler come la time slice e le misure delle tabelle di scheduling.

## Domande

- 7.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
- Se l'overhead di scheduling è trascurabile, la lunghezza della schedulazione è identica sia nei sistemi batch che in quelli multiprogrammati.
  - Se tutte le richieste arrivano allo stesso istante di tempo in un sistema che usa la politica di scheduling shortest job first (SJF) e il sistema termina l'esecuzione di queste richieste nella sequenza  $r_1, r_2, \dots, r_n$ , allora il tempo di completamento pesato di  $r_i$  è maggiore del tempo di completamento pesato di  $r_j$  se  $i > j$ .
  - La politica di scheduling round-robin con time-slicing fornisce approssimativamente uguali rapporti di risposta alle richieste che arrivano allo stesso istante.
  - Se i processi non eseguono I/O, la politica di scheduling round-robin con time-slicing è simile alla politica di scheduling least completed next (LCN).
  - Quando sono presenti sia richieste CPU-bound che I/O-bound, la politica di scheduling least completed next (LCN) fornisce migliori tempi di completamento per richieste I/O-bound di quelli che fornirebbe la politica di scheduling round-robin con time-slicing.
  - La politica di scheduling highest response ratio next (HRN) evita la starvation.
  - Se esiste una sequenza schedulabile per un'applicazione real-time, l'uso della politica di scheduling earliest deadline first (EDF) garantisce che non si verifica alcun deadline overrun.
  - Un processo I/O-bound è eseguito due volte, una volta in un sistema che usa lo scheduling RR e di nuovo in uno che usa lo scheduling multilivello adattivo. Il numero di volte che è schedulato dallo scheduler RR e da quello multilivello è identico.
  - Un processo CPU-bound non può andare in starvation quando è impiegato uno scheduling multilivello adattivo.
  - Se i processi non eseguono I/O, la politica di scheduling di Unix degenera in quella RR.
- 7.2. I processi A, B e C arrivano agli istanti 0, 1 e 2, rispettivamente. I processi non eseguono I/O e richiedono 5, 3 e 1 secondi di tempo di CPU. Si consideri che il tempo di commutazione di processi sia trascurabile. A quale istante il processo B viene completato se lo scheduler usa la politica shortest time to go (STG)?
- 8.
  - 4.
  - 5.
  - 9.
- 7.3. Quale delle seguenti politiche di scheduling fornirà il minimo tempo di completamento per un processo I/O-bound? (Nel sistema sono presenti sia richieste di I/O-bound che CPU-bound.)
- RR,
  - LCN,
  - Scheduling multilivello adattivo,
  - Nessuno di questi.
- 7.4. Quale delle seguenti politiche di scheduling fornirà il minimo tempo di completamento per un processo CPU-bound? (Nel sistema sono presenti sia richieste di I/O-bound che CPU-bound.)
- RR,
  - LCN,
  - Scheduling multilivello adattivo

## Problemi

- 7.1. Fornire esempi di conflitti tra politiche di scheduling relativi all'utente e al sistema.
- 7.2. Studiare le prestazioni delle politiche di scheduling nonpreemptive e preemptive sui processi descritti in [Tabella 7.2](#) supponendo che i loro tempi di arrivo siano 0, 1, 3, 7 e 10 secondi, rispettivamente. Tracciare il diagramma dei tempi analoghi a quelli nei Paragrafi 7.2 e 7.3 per mostrare il funzionamento di queste politiche.
- 7.3. Dimostrare che lo scheduling SJF fornisce il tempo di completamento medio minimo per un insieme di richieste che arrivano allo stesso istante. Fornirebbero il minimo tempo di completamento medio se le richieste arrivassero in tempi differenti?
- 7.4. Un programma contiene un unico loop che viene eseguito 50 volte. Il loop comprende un calcolo che impiega 50 ms seguito da un'operazione di I/O che impiega 200 ms. Dieci esecuzioni indipendenti di questo programma iniziano allo stesso istante. L'overhead di scheduling del kernel è 3 ms. Calcolare il tempo di risposta del primo processo nella prima e nelle successive iterazioni se:
  - a. la time slice è 50 ms;
  - b. la time slice è 20 ms.
- 7.5. Il kernel di un SO implementa la politica HRN preemptive come segue: ogni  $t$  secondi, i rapporti di risposta di tutti i processi vengono calcolati e il processo con il più alto rapporto di risposta è selezionato. Descrivere questa politica per valori grandi e piccoli di  $t$ . Inoltre, confrontarla con le seguenti politiche.
  - a. Politica Shortest time to go (STG).
  - b. Politica Least completed next (LCN).
  - c. Politica Round-robin con time-slicing (RR).
- 7.6. Un processo è composto da due parti che sono funzionalmente indipendenti l'una dall'altra. Si propone di separare le due parti e creare due processi per elaborarle. Identificare quelle politiche di scheduling per le quali l'utente riceverebbe il miglior servizio utente attraverso l'uso dei due processi invece dell'originario processo singolo.
- 7.7. Per ognuna delle politiche di scheduling discusse nei Paragrafi 7.2 e 7.3, un gruppo di 20 richieste viene elaborato con trascurabili overhead e viene determinato il tempo di completamento medio. Le richieste vengono ora organizzate arbitrariamente in due gruppi di 10 richieste ciascuno. Questi gruppi di richieste sono ora elaborati uno dopo l'altro tramite ognuna delle politiche di scheduling usate precedentemente e viene calcolato il tempo di completamento medio. Confrontare i due tempi di completamento medio per ogni politica di scheduling e indicare le condizioni per le quali le due potrebbero essere differenti.
- 7.8. Uno scheduler multilivello adattivo usa cinque livelli di priorità numerati da 1 a 5, essendo il livello 1 il livello di priorità più alta. La time slice per un livello di priorità è  $0.1 \times n$ , dove  $n$  è il numero di livelli. Secondo tale scheduling, ogni processo viene posto al livello 1 inizialmente. Un processo che richiede 5 secondi di tempo di CPU è elaborato tramite questo scheduler. Confrontare il tempo di risposta del processo e l'overhead totale di scheduling sostenuto se non ci sono altri processi nel sistema. Se il processo è elaborato tramite uno scheduler round-robin che usa un time slice di 0.1 secondi di CPU, quale sarebbe il tempo di risposta del processo e l'overhead totale di scheduling sostenuto?
- 7.9. Una politica di scheduling multilivello adattivo evita la starvation promuovendo un processo a un livello di priorità più alto se ha speso 3 secondi nel suo attuale livello di priorità senza essere schedulato. Commentare i vantaggi e svantaggi dei seguenti metodi di implementazione della promozione:
  - a. promuovere un processo al massimo livello di priorità;
  - b. promuovere un processo al livello successivo di priorità maggiore.
- 7.10. Il sistema Houston Automatic Spooling (HASP) era un sottosistema di scheduling usato negli IBM/360. HASP assegnava elevata priorità ai processi I/O-bound e bassa priorità ai processi CPU-bound. Un processo era classificato come CPU-bound o I/O-bound in base al suo comportamento recente in relazione alla time slice: era considerato un processo CPU-bound se usava interamente la sua time slice quando schedulato; altrimenti, era un processo I/O-bound. Per ottenere un buon throughput, HASP richiedeva che una percentuale fissata di processi nella coda di scheduling dovessero essere processi I/O-bound. Periodicamente, HASP rettificava la time slice per soddisfare questo requisito: la time slice veniva ridotta se un numero maggiore di processi rispetto a quanto voluto erano considerati I/O-bound, e incrementato se un minor numero di processi erano I/O-bound. Spiegare lo scopo della rettifica della time slice. Descrivere il funzionamento di HASP se la maggior parte dei processi nel

- sistema fossero stati (a) CPU-bound e (b) I/O-bound.
- 7.11. Commentare le similarità e le differenze tra:
- gli scheduling LCN e Unix;
  - gli scheduling HASP e multilivello adattivo (Problema 7.10).
- 7.12. Determinare le starting deadline per i processi dell'Esempio 7.11.
- 7.13. Un SO che utilizza una politica di scheduling preemptive assegna dinamicamente le priorità modificate. La priorità di un processo cambia a diversi rate in base al suo stato, come segue.
- $\alpha$  Rate di cambio di priorità quando un processo è *running*.
- $\beta$  Rate di cambio di priorità quando un processo è *ready*.
- $\gamma$  Rate di cambio di priorità quando un processo sta eseguendo un I/O.
- Notare che il rate di cambio di priorità può essere positivo, negativo o zero. Un processo ha priorità 0 quando viene creato. Un processo con un valore numerico della priorità maggiore possiede una priorità maggiore per lo scheduling.
- Commentare le proprietà delle politiche di scheduling in ognuno dei seguenti casi.
- $\alpha > 0, \beta = 0, \gamma = 0$ .
  - $\alpha = 0, \beta > 0, \gamma = 0$ .
  - $\alpha = \beta = 0, \gamma > 0$ .
  - $\alpha < 0, \beta = 0, \gamma = 0$ .
- Cambierà il comportamento delle politiche di scheduling se la priorità di un processo è posta a 0 ogni volta che viene schedulato?
- 7.14. Un processo di *background* dovrebbe funzionare in modo da non degradare significativamente il servizio fornito agli altri processi. Quale delle seguenti alternative si suggerisce per implementarlo?
- Assegnare la minima priorità al processo di background.
  - Attribuire un quanto minore al processo di background rispetto agli altri processi (Paragrafo 7.4.8).
- 7.15. Preparare uno schedule per il funzionamento dei processi periodici  $P_1 - P_3$  del Paragrafo 7.5.3, usando lo scheduling EDF.
- 7.16. Se il requisito di risposta dell'applicazione di [Figura 7.13](#) è 30 secondi e i tempi di servizio dei processi  $P_2 - P_5$  sono quelli mostrati in [Figura 7.13](#), qual è il massimo tempo di servizio di  $P_1$  per cui esiste un schedule ammissibile? Rispondere a questa domanda in due condizioni:
- nessuno dei processi esegue operazioni di I/O;
  - il processo  $P_2$  esegue I/O per 3 secondi, 2 secondi dei quali possono essere sovrapposti all'elaborazione del processo  $P_3$ .
- 7.17. I tempi di servizio di tre processi  $P_1, P_2$  e  $P_3$  sono 5 ms, 3 ms e 10 ms, rispettivamente;  $T_1 = 25$  ms e  $T_2 = 8$  ms. Qual è il valore minimo di  $T_3$  per cui la politica di scheduling rate monotonic rispetterebbe le deadline di tutti i processi?
- 7.18. Un sistema usa la politica di scheduling FCFS. Uguali richieste di elaborazione arrivano nel sistema al rate di 20 richieste al secondo. Si vuole che il tempo di attesa medio nel sistema non superi 2.0 secondi. Calcolare la durata di ciascuna richiesta in secondi di CPU.
- 7.19. Uguali richieste, ciascuna delle quali richiede 0.05 secondi di CPU, arrivano in un SO al rate di 10 richieste al secondo. Il kernel usa una coda dei ready a dimensione fissa. Una nuova richiesta viene inserita nella coda dei ready se la coda non è già piena, altrimenti la richiesta viene scartata. Quale dovrebbe essere la dimensione della coda dei ready se fosse scartato meno dell'1 per cento di richieste?
- 7.20. Il tasso medio di arrivo di richieste in un sistema che usa lo scheduling FCFS è 5 richieste al secondo. Il tempo medio di attesa per una richiesta è 3 secondi. Trovare il tasso di esecuzione medio.
- 7.21. Si definisce "breve" una richiesta il cui tempo di servizio è minore del 5 per cento di  $\frac{1}{\omega}$ . Calcolare il tempo di completamento per una breve richiesta in un sistema che utilizza la politica di scheduling HRN quando  $\alpha = 5$  e  $\omega = 8$ .

## Problemi avanzati

7.1. Si considerino i seguenti processi attivi in un sistema multiprogrammato:

| Processo | Tempo di arrivo | CPU-burst | Priorità |
|----------|-----------------|-----------|----------|
| $P_1$    | 1 ms            | 7 ms      | 3        |
| $P_2$    | 4 ms            | 5 ms      | 4        |
| $P_3$    | 6 ms            | 8 ms      | 5        |
| $P_4$    | 10 ms           | 6 ms      | 2        |
| $P_5$    | 11 ms           | 8 ms      | 1        |

Supponendo che il cambio di contesto duri 1 ms, si mostri l'ordine di esecuzione dei processi e quanto vale il tempo di attesa medio, il tempo di turnaround medio e il throughput per ciascuno dei seguenti algoritmi di scheduling: (a) priorità con prelazione (la priorità massima è 5) e (b) round-robin con quanto di tempo  $q = 3$  ms.

7.2. Si considerino i seguenti processi attivi in un sistema multiprogrammato:

| Processo | Tempo di arrivo | CPU-burst | Priorità |
|----------|-----------------|-----------|----------|
| $P_1$    | 4 ms            | 2 ms      | 3        |
| $P_2$    | 2 ms            | 8 ms      | 2        |
| $P_3$    | 4 ms            | 8 ms      | 1        |
| $P_4$    | 6 ms            | 3 ms      | 4        |
| $P_5$    | 9 ms            | 6 ms      | 5        |

Supponendo che il cambio di contesto duri 1 ms, si mostri l'ordine di esecuzione dei processi e quanto vale il tempo di attesa medio, il tempo di turnaround medio e il tempo di turnaround normalizzato per ciascuno dei seguenti algoritmi di scheduling: (a) Priorità con prelazione (la priorità massima è 1) e (b) round-robin con quanto di tempo  $q = 2$  ms.

## Note bibliografiche

Corbato et al. (1962) trattano l'uso di code multilivello con retroazione nel sistema operativo CTSS. Coffman e Denning (1973) riportano studi riguardanti lo scheduling multilivello. Uno scheduler fair share viene descritto in Kay e Lauder (1988), e lo scheduling a lotteria viene descritto in Waldspurger e Weihl (1994). Lo scheduling real-time è discusso in Liu e Layland (1973), Zhao (1989), Khanna et al. (1992), e Liu (2000). La conservazione dell'energia è un nuovo elemento cruciale nello scheduling. L'energia può essere conservata utilizzando la CPU a velocità inferiori. Zhu et al. (2004) discutono algoritmi di scheduling speculativo che conservano energia variando la velocità della CPU e riducendo il numero di cambiamenti di velocità assicurando nel contempo che un'applicazione rispetti i vincoli temporali.

Bach (1986), McKusick et al. (1996), e Vahalia (1996) trattano lo scheduling in Unix; O'Gorman (2003), Bovet e Cesati (2005), e Love (2005) trattano lo scheduling in Linux; Mauro e McDougall (2006) trattano lo scheduling in Solaris; mentre Russinovich e Solomon (2005) trattano lo scheduling in Windows.

Trivedi (1982) tratta la teoria delle code. Hellerman e Conroy (1975) descrivono l'uso della teoria delle code nella valutazione delle prestazioni.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
2. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol.
3. Brinch Hansen, P. (1972): *Operating System Principles*, Prentice Hall, Englewood Cliffs, N.J.
4. Coffman, E. G., and R. C. Wood (1996): "Interarrival Statistics for time sharing Systems," *Communications of The ACM*, 39(7), 500-503.
5. Coffman, E.G., and P.J. Denning (1973): *Operating Systems Theory*, Prentice Hall, Englewood Cliffs, N.J.

6. Corbato, F.J., M. Merwin-Daggett, and R.C. Daley (1962): "An experimental time-sharing system," *Proceedings of the AFIPS Fall Joint Computer Conference*, 335-344.
7. Hellerman, H., and T.F. Conroy (1975): *Computer System Performance*, McGraw-Hill Kogakusha, Tokyo.
8. Kay, J., and P. Lauder (1988): "A fair share scheduler," *Communications of the ACM*, **31** (1), 44-55.
9. Khanna, S., M. Sebree, and J. Zolnowsky (1992): "Real-time scheduling in SunOS 5.0," *Proceedings of the Winter 1992 USENIX Conference*, San Francisco, January 1992, 375-390.
10. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
11. Liu, C.L., and J.W. Layland (1973): "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, **20**, 1, 46-61.
12. Liu, J.W.S. (2000): *Real-Time Systems*, Pearson Education, New York.
13. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
14. McKusick, M.K., K. Bostic, M.J. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, Reading, Mass.
15. O'Gorman, J. (2003): *Linux Process Manager: The Internals of Scheduling, Interrupts and Signals*, John Wiley, New York.
16. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
17. Trivedi, K.S. (1982): *Probability and Statistics with Reliability - Queuing and Computer Science Applications*, Prentice Hall, Englewood Cliffs, N.J.
18. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.
19. Waldspurger, C.A., and W.E. Weihl (1994): "Lottery scheduling," *Proceedings of the First USENIX Symposium on Operating System Design and Implementation (SODI)*, 1-11.
20. Zhao, W. (1989): Special issue on real-time operating systems, *Operating System Review*, **23**, 7.
21. Zhu, D., D. Mosse, and R. Melhem (2004): "Power-aware scheduling for AND/OR graphs in real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, **15** (9), 849-864.

---

# CAPITOLO 8

## Deadlock

---

### Obiettivi di apprendimento

- Definizione di deadlock
- Gestione dei deadlock: rilevamento, risoluzione, prevenzione
- Modellazione dei deadlock tramite grafi
- Gestione dei deadlock nei sistemi operativi: Unix e Windows

Nella vita reale, si verifica uno stallo quando due persone sono in attesa di una telefonata reciproca, oppure quando persone, che camminano su una scala stretta in direzioni opposte, si incontrano e non riescono a passare contemporaneamente. Uno stallo (*deadlock*) è caratterizzato dal fatto che le persone attendono indefinitamente che un altro esegua determinate azioni che potrebbero non verificarsi mai.

I deadlock in un sistema operativo si verificano in modo analogo; essi appaiono quando si verifica che alcuni processi attendono indefinitamente le azioni da parte di qualche altro processo. I deadlock si verificano nella sincronizzazione quando i processi attendono segnali l'uno dall'altro, oppure nella condivisione delle risorse quando i processi attendono che altri processi rilascino le risorse di cui hanno bisogno. I processi in deadlock rimangono bloccati indefinitamente, il che pregiudica il servizio per l'utente, il throughput e l'efficienza delle risorse.

Nell'ambito della condivisione delle risorse, i deadlock si realizzano se si verificano contemporaneamente un insieme di condizioni riguardanti le richieste e le allocazioni di risorse. I sistemi operativi usano diversi approcci per la gestione dei deadlock che fondamentalmente indichiamo di seguito. In uno dei possibili approcci, indicato come *individuare e risolvere i deadlock*, il kernel controlla se si verificano contemporaneamente le condizioni di deadlock ed elimina il deadlock interrompendo opportunamente alcuni processi, in modo che i rimanenti non risultino più in deadlock. In un differente approccio denominato *prevenire i deadlock*, il kernel utilizza politiche di allocazione delle risorse assicurando che non si verifichino simultaneamente le condizioni di deadlock, rendendoli impossibili. In un terzo possibile approccio, detto *evitare i deadlock*, il kernel non effettua allocazioni di risorse che possano creare un deadlock, che quindi non potranno verificarsi.

Discussiamo questi approcci di gestione dei deadlock e le politiche di allocazione delle risorse utilizzate nei sistemi operativi.

### 8.1 Definizione di deadlock

Un *deadlock* si ha quando in un insieme di processi esistono processi dell'insieme che attendono un evento che deve essere generato da un altro processo dell'insieme. Ogni processo è dunque in attesa di un evento che potrebbe non verificarsi mai. L'Esempio 8.1 illustra come potrebbe crearsi un deadlock quando due processi tentano di condividere delle risorse.

#### Esempio 8.1 - Deadlock di due processi

Un sistema contiene un'unità nastro e una stampante. Due processi  $P_i$  e  $P_j$  utilizzano l'unità nastro e la stampante tramite i seguenti programmi:

| Processo $P_i$  | Processo $P_j$  |
|---|---|
| Richiesta unità nastro;<br>Richiesta stampante;<br>Uso unità nastro e stampante;<br>Rilascio stampante;<br>Rilascio unità nastro; | Richiesta stampante;<br>Richiesta unità nastro;<br>Uso unità nastro e stampante;<br>Rilascio unità nastro;<br>Rilascio stampante; |

Quando i due processi vanno in esecuzione, le richieste di risorse seguono il seguente ordine:

1. processo  $P_i$  richiede l'unità nastro;
2. processo  $P_j$  richiede la stampante;
3. processo  $P_i$  richiede la stampante;
4. processo  $P_j$  richiede l'unità nastro.

Le prime due richieste di risorse vengono concesse direttamente perché il sistema include sia l'unità nastro che la stampante. Ora,  $P_i$  ha allocata l'unità nastro e  $P_j$  ha allocata la stampante. Quando  $P_i$ , chiede la stampante, risulta bloccata fin quando  $P_j$  la rilascerà. Allo stesso modo,  $P_j$  è bloccato fin quando  $P_i$  rilascerà l'unità nastro. Entrambi i processi sono bloccati indefinitamente perché ognuno è in attesa della risorsa allocata all'altro.

Il deadlock illustrato nell'Esempio 8.1 è riferito al *deadlock generato da risorse* (resource deadlock). In un SO si possono verificare anche altri tipi di deadlock. Per esempio, un *deadlock di sincronizzazione* si verifica quando gli eventi attesi sono sotto forma di segnali tra processi. Per esempio, se un processo  $P_i$  decide di eseguire un'azione  $a_i$  solo dopo che il processo  $P_j$  ha eseguito l'azione  $a_j$  e il processo  $P_j$  decide di eseguire l'azione  $a_j$  solo dopo che  $P_i$  ha eseguito  $a_i$ , entrambi i processi sono bloccati finché l'altro processo non gli invia un segnale (Paragrafo 6.4). Analogamente, si verifica un *deadlock di comunicazione* tra un insieme di processi se ogni processo invia un messaggio solo dopo aver ricevuto un messaggio da qualche altro processo dell'insieme (Capitolo 9). Il deadlock di risorse riguarda principalmente il SO essendo l'allocazione delle risorse una responsabilità del SO. Le altre due forme di deadlock sono gestite raramente dal SO dato che ci si aspetta che i processi utente gestiscano tali deadlock.

Formalmente, diciamo che un deadlock si verifica se sono soddisfatte le condizioni nella definizione che segue.

**Definizione 8.1 Deadlock** Una situazione che coinvolge un insieme di processi  $D$  in cui ogni processo  $P_i$  in  $D$  soddisfa due condizioni.

1. Il processo  $P_i$  è bloccato da qualche evento  $e_j$ .
2. l'evento  $e_j$  può essere causato solo da azioni prodotte da un altro o più processi in  $D$ .

Nel caso di deadlock, ogni processo in grado di generare l'evento  $e_j$  atteso dal processo  $P_i$  appartiene a  $D$ . Questa proprietà fa sì che risulta impossibile che l'evento  $e_j$  si verifichi; di conseguenza ogni processo  $P_i$  di  $D$  aspetta indefinitamente.

## 8.2 I deadlock nell'allocazione delle risorse

I processi utilizzano risorse hardware, quali la memoria, i dispositivi di I/O e risorse software, quali i file. Un SO può avere a disposizione diverse risorse di una stessa tipologia, cioè diversi dischi, unità nastro o stampanti. Usiamo il termine *unità di risorsa* per identificare una risorsa di un tipo specifico, e usiamo il termine *classe di risorse* per indicare l'insieme di tutte le unità di risorsa di un determinato tipo. Pertanto, una classe di risorsa contiene una o più unità di risorsa; per esempio, la classe delle stampanti può contenere due o più stampanti. Usiamo la notazione  $R_i$  per una classe di risorsa e  $r_j$  per un'unità di risorsa in una classe di risorsa. Ricordiamo dal Paragrafo 1.3.2 che il kernel

gestisce una tabella delle risorse per tenere traccia dello stato di allocazione di una risorsa.

L'allocazione di una risorsa in un sistema genera tre tipi di eventi: la *richiesta* per la risorsa, l'*Allocazione* attuale della risorsa e il *rilascio* della risorsa. La [Tabella 8.1](#) descrive tali eventi. Si verifica un evento di richiesta quando qualche processo  $P_i$  effettua una richiesta di una risorsa  $r_l$ . Il processo  $P_i$  sarà bloccato sull'evento di allocazione di  $r_l$ , se  $r_l$  è attualmente allocata a un processo  $P_k$ . In effetti,  $P_i$  è in attesa che  $P_k$  rilasci  $r_l$ . Un evento di rilascio da parte di  $P_k$  libera la risorsa  $r_l$  e il kernel può decidere di allocare la risorsa  $r_l$  a  $P_i$ . Quindi, un evento di rilascio da parte di  $P_k$  può causare l'evento di allocazione del quale è in attesa  $P_i$ ; in questo caso  $P_i$  diventerà *holder* (possessore) della risorsa e entrerà nello stato *ready*. Tuttavia, come abbiamo visto nell'[Esempio 8.1](#), il processo  $P_i$  andrà incontro a un'attesa indefinita se il rilascio di  $r_l$  da parte di  $P_k$  viene ritardato indefinitamente.

| Evento      | Descrizione  |
|-------------|--|
| Richiesta   | Un processo richiede una risorsa tramite una chiamata di sistema. Se la risorsa è libera, il kernel la alloca al processo immediatamente; altrimenti, cambia lo stato del processo a <i>blocked</i> .  |
| Allocazione | Il processo diventa <i>holder</i> della risorsa a esso allocata. Le informazioni sullo stato della risorsa vengono aggiornate e lo stato del processo diventa <i>ready</i> .   |
| Rilascio    | Un processo rilascia una risorsa tramite una chiamata di sistema. Se vari processi sono bloccati sull'evento allocazione della risorsa, il kernel usa alcune regole, come l'allocazione FCFS, per decidere a quale processo allocare la risorsa. |

**Tabella 8.1** Eventi relativi all'allocazione di risorse.

### 8.2.1 Condizioni per un deadlock di risorsa

Riformulando leggermente le condizioni 1 e 2 della [Definizione 8.1](#), otteniamo le condizioni sotto le quali si verifica un deadlock di risorsa: (1) ogni processo  $P_i$  in  $D$  è bloccato da un evento di allocazione che deve verificarsi e (2) l'evento di allocazione può essere causato solo dalle azioni di qualche altro processo  $P_j$  appartenente a  $D$ . Poiché  $P_j$  appartiene a  $D$ , le condizioni 1 e 2 della [Definizione 8.1](#) sono valide anche a  $P_j$ . In altre parole, la risorsa richiesta dal processo  $P_i$  è attualmente allocata a  $P_j$ , ma esso stesso è in attesa che gli venga allocata qualche risorsa. Questa condizione per ciascun processo è detta condizione *possesso e attesa* (*hold-and-wait*).

Ma le parti 1 e 2 della [Definizione 8.1](#) implicano anche che i processi in  $D$  si debbano attendere a vicenda. Questa condizione è detta condizione di *attesa circolare* (circular wait). Un'attesa circolare può essere diretta, cioè  $P_i$  attende  $P_j$  e  $P_j$  attende a sua volta  $P_i$  oppure può verificarsi tramite altri (uno o più) processi appartenenti a  $D$ , per esempio  $P_i$  attende  $P_j$ ,  $P_j$  attende  $P_k$  e  $P_k$  attende  $P_i$ .

Devono verificarsi altre due condizioni per far sì che un deadlock di risorsa si verifichi. Se il processo  $P_i$  necessita una risorsa attualmente allocata a  $P_j$ ,  $P_i$  non deve essere in grado di (1) condividere la risorsa con  $P_j$  o (2) prelevarla da  $P_j$  per utilizzarla.

La [Tabella 8.2](#) riassume le condizioni che devono essere soddisfatte per l'esistenza di un deadlock di risorsa. Tutte queste condizioni devono verificarsi simultaneamente: un'attesa circolare è essenziale per un deadlock, una condizione hold-and-wait è essenziale per un'attesa circolare e la mutua esclusione e l'assenza di prelazione delle risorse sono essenziali per una condizione hold-and-wait.

| Condizione                                   | Descrizione   |
|--|---|
| Risorse non condivisibili o mutua esclusione | Risorse che non possono essere condivise; un processo necessita di accesso esclusivo alla risorsa.  |
| Assenza di prelazione                        | Una risorsa non può essere prelevata di autorità da un processo e allocata a un altro.  |
| Possesso e attesa                            | Un processo continua a tenere la risorsa allocata in attesa di altre risorse.   |
| Attesa circolare                             | Esiste nel sistema una catena circolare di condizioni possesso e attesa; per esempio, il processo $P_i$ aspetta $P_j$ , $P_j$ aspetta $P_k$ e $P_k$ aspetta $P_i$ . |

**Tabella 8.2** Condizioni per un deadlock di risorse.

Oltre alle condizioni elencate in [Tabella 8.2](#), un'altra condizione è essenziale per i deadlock:

- *nessun ritiro di richieste di risorse*: un processo bloccato su una richiesta di risorsa non può ritirare la sua richiesta.

Questa condizione è essenziale perché l'attesa potrebbe non essere indefinita se a un processo bloccato fosse consentito di ritirare una richiesta di risorsa e continuare la sua esecuzione. Comunque, non è stato affermato esplicitamente in letteratura, perché molti sistemi operativi tipicamente non consentono ai processi di ritirare le richieste di risorse.

### 8.2.2 Modelli dello stato di allocazione delle risorse

L'Esempio 8.1 mostra come sia necessario analizzare le informazioni relative alle risorse allocate ai processi e quelle relative alle richieste di risorse in attesa per stabilire se un insieme di processi è in deadlock. Tutte queste informazioni costituiscono lo *stato di allocazione delle risorse* di un sistema, chiamato semplicemente *stato di allocazione* di un sistema.

Per rappresentare lo stato di allocazione di un sistema si utilizzano due tipi di modelli. Un *modello basato su grafo* può raffigurare lo stato di allocazione di una classe ristretta di sistemi in cui un processo può richiedere e usare esattamente un'unità di risorsa di ogni classe di risorsa. Il modello consente l'uso di un semplice algoritmo sui grafi per determinare se la condizione di attesa circolare è soddisfatta dai processi. Un *modello basato su matrice* ha il vantaggio di essere generale. Tale modello può rappresentare lo stato di allocazione in sistemi che consentono a un processo di richiedere qualsiasi numero di unità di una classe di risorsa.

#### Modelli basati su grafo

Un *grafo di richiesta e allocazione risorse* (RRAG) contiene due tipi di nodi: i nodi processo e i nodi risorsa. Un *nodo processo* è rappresentato da un cerchio, mentre un *nodo risorsa* è rappresentato da un rettangolo e rappresenta una classe di risorse. Il numero nel nodo risorsa indica quante unità di quella classe di risorse esistono nel sistema. Sono previsti due tipi di archi tra un nodo processo e un nodo risorsa di un RRAG. Un *arco di allocazione* è diretto da un nodo risorsa a un nodo processo e indica che un'unità della classe di risorsa è allocata al processo. Un *arco di richiesta* è diretto da un nodo processo a un nodo risorsa e indica che il processo è bloccato su una richiesta per un'unità della classe di risorse. Un arco di allocazione  $(R_k, P_j)$  viene cancellato quando il processo  $P_j$  rilascia un'unità di risorsa della classe di risorsa  $R_k$  a esso allocata. Quando è concessa una richiesta in attesa del processo  $P_i$  per un'unità di una classe di risorsa  $R_k$  l'arco di richiesta  $(P_i, R_k)$  viene cancellato e viene aggiunto l'arco di allocazione  $(R_k, P_i)$ .

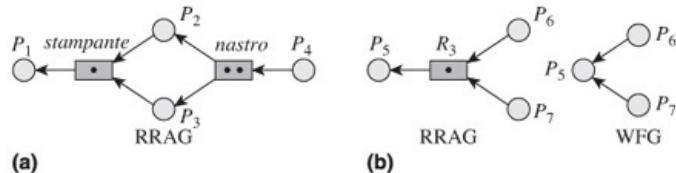
Un *grafo di attesa* (grafo wait-for o WFG) può rappresentare lo stato di allocazione in modo più conciso di un RRAG se ogni classe di risorsa nel sistema contiene solo un'unità di risorsa. Il WFG contiene i nodi solo di un tipo, detti, nodi processo. Un arco  $(P_i, P_j)$  nel WFG rappresenta il fatto che il processo  $P_i$ , è bloccato su una richiesta per una risorsa attualmente allocata al processo  $P_j$ ; o, equivalentemente, il processo  $P_i$  sta attendendo che il processo  $P_j$  rilasci una risorsa. Da cui, il nome grafo *waitfor*. Rappresentare le

stesse informazioni in un RRAG avrebbe richiesto due archi. L'esempio successivo illustra e confronta l'uso di un RRAG e di un WFG.

### Esempio 8.2 - RRAG e WFG

La [Figura 8.1](#) (a) mostra un RRAG. La classe delle stampanti contiene soltanto una unità di risorsa, che è allocata al processo  $P_1$ . Sono in attesa le richieste per una stampante effettuate da parte dei processi  $P_2$  e  $P_3$ . La classe delle unità nastro contiene due unità nastro, che sono allocate ai processi  $P_2$  e  $P_3$ . È in attesa una richiesta da parte del processo  $P_4$  per una unità nastro.

La [Figura 8.1](#) (b) mostra un RRAG e un WFG per un sistema che ha una classe di risorsa  $R_3$  che contiene soltanto una unità di risorsa e tre processi  $P_5$ ,  $P_6$  e  $P_7$ . Gli archi  $(P_6, P_3)$  e  $(P_3, P_5)$  nel RRAG indicano complessivamente che il processo  $P_6$  è in attesa della risorsa attualmente allocata a  $P_5$ . Pertanto, è presente l'arco  $(P_6, P_5)$  nel WFG. Analogamente, l'arco  $(P_7, P_5)$  indica che il processo  $P_7$  è in attesa della risorsa attualmente allocata  $P_5$ .



**Figura 8.1** (a) Grafo di richiesta e allocazione di risorsa (RRAG); (b) Un grafo RRAG e un grafo di attesa (WFG) equivalenti quando ogni classe di risorsa contiene soltanto una unità di risorsa.

### Percorsi nei RRAG e nei WFG

Possiamo dedurre la presenza di deadlock dalla natura dei percorsi nei RRAG e nei WFG. A tale scopo, introduciamo la seguente notazione:

|              |   |
|--------------|---|
| $Blocked\_P$ | insieme dei processi bloccati;  |
| $WF_i$       | l'insieme delle attese di $P_i$ , cioè l'insieme dei processi che detengono la risorsa richiesta dal processo $P_i$ . |

Con questa notazione, le Condizioni 1 e 2 della Definizione 8.1 possono essere riformulate come segue:

$$D \subseteq Blocked\_P \quad (8.1)$$

$$\text{per tutti i } P_i \in D, WF_i \subseteq D \quad (8.2)$$

Consideriamo un sistema in cui ogni classe di risorsa contiene soltanto una unità di risorsa. Supponiamo che il sistema contenga un singolo percorso  $P_1 - R_1 - P_2 - R_2 \dots P_{n-1} - R_{n-1} - P_n$  nel suo RRAG. Quindi, il processo  $P_n$  non è bloccato su alcuna risorsa e nessuna risorsa è attualmente allocata a  $P_1$ . Il WFG di questo sistema dovrebbe contenere il solo percorso  $P_1 - P_2 - \dots - P_n$ .

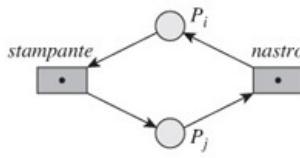
Stabiliamo l'assenza di un deadlock in questo sistema mostrando che le Condizioni 8.1 e 8.2 non sono soddisfatte per qualsiasi insieme di processi nel sistema.  $Blocked\_P$  è  $\{P_1, \dots, P_{n-1}\}$ . Dapprima, consideriamo l'insieme di processi  $\{P_1, \dots, P_n\}$ . Questo insieme non è un sottoinsieme di  $Blocked\_P$ , violando la Condizione 8.1, e dunque questo insieme di processi non è in deadlock. Ora, consideriamo l'insieme  $\{P_1, \dots, P_{n-1}\}$ . In questo caso,  $WF_{n-1} = \{P_n\}$  viola la Condizione 8.2. Analogamente, può essere dimostrato che qualsiasi altro sottoinsieme di  $\{P_1, \dots, P_n\}$  viola la Condizione 8.2 per qualche processo; di conseguenza, non c'è alcun deadlock nel sistema.

Se l'unità della classe di risorsa  $R_{n-1}$  fosse stata allocata a  $P_1$  invece che a  $P_n$ , il

percorso nel RRAG sarebbe stato  $P_1 - R_1 - P_2 - R_2 \dots P_{n-1} - R_{n-1} - P_1$ . Questo è un percorso ciclico, anche detto *ciclo*, perché termina allo stesso nodo dal quale inizia, ossia, il nodo  $P_1$ . Il WFG contiene inoltre un ciclo, dato da  $P_1 - P_2 \dots - P_1$ .  $Blocked\_P$  è  $\{P_1, \dots, P_{n-1}\}$ , lo stesso di prima. Esiste pertanto un deadlock perché l'insieme  $\{P_1, \dots, P_{n-1}\}$  soddisfa entrambe le Condizioni 8.1 e 8.2, infatti:

- $\{P_1, \dots, P_{n-1}\} \subseteq Blocked\_P$ ;
- per tutti i  $P_i \in \{P_1, \dots, P_{n-1}\}$ ,  $WF_i$  contiene un solo processo  $P_1$  tale che  $P_1 \in \{P_1, \dots, P_{n-1}\}$ .

Da questa analisi possiamo concludere che la Condizione 8.2, che implica l'esistenza di relazioni di attesa reciproche tra i processi di  $D$ , può essere soddisfatta solo da percorsi ciclici. In sintesi, un deadlock non può esistere senza un RRAG, o un WFG che contiene un ciclo.



**Figura 8.2** RRAG per il sistema dell'Esempio 8.1.

### Esempio 8.3 - Un RRAG con deadlock

La [Figura 8.2](#) mostra il RRAG dell'Esempio 8.2. Il RRAG contiene un percorso ciclico  $P_i$ -stampante- $P_j$ -nastro- $P_i$ . Qui  $WF_i = \{P_j\}$  e  $WF_j = \{P_i\}$ .  $D = \{P_1, P_2\}$  soddisfa entrambe le Condizioni 8.1 e 8.2. Di conseguenza, i processi  $P_i$  e  $P_j$  sono in deadlock.

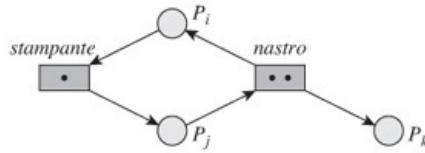
La presenza di un ciclo in un RRAG o in un WFG implica un deadlock? Nel sistema finora discusso, ogni classe di risorsa conteneva una singola unità di risorsa, quindi un ciclo nel RRAG o nel WFG implicava un deadlock. Comunque, può non essere così in tutti i sistemi. Consideriamo per esempio un percorso  $P_1 - R_1 \dots P_i - R_i - P_{i+1} - \dots P_n$  in un sistema in cui una classe di risorsa  $R_i$  contiene più di un'unità di risorsa. Non può essere utilizzato un WFG per raffigurare lo stato di allocazione di questo sistema; di conseguenza, per questo sistema è necessario utilizzare un RRAG. Se qualche processo  $P_k$ , non incluso nel percorso, è in possesso di un'unità della classe di risorsa  $R_i$ , tale unità può essere allocata a  $P_i$  quando  $P_k$  la rilascia. L'arco  $(P_i, R_i)$  potrebbe quindi essere eliminato anche senza che  $P_{i+1}$  rilasci l'unità di  $R_i$ , che possiede.

Quindi, un percorso ciclico in un RRAG può essere interrotto quando qualche processo non incluso nel ciclo rilascia una unità della risorsa. In sintesi, se una classe di risorsa contiene più di una unità di risorsa, la presenza di un ciclo in un RRAG non necessariamente implica l'esistenza di un deadlock. Nel Paragrafo 8.8 faremo riferimento a questi concetti per formulare una caratterizzazione formale dei deadlock. L'Esempio 8.4 illustra tale situazione.

### Esempio 8.4 - Un ciclo in un RRAG non implica un deadlock

Si consideri un sistema con una stampante, due unità nastro e tre processi  $P_i$ ,  $P_j$  e  $P_k$ . La natura dei processi  $P_i$  e  $P_j$  è la stessa descritta nell'Esempio 8.1, cioè ognuno di essi richiede un nastro e una stampante. Il processo  $P_k$  richiede soltanto un'unità nastro per il suo funzionamento. Supponiamo che il processo  $P_k$  richieda un'unità nastro prima delle richieste 1-4 come nell'Esempio 8.1.

La [Figura 8.3](#) mostra il RRAG dopo che sono state eseguite tutte le richieste. Il grafo ha un ciclo che include  $P_i$  e  $P_j$ . Questo ciclo sarà interrotto quando il processo  $P_k$  rilascerà la risorsa perché l'unità nastro da esso rilasciata verrà allocata a  $P_j$ . Di conseguenza non c'è alcun deadlock. Giungiamo alla stessa conclusione quando analizziamo l'insieme dei processi  $\{P_i, P_j\}$  in base alla Definizione 8.1 perché  $WF_j = \{P_i, P_k\}$  e  $P_k \notin \{P_i, P_j\}$  violano la Condizione 8.2.



**Figura 8.3** RRAG dopo che sono state eseguite tutte le richieste dell'Esempio 8.4.

### Modello basato su matrice

Nel modello basato su matrice, lo stato di allocazione di un sistema è rappresentato principalmente da due matrici. La matrice delle *Risorse\_allocate* indica quante unità di risorsa di ogni classe di risorse sono allocate ad ogni processo del sistema. La matrice *Risorse\_richieste* tiene conto delle richieste in attesa: indica quante unità di risorsa di ogni classe di risorse sono state richieste da ogni processo del sistema. Se un sistema contiene  $n$  processi ed  $r$  classi di risorse, ognuna di queste matrici è una matrice  $n \times r$ . Lo stato di allocazione rispetto a una classe di risorsa  $R_k$  indica il numero di unità di  $R_k$  allocate per ogni processo, e il numero di unità di  $R_k$  richieste da ogni processo. Queste sono rappresentate come n-uple ( $Risorse\_allocate_{1,k}, \dots, Risorse\_allocate_{n,k}$ ) e ( $Risorse\_richieste_{1,k}, \dots, Risorse\_richieste_{n,k}$ ).

Inoltre, vengono usate altre matrici ausiliarie per rappresentare informazioni aggiuntive richieste per scopi specifici. Due di queste matrici sono *Risorse\_totali* e *Risorse\_disponibili*, che indicano, rispettivamente, il numero totale di unità di risorse in ogni classe di risorse e il numero di unità di risorse di ogni classe di risorse che sono libere. Ognuna di queste matrici è una matrice colonna che possiede  $r$  elementi. L'Esempio 8.5 è un esempio di modello basato su matrice.

### Esempio 8.5 - Modello a matrice dello stato di allocazione

Usando il modello basato su matrice, lo stato di allocazione del sistema di [Figura 8.3](#) è rappresentato come segue:

| Stampante        |   | Nastro |   | Stampante |   | Nastro            |   | Stampante |   | Nastro |   |
|------------------|---|--------|---|-----------|---|-------------------|---|-----------|---|--------|---|
| $P_i$            | 0 |        | 1 | $P_i$     | 1 |                   | 0 | $P_i$     | 1 |        | 2 |
| $P_j$            | 1 |        | 0 | $P_j$     | 0 |                   | 1 | $P_j$     | 0 |        | 0 |
| $P_k$            | 0 |        | 1 | $P_k$     | 0 |                   | 0 | $P_k$     | 0 |        | 0 |
| Risorse allocate |   |        |   |           |   | Risorse richieste |   |           |   |        |   |

Le relazioni di attesa del sistema non sono rappresentate con il modello a matrice; devono essere dedotte tramite un algoritmo. Nei paragrafi che seguono descriviamo gli Algoritmi 8.1 e 8.2 che usano il modello basato su matrice.

## 8.3 Gestione dei deadlock

La [Tabella 8.3](#) descrive i tre approcci fondamentali alla gestione dei deadlock. Ogni approccio ha comportamenti differenti in termini di possibili ritardi nell'allocazione delle risorse, di tipo di richieste di risorse che i processi utente possono fare e di overhead di SO.

| Approccio                              | Descrizione   |
|--|---|
| Rilevamento e risoluzione dei deadlock | Il kernel analizza lo stato della risorsa per controllare se esiste un deadlock. Se è così, interrompe alcuni processi e assegna le risorse allocate ad altri processi affinché il deadlock cessi di esistere.  |
| Prevenzione dei deadlock               | Il kernel usa una politica di allocazione delle risorse che assicura che le quattro condizioni per i deadlock di risorsa menzionate in <a href="#">Tabella 8.2</a> non si verifichino simultaneamente. Questo rende impossibili i deadlock.   |
| Evitare i deadlock                     | Il kernel analizza lo stato di allocazione per determinare se l'accettazione di una richiesta di risorsa può determinare un deadlock. Vengono accettate solo richieste che non conducono a deadlock, le altre vengono tenute in attesa finché possono essere accettate. In questo modo, non si verificano deadlock. |

**Tabella 8.3** Approcci alla gestione dei deadlock.

Nell'approccio *individuazione e risoluzione dei deadlock*, il kernel interrompe alcuni processi quando individua un deadlock analizzando lo stato di allocazione. Questa azione libera le risorse assegnate al processo interrotto, che possono essere in tal modo allocate ad altri processi che ne hanno fatto richiesta. I processi interrotti devono essere rieseguiti. Il costo di questo approccio include pertanto il costo dell'individuazione dei deadlock e quello della riesecuzione dei processi interrotti. Nel sistema dell'Esempio 8.1, il kernel dovrebbe individuare un deadlock dopo l'esecuzione della quarta richiesta. Questo deadlock può essere risolto interrompendo  $P_i$  oppure  $P_j$  e allocando la risorsa allocata all'altro processo.

Nell'approccio *prevenire i deadlock*, il kernel adotta una politica di allocazione delle risorse che rende impossibili i deadlock: i processi devono attenersi a ogni restrizione che la politica impone. Per esempio, una semplice politica di prevenzione dei deadlock consisterebbe nell'allocare tutte le risorse richieste da un processo allo stesso tempo. Questa politica richiede che un processo effettui le sue richieste di risorse tutte insieme. Nell'Esempio 8.1, entrambi i processi richiedono sia una stampante che un'unità nastro allo stesso istante. Non si verificherebbe un deadlock se uno dei processi ottenessesse entrambe le risorse di cui necessita; d'altra parte, la politica dovrebbe forzare un processo a ottenere una risorsa molto prima che sia effettivamente necessario.

Nell'approccio *evitare i deadlock*, il kernel accetta una richiesta di risorsa solo se constata che, accettando la richiesta, non si arriverà a un deadlock successivamente; altrimenti, mantiene la richiesta in attesa finché può essere accettata. Ne consegue che un processo può andare incontro a lunghi ritardi per ottenere una risorsa. Nell'Esempio 8.1, il kernel realizza la possibilità del verificarsi di un deadlock mentre elabora la seconda richiesta; pertanto, non concede la stampante al processo  $P_j$  finché il processo  $P_i$  non termina.

## 8.4 Individuazione e risoluzione dei deadlock

Consideriamo un sistema in cui sono presenti un processo  $P_i$  che detiene una stampante e un processo  $P_j$  che è bloccato sulla richiesta che ha avanzato per una stampante. Se il processo  $P_i$  non è in stato *blocked*, c'è la possibilità che esso possa completare la sua operazione senza richiedere più risorse; a compimento, rilascerebbe la stampante allocata per sé, che potrebbe allora essere allocata al processo  $P_j$ . Così, se  $P_i$  non è in stato *blocked*, l'attesa di  $P_j$  per la stampante non è indefinita a causa della seguente sequenza di eventi: il processo  $P_i$  termina, rilascia la stampante e la stampante è allocata a  $P_j$ . Se qualche altro processo  $P_l$  attende qualche altra risorsa allocata a  $P_j$ , la sua attesa non è indefinita. Di conseguenza i processi  $P_i$ ,  $P_j$  e  $P_l$  non sono coinvolti in un deadlock.

Da questa osservazione, possiamo formulare la seguente regola per l'individuazione dei deadlock: un processo in stato *blocked* non è coinvolto in un deadlock se la richiesta su cui è bloccato può essere soddisfatta tramite una sequenza di completamento del processo, rilascio della risorsa ed eventi di allocazione delle risorse. Se ogni classe di

risorse nel sistema contiene una sola unità di risorsa, questo controllo può essere fatto verificando la presenza di un ciclo in un RRAG o WFG. Tuttavia, se le classi di risorse possono contenere più di una unità di risorse devono essere utilizzati algoritmi molto complessi basati sui grafi (Paragrafo 8.8), per cui discutiamo un approccio all'individuazione dei deadlock usando il modello basato su matrice.

Controlliamo la presenza di un deadlock in un sistema cercando di costruire in modo efficiente sequenze fittizie ma fattibili di eventi in base ai quali *tutti* i processi bloccati possono avere le risorse che hanno richiesto. Riuscire con successo a costruire una tale sequenza implica l'assenza di un deadlock all'istante attuale; un fallimento, invece, implica la presenza di un deadlock. Quando applichiamo questa regola agli Esempi 8.3 e 8.4, si deduce correttamente che i processi  $P_i$ , e  $P_j$  dell'Esempio 8.3 sono in un deadlock, e che invece nell'Esempio 8.4 non c'è deadlock.

Eseguiamo i controlli sopra esposti simulando il funzionamento di un sistema che parte con il suo stato corrente. Ci riferiamo a qualsiasi processo che non è bloccato su una richiesta di risorsa come un processo *running*, ossia non distinguiamo tra gli stati *ready* e *running* tipici dei processi. Nella simulazione consideriamo soltanto due eventi: completamento di un processo che non è bloccato su una richiesta di risorsa e allocazione di risorsa(e) per un processo che è bloccato su una richiesta di risorsa. Si assume che un processo *running* terminerà senza effettuare ulteriori richieste di risorse e che alcune delle risorse rilasciate al suo termine verranno allocate a un processo *blocked* solo se l'allocazione metterà il processo in stato *running*. La simulazione finisce quando tutti i processi *running* terminano. I processi che sono in stato *blocked* al termine della simulazione sono quelli che non hanno ottenuto le risorse richieste quando gli altri processi hanno terminato la propria esecuzione; di conseguenza questi processi sono in deadlock allo stato attuale. Non c'è deadlock allo stato attuale se non esiste alcun processo *blocked* quando la simulazione termina. L'Esempio 8.4 illustra questo approccio.

#### Esempio 8.6 - Individuazione dei deadlock

Si consideri il seguente stato di allocazione di un sistema con 10 unità di una classe di risorse  $R_1$ , e tre processi  $P_1$ ,  $P_2$  e  $P_3$ :

|       | $R_1$            | $R_1$             |                | $R_1$ |
|-------|------------------|-------------------|----------------|-------|
| $P_1$ | 4                | 6                 | Risorse totali | 10    |
| $P_2$ | 4                | 2                 | Risorse libere | 0     |
| $P_3$ | 2                | 0                 |                |       |
|       | Risorse allocate | Risorse richieste |                |       |

Il processo  $P_3$  è in stato *running* perché non è bloccato da una richiesta di risorse. Tutti i processi nel sistema possono terminare nel seguente modo: il processo  $P_3$  termina e rilascia 2 unità della risorsa a esso allocata. Queste unità possono essere allocate a  $P_2$ . Quando questo termina, 6 unità della risorsa possono essere allocate a  $P_1$ . Così, non c'è alcun processo *blocked* quando termina la simulazione, quindi non c'è deadlock nel sistema.

Se le richieste dai processi  $P_1$  e  $P_2$  fossero per 6 e 3 unità, rispettivamente, nessuno di loro potrebbe terminare anche dopo che il processo  $P_3$  ha rilasciato 2 unità di risorsa. Questi processi sarebbero in stato *blocked* al termine della simulazione, e quindi sarebbero in deadlock allo stato attuale del sistema.

Nella nostra simulazione, abbiamo assunto che un processo *running* termini la sua esecuzione senza effettuare ulteriori richieste di risorse. Questa assunzione ha due conseguenze. Primo, le nostre conclusioni relative all'esistenza di un deadlock non dipendono dall'ordine in cui assumiamo che i processi *blocked* diventino *running* o dall'ordine in cui assumiamo che i processi *running* terminino. Secondo, anche se un sistema è libero da deadlock (deadlock-free) all'istante attuale, si potrebbe verificare un deadlock successivamente. Nell'Esempio 8.4, questo potrebbe accadere se  $P_3$  effettua una richiesta per un'altra unità di  $R_1$ . Di conseguenza, si deve, ripetutamente, eseguire l'individuazione dei deadlock durante il funzionamento del SO. Ciò si può realizzare dedicando un processo di sistema esclusivamente all'individuazione dei deadlock e attivandolo a intervalli prefissati. Alternativamente, l'individuazione dei deadlock può essere eseguita ogni volta che un processo diventa *blocked* su una richiesta di risorsa.

L'overhead per l'individuazione dei deadlock dipenderebbe da diversi fattori come il numero di processi e delle classi di risorse nel sistema e la frequenza di esecuzione.

### 8.4.1 Un algoritmo di individuazione dei deadlock

L'Algoritmo 8.1 consente l'individuazione dei deadlock. Gli input dell'algoritmo sono due insiemi di processi *Blocked* e *Running*, e un modello basato su matrice dello stato di allocazione comprendente le matrici *Risorse\_allocate*, *Risorse\_richieste* e *Risorse\_libere*.

L'algoritmo simula il completamento di un processo running  $P_i$  trasferendolo dall'insieme *Running* all'insieme *Exit* [Passi 1(a), 1(b)]. Le risorse allocate per  $P_i$  vengono aggiunte alle *Risorse\_libere* [Passo 1(c)]. L'algoritmo seleziona ora un processo dell'insieme *Blocked* la cui richiesta di risorsa può essere soddisfatta dalle risorse libere [Passo 1(d)], e lo trasferisce dall'insieme *Blocked* all'insieme *Running*. Successivamente l'algoritmo simula il suo completamento e lo trasferisce da *Running* a *Exit*. L'algoritmo termina quando non è rimasto alcun processo nell'insieme *Running*. I processi che rimangono nell'insieme *Blocked*, se ce ne sono, sono in deadlock.

La complessità dell'algoritmo può essere analizzata come segue: gli insiemi *Running* e *Blocked* possono contenere fino a  $n$  processi, dove  $n$  è il numero totale di processi del sistema. Il loop del Passo 1 itera al più  $n$  volte e il Passo 1(d) esegue un numero di operazioni dell'ordine di  $n \times r$  a ogni iterazione. Di conseguenza, l'algoritmo richiede una complessità dell'ordine di  $n^2 \times r$ . L'Esempio 8.7 illustra il funzionamento di questo algoritmo.

#### Algoritmo 8.1 Individuazione dei deadlock

##### Input

|                          |   |                                |
|--------------------------|---|--------------------------------|
| $n$                      | : | numero di processi;            |
| $r$                      | : | numero di classi di risorse;   |
| <i>Blocked</i>           | : | insieme di processi;           |
| <i>Running</i>           | : | insieme di processi;           |
| <i>Risorse.libere</i>    | : | array [1..r] of integer;       |
| <i>Risorse_allocate</i>  | : | array [1..n, 1..r] of integer; |
| <i>Risorse_richieste</i> | : | array [1..n, 1..r] of integer; |

##### Strutture dati

|             |   |                      |
|-------------|---|----------------------|
| <i>Exit</i> | : | insieme di processi; |
|-------------|---|----------------------|

1. **repeat until** l'insieme *Running* è vuoto
  - a. seleziona un processo  $P_i$  dall'insieme *Running*;
  - b. cancella  $P_i$  dall'insieme *Running* e aggiungilo all'insieme *Exit*;
  - c. **for**  $k=1..r$ 

$$\text{Risorse.Libere}[k] = \text{Risorse.Libere}[k] + \text{Risorse.allocate}[i, k];$$
  - d. **while** l'insieme *Blocked* contiene un processo  $P_j$  tale che
    - for**  $k = 1..r$ ,  $\text{Risorse.richieste}[j, k] \leq \text{Risorse.libere}[k]$ 
      - i. **for**  $k = 1, r$ 

$$\text{Risorse.Libere}[k] = \text{Risorse.Libere}[k] - \text{Risorse.richieste}[j, k];$$

$$\text{Risorse.allocate}[j, k] = \text{Risorse.allocate}[j, k] + \text{Risorse.richieste}[j, k];$$
      - ii. Rimuovi  $P_j$  dall'insieme *Blocked* e aggiungilo all'insieme *Running*;
  2. **if** l'insieme *Blocked* è non vuoto **then**  
imposta i processi nell'insieme *Blocked* come processi in deadlock.

#### Esempio 8.7 - Funzionamento dell'algoritmo di individuazione dei deadlock

Un sistema ha quattro processi  $P_1, P_2, P_3$  e  $P_4$  e 5, 7 e 5 unità nelle classi di risorsa  $R_1, R_2$  ed  $R_3$  rispettivamente. Si trova nel seguente stato appena prima che il processo  $P_3$  effettui una richiesta di una unità della classe di risorsa  $R_1$ :

|       | $R_1$            | $R_2$ | $R_3$ |  | $R_1$             | $R_2$ | $R_3$ |   | $R_1$          | $R_2$ | $R_3$ |   |
|-------|------------------|-------|-------|--|-------------------|-------|-------|---|----------------|-------|-------|---|
| $P_1$ | 2                | 1     | 0     |  | $P_1$             | 2     | 1     | 3 |                | 5     | 7     | 5 |
| $P_2$ | 1                | 3     | 1     |  | $P_2$             | 1     | 4     | 0 |                |       |       |   |
| $P_3$ | 0                | 1     | 1     |  | $P_3$             |       |       |   |                |       |       |   |
| $P_4$ | 1                | 2     | 2     |  | $P_4$             | 1     | 0     | 2 |                | 1     | 0     | 1 |
|       | Risorse allocate |       |       |  | Risorse richieste |       |       |   | Risorse totali |       |       |   |

Una unità di risorsa della classe di risorsa  $R_1$  è allocata al processo  $P_3$  ed è utilizzato l'Algoritmo 8.1 per controllare se il sistema è in deadlock. La Figura 8.4 mostra i passi del funzionamento dell'algoritmo. Gli insiemi *Blocked* e *Running* sono i suoi input inizializzati a  $\{P_1, P_2, P_4\}$  e  $\{P_3\}$ , rispettivamente, e le matrici *Risorse\_allocate*, *Risorse\_richieste* e *Risorse\_libere* sono mostrate in Figura 8.4(a). L'algoritmo trasferisce il processo  $P_3$  nell'insieme *Exit* libera le risorse a esso allocate. Il numero delle unità libere delle classi di risorsa è ora 1, 1 e 2, rispettivamente. L'algoritmo verifica che ora può essere soddisfatta la richiesta in attesa del processo  $P_4$ , quindi alloca le risorse richieste da  $P_4$  e trasferisce  $P_4$  nell'insieme dei *Running* [Figura 8.4(b)]. Poiché  $P_4$  è il solo processo in *Running*, viene trasferito nell'insieme *Exit*. Dopo aver liberato le risorse di  $P_4$ , l'algoritmo verifica che ora può essere soddisfatta la richiesta di risorsa di  $P_1$  [Figura 8.4(c)] e, dopo che  $P_1$  termina, può essere soddisfatta la richiesta di risorsa di  $P_2$  [Figura 8.4(d)]. Ora l'insieme *Running* è vuoto, quindi l'algoritmo termina. Nel sistema non esiste deadlock perché l'insieme *Blocked* risulta vuoto.

(a) Stato iniziale

|       | $R_1$            | $R_2$ | $R_3$ |  | $R_1$             | $R_2$ | $R_3$ |   | $R_1$          | $R_2$ | $R_3$ |   |
|-------|------------------|-------|-------|--|-------------------|-------|-------|---|----------------|-------|-------|---|
| $P_1$ | 2                | 1     | 0     |  | $P_1$             | 2     | 1     | 3 |                | 0     | 0     | 1 |
| $P_2$ | 1                | 3     | 1     |  | $P_2$             | 1     | 4     | 0 |                |       |       |   |
| $P_3$ | 0                | 1     | 1     |  | $P_3$             |       |       |   |                |       |       |   |
| $P_4$ | 1                | 2     | 2     |  | $P_4$             | 1     | 0     | 2 |                |       |       |   |
|       | Risorse allocate |       |       |  | Risorse richieste |       |       |   | Risorse libere |       |       |   |

(b) Dopo la simulazione dell'assegnazione delle risorse a  $P_4$  quando termina il processo  $P_3$

|       | $R_1$            | $R_2$ | $R_3$ |  | $R_1$             | $R_2$ | $R_3$ |   | $R_1$          | $R_2$ | $R_3$ |   |
|-------|------------------|-------|-------|--|-------------------|-------|-------|---|----------------|-------|-------|---|
| $P_1$ | 2                | 1     | 0     |  | $P_1$             | 2     | 1     | 3 |                | 0     | 1     | 0 |
| $P_2$ | 1                | 3     | 1     |  | $P_2$             | 1     | 4     | 0 |                |       |       |   |
| $P_3$ | 0                | 0     | 0     |  | $P_3$             |       |       |   |                |       |       |   |
| $P_4$ | 2                | 2     | 4     |  | $P_4$             |       |       |   |                |       |       |   |
|       | Risorse allocate |       |       |  | Risorse richieste |       |       |   | Risorse libere |       |       |   |

(c) Dopo la simulazione dell'assegnazione delle risorse a  $P_1$  quando termina il processo  $P_4$

|       | $R_1$            | $R_2$ | $R_3$ |  | $R_1$             | $R_2$ | $R_3$ |   | $R_1$          | $R_2$ | $R_3$ |   |
|-------|------------------|-------|-------|--|-------------------|-------|-------|---|----------------|-------|-------|---|
| $P_1$ | 4                | 2     | 3     |  | $P_1$             |       |       |   |                | 0     | 2     | 1 |
| $P_2$ | 1                | 3     | 1     |  | $P_2$             | 1     | 4     | 0 |                |       |       |   |
| $P_3$ | 0                | 0     | 0     |  | $P_3$             |       |       |   |                |       |       |   |
| $P_4$ | 0                | 0     | 0     |  | $P_4$             |       |       |   |                |       |       |   |
|       | Risorse allocate |       |       |  | Risorse richieste |       |       |   | Risorse libere |       |       |   |

(d) Dopo la simulazione dell'assegnazione delle risorse a  $P_2$  quando termina il processo  $P_1$

|       | $R_1$            | $R_2$ | $R_3$ |  | $R_1$             | $R_2$ | $R_3$ |  | $R_1$          | $R_2$ | $R_3$ |   |
|-------|------------------|-------|-------|--|-------------------|-------|-------|--|----------------|-------|-------|---|
| $P_1$ | 0                | 0     | 0     |  | $P_1$             |       |       |  |                | 3     | 0     | 4 |
| $P_2$ | 2                | 7     | 1     |  | $P_2$             |       |       |  |                |       |       |   |
| $P_3$ | 0                | 0     | 0     |  | $P_3$             |       |       |  |                |       |       |   |
| $P_4$ | 0                | 0     | 0     |  | $P_4$             |       |       |  |                |       |       |   |
|       | Risorse allocate |       |       |  | Risorse richieste |       |       |  | Risorse libere |       |       |   |

Figura 8.4 Funzionamento dell'Algoritmo 8.1 di individuazione dei deadlock.

## 8.4.2 Risoluzione dei deadlock

Dato un insieme  $D$  di processi in stato di deadlock, la *risoluzione dei deadlock* comporta l'interruzione del deadlock per assicurare l'avanzamento di alcuni processi di  $D$ , ossia

per processi appartenenti a qualche sottoinsieme  $D' \in D$ . Ciò può essere realizzato interrompendo uno o più processi nell'insieme  $D$  e allocando le loro risorse a qualche processo in  $D'$ . Ogni processo interrotto è detto *vittima* della risoluzione del deadlock.

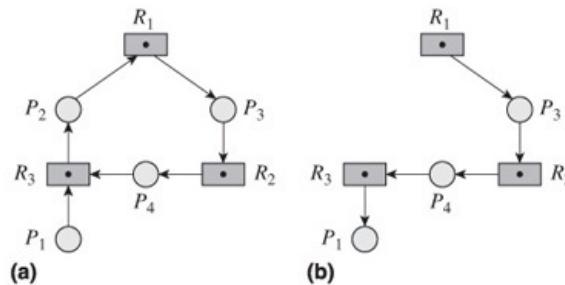
Quindi, la risoluzione dei deadlock può essere vista come l'azione di dividere un insieme  $D$  di processi in deadlock in due insiemi tali che  $D = D' \cup D_v$ , dove:

- ogni processo in  $D_v$  è una vittima della risoluzione del deadlock;
- l'insieme  $D'$  dei processi deadlock-free terminano dopo le azioni di risoluzione dei deadlock. Ossia, ogni processo in  $D'$  può terminare il suo funzionamento attraverso una sequenza di completamento di processo, rilascio risorse ed eventi di allocazione delle risorse.

La scelta del processo vittima viene fatta tramite criteri quali la priorità di un processo, le risorse già utilizzate, ecc. Il prossimo esempio illustra la risoluzione dei deadlock.

#### Esempio 8.8 - Risoluzione dei deadlock

Il RRAG di [Figura 8.5\(a\)](#) mostra un deadlock che coinvolge i processi  $P_1, P_2, P_3$  e  $P_4$ . Questo deadlock è risolto scegliendo il processo  $P_2$  come vittima. La parte (b) della figura mostra il RRAG dopo l'interruzione del processo  $P_2$  e l'allocazione al processo  $P_1$  della risorsa  $R_3$  precedentemente allocata a  $P_2$ . Il processo  $P_4$ , che era in attesa della vittima prima della risoluzione del deadlock, ora attende  $P_1$ , il nuovo possessore della risorsa. Quanto detto è importante per l'individuazione dei deadlock futuri. Se lo stato di allocazione è rappresentato dal modello basato su matrice, è sufficiente cancellare le righe corrispondenti a  $P_2$  in *Risorse\_allocate* e *Risorse\_richieste*, modificare le righe del processo  $P_1$  e modificare *Risorse\_libere* di conseguenza.



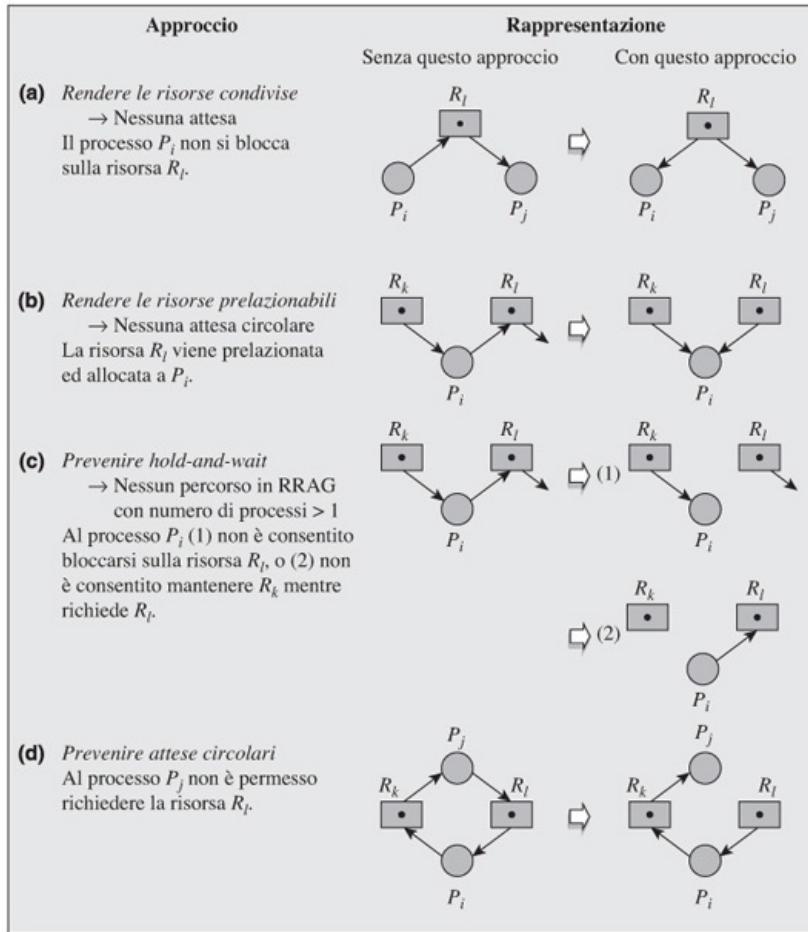
**Figura 8.5** Risoluzione dei deadlock: (a) un deadlock; (b) lo stato di allocazione delle risorse dopo la risoluzione dei deadlock.

## 8.5 Prevenzione dei deadlock

Le quattro condizioni descritte in [Tabella 8.2](#) devono verificarsi contemporaneamente affinché si abbia un deadlock di risorsa in un sistema. Per prevenire i deadlock, il kernel deve utilizzare una politica di allocazione di risorse che assicuri che una di queste condizioni non possa verificarsi. In questo paragrafo, verranno dapprima presentati differenti approcci alla prevenzione dei deadlock e poi illustriamo alcune politiche di allocazione di risorse che utilizzano questi approcci.

### Condivisione di risorse

In un sistema non esisterebbero relazioni di attesa se tutte le risorse potessero essere rese condivisibili. Questi percorsi in un RRAG conterrebbero solo archi di allocazione, quindi non si avrebbero mai attese circolari. La [Figura 8.6\(a\)](#) illustra l'effetto dell'utilizzo di questo approccio: l'arco di richiesta ( $P_i, R_l$ ) sarebbe sostituito da un arco di allocazione ( $R_l, P_i$ ) se l'unità di risorsa della classe  $R_l$  fosse condivisibile.



**Figura 8.6** Approcci alla prevenzione dei deadlock.

Tuttavia, alcune risorse come le stampanti sono intrinsecamente non condivisibili, quindi come potrebbero essere rese condivisibili? I SO usano alcune tecniche innovative per risolvere questo problema. Un esempio si trova nel sistema multiprogrammato THE degli anni '60. THE gestiva una sola stampante, quindi bufferizzava l'output prodotto dai differenti processi, lo formattava per produrre "immagini di pagina" e usava la stampante per stampare un'immagine di pagina alla volta. Questa soluzione confondeva le pagine prodotte dai diversi processi e quindi l'output dei diversi processi doveva essere separato manualmente. (Il motivo per cui il sistema THE eseguiva la formattazione della pagina non era la prevenzione dei deadlock, ma migliorare l'utilizzazione della stampante. Infatti, il sistema THE non disponeva di criteri per la gestione dei resource deadlock.) La non condivisibilità di un dispositivo può essere superata anche creando dispositivi virtuali (Paragrafo 1.3.2); ossia possono essere create stampanti virtuali allocate ai processi. Tuttavia, questo approccio non può funzionare per risorse software come file condivisi, che dovrebbero essere modificati in maniera mutuamente esclusiva per evitare race condition.

### Prelazione di risorse

Se le risorse sono rese prelazionabili, il kernel può assicurare che i processi abbiano tutte le risorse di cui necessitano, il che evita percorsi circolari nei RRAG. Per esempio, in [Figura 8.6\(b\)](#) la risorsa  $R_l$  può essere prelazionata al processo che attualmente la possiede e allocata al processo  $P_i$ . Tuttavia, l'assenza di prelazione delle risorse può essere elusa soltanto selettivamente. L'approccio di formattazione della pagina del sistema THE può essere usata per rendere le stampanti prelazionabili, ma, in generale, i dispositivi sequenziali di I/O non possono essere prelazionati.

### Hold-and-wait

Per evitare la condizione relativa al possesso e attesa, detta hold-and-wait, è necessario che un processo che abbia acquisito delle risorse non possa effettuare richieste di risorse, oppure che un processo bloccato su una richiesta di risorsa non possa poter impegnare altre risorse. Così, in [Figura 8.6\(c\)](#), o non compare l'arco  $(P_i, R_l)$ , o non esiste l'arco  $(R_k, P_l)$  se compare  $(P_i, R_l)$ . In entrambi i casi, i percorsi del RRAG che coinvolgono più di un processo non possono comparire, quindi non possono esistere percorsi circolari. Una semplice politica per l'implementazione di questo approccio è consentire a un processo di effettuare solo una richiesta di risorse durante la propria esecuzione in cui chiede tutte le risorse di cui necessita. Discutiamo questa politica nel [Paragrafo 8.5.1](#).

### Attesa circolare

Un'attesa circolare può dipendere dalla condizione hold-and-wait, che a sua volta è conseguenza delle condizioni di mutua esclusione e assenza di prelazione; quindi, non si verifica se non si verifica nessuna di queste condizioni. Le attese circolari possono essere evitate separatamente non consentendo ad alcuni processi di attendere determinate risorse; per esempio, non può essere consentito al processo  $P_j$  in [Figura 8.6\(d\)](#) di attendere la risorsa  $R_l$ . Ciò può essere ottenuto applicando un *vincolo di validità* a ogni richiesta di risorsa. Il vincolo di validità è una funzione booleana dello stato di allocazione. Assume valore *falso* se la richiesta può condurre a un'attesa circolare nel sistema e, quindi, una tale richiesta è subito rifiutata. Se il vincolo di validità assume valore *vero*, la risorsa è allocata se è disponibile; altrimenti, il processo è bloccato in attesa che si liberi la risorsa. Nel [Paragrafo 8.5.2](#) discutiamo una politica di prevenzione dei deadlock che considera questo approccio.

|                    |   |                     |   |                      |   |                    |    |
|--------------------|---|---------------------|---|----------------------|---|--------------------|----|
| $P_1$              | 8 | $P_1$               | 3 | $P_1$                | 1 | Totale<br>allocate | 7  |
| $P_2$              | 7 | $P_2$               | 1 | $P_2$                | 0 | Risorse<br>totali  | 10 |
| $P_3$              | 5 | $P_3$               | 3 | $P_3$                | 0 |                    |    |
| Risorse<br>massime |   | Risorse<br>allocate |   | Risorse<br>richieste |   |                    |    |
|                    |   |                     |   |                      |   |                    |    |

**Figura 8.7** Uno stato di allocazione nell'algoritmo del banchiere per una sola classe di risorse.

## 8.5.1 Allocazione globale

Questa è la più semplice tra le politiche di prevenzione dei deadlock. Un processo deve richiedere tutte le risorse di cui ha necessità in un'unica richiesta e così il kernel le alloca tutte insieme. In questo modo, ad un processo bloccato non viene allocata alcuna risorsa, quindi la condizione hold-and-wait non è mai soddisfatta. Di conseguenza, le attese circolari e i deadlock non si possono verificare. Con questa politica, entrambi i processi dell'[Esempio 8.1](#) devono richiedere insieme un'unità nastro e una stampante. Ora un processo o acquisirà entrambe le risorse o nessuna, e la condizione hold-and-wait non sarà soddisfatta.

La semplicità di implementazione rende questa una politica interessante per piccoli sistemi operativi. Tuttavia, presenta una controindicazione dal punto di vista pratico: pregiudica l'efficienza delle risorse. Per esempio, se un processo  $P_i$  richiede un'unità nastro all'inizio della sua esecuzione e una stampante solo verso la fine della sua esecuzione, sarà costretto a richiedere sia l'unità nastro che la stampante all'inizio. La stampante rimarrà idle fino alla seconda parte dell'esecuzione di  $P_i$  e qualsiasi processo che richiede una stampante sarà ritardato finché  $P_i$  non termina la sua esecuzione. Questa situazione riduce, inoltre, il grado di multiprogrammazione e perciò riduce l'efficienza della CPU.

## 8.5.2 Ranking delle risorse

Con questa politica di prevenzione dei deadlock, un numero identificativo chiamato *rank della risorsa* è associato a ogni classe di risorsa. Quando un processo  $P_i$  effettua una richiesta di una risorsa, il kernel applica un vincolo di validità per decidere se la richiesta deve essere considerata. Il vincolo di validità assume il valore *vero* solo se il rank della

risorsa richiesta è maggiore del rank massimo della risorsa attualmente allocata per  $P_i$ . In questo caso, la risorsa è allocata a  $P_i$  se è disponibile; altrimenti,  $P_i$  viene bloccato in attesa che la risorsa venga rilasciata. Se il vincolo di validità è *falso*, la richiesta viene rifiutata e il processo  $P_i$ , che ha effettuato la richiesta, sarà interrotto.

L'assenza di relazioni di attesa circolari in un sistema che utilizza il ranking delle risorse può essere spiegata in questo modo. Sia  $rank_k$  il rank assegnato alla classe di risorsa  $R_k$ , e sia  $P_i$ , il processo a cui sono allocate alcune unità della classe di risorsa  $R_k$ .  $P_i$  può essere bloccato su una richiesta per una unità di una classe di risorsa  $R_l$  solo se  $rank_k < rank_l$ . Consideriamo ora il processo  $P_j$  a cui sono allocate alcune unità della classe di risorsa  $R_l$ . Il processo  $P_j$  non può richiedere un'unità della classe di risorsa  $R_k$  poiché  $rank_k > rank_l$ . Quindi, se  $P_i$ , può attendere che  $P_j$ , rilasci la risorsa,  $P_j$  non può aspettare  $P_i$ ! Di conseguenza i due processi non possono trovarsi in una condizione di attesa circolare. Un'analoga considerazione si può fare in merito all'assenza di un'attesa circolare che coinvolge un numero maggiore di processi.

L'Esempio 8.9 illustra il funzionamento della politica di ranking delle risorse.

### Esempio 8.9 - Ranking delle risorse

Nell'Esempio 8.1, sia  $rank_{stampante} > rank_{nastro}$ . La Richiesta 1 è di allocazione dell'unità nastro per  $P_i$ , la Richiesta 2 è di allocazione della stampante per  $P_j$ . La Richiesta 3, cioè la richiesta della stampante da parte di  $P_i$ , soddisfa il vincolo di validità perché  $rank_{stampante} > rank_{nastro}$ , ma rimane in attesa perché la stampante non è disponibile. La Richiesta 4 sarà rifiutata poiché viola il vincolo di validità e il processo  $P_j$  verrà interrotto. Questa azione rilascerà la stampante, che sarà, dunque, allocata a  $P_i$ .

La politica del ranking delle risorse funziona al meglio quando tutti i processi richiedono le risorse in ordine crescente di rank. Tuttavia, le difficoltà nascono quando un processo richiede una risorsa che ha un rank minore. L'unico modo in cui può avere questa risorsa è rilasciando prima la risorsa con maggiore rank. Infatti, nel-l'Esempio 8.9, il processo  $P_j$  può avere il nastro rilasciando prima la stampante e, avendo il nastro allocato, richiede un'altra volta la stampante. Ciò è difficile in pratica poiché molte risorse sono non prelazionabili. I processi possono tentare di eludere queste difficoltà acquisendo risorse con minori rank molto prima che ne abbiano effettivamente bisogno. Per esempio, il processo  $P_j$  dell'Esempio 8.1 potrebbe acquisire l'unità nastro prima di acquisire la stampante. Nel peggiore dei casi, questa politica può degenerare nella politica di allocazione globale delle risorse.

Nonostante questi svantaggi, la politica del ranking delle risorse è interessante per la sua semplicità. Un kernel può usare questa politica per le sue richieste di risorse quando ha bisogno di risorse in un ordine prefissato. Discuteremo questo aspetto nel Paragrafo 8.9.

## 8.6 Evitare i deadlock

La politica di evitare i deadlock ammette una richiesta di risorsa solo se si riesce a stabilire che l'accettazione della richiesta non può condurre a un deadlock né immediatamente né successivamente. Ma nasce una questione ovvia: l'Algoritmo 8.1 descritto nel Paragrafo 8.4 può essere usato per controllare se ammettendo la richiesta di risorsa ne risulta un deadlock immediatamente, ma come potrebbe il kernel sapere se un deadlock può verificarsi successivamente?

Il kernel non ha una conoscenza dettagliata del comportamento futuro dei processi, quindi non può prevedere i deadlock in modo accurato. Per semplificare la tecnica per evitare i deadlock sotto tali condizioni, viene utilizzato il seguente approccio conservativo: ogni processo dichiara il massimo numero di unità di risorsa di ogni classe che può richiedere. Il kernel permette che un processo richieda queste unità di risorsa a stadi successivi, ossia, poche unità di risorsa alla volta, in base al massimo numero dichiarato dal processo, e utilizza una tecnica di analisi nel caso peggiore per controllare la possibilità di deadlock successivi. Una richiesta è ammessa solo se non c'è possibilità di deadlock; altrimenti, rimane in attesa finché può essere ammessa. Questo approccio è conservativo perché un processo può terminare il suo funzionamento senza richiedere il massimo numero dichiarato di unità. Così, il kernel può rimandare l'ammissione di

alcune richieste di risorse che avrebbe ammesso immediatamente se avesse conosciuto il comportamento successivo dei processi. Questo effetto e l'overhead per il controllo a ogni richiesta di risorsa rappresentano il costo per evitare il deadlock. Nel seguito è descritto un ben noto algoritmo, denominato l'algoritmo del banchiere, che adotta tale approccio.

La [Tabella 8.4](#) descrive la notazione dell'algoritmo del banchiere.  $Risorse_{massime}_{j,k}$  indica il massimo numero di unità di risorsa della classe di risorsa  $R_k$  che un processo  $P_j$  può richiedere. Il kernel ammette il processo  $P_j$  solo se  $Risorse_{massime}_{j,k} \leq Risorse_{totali}_k$  per tutti i  $k$ . Il kernel può ammettere un qualsiasi numero di processi che soddisfano questo criterio di ammissione. Così  $\sum_j Risorse_{massime}_{j,k}$  può superare  $Risorse_{totali}_k$ .  $Risorse_{allocate}_{j,k}$  indica l'attuale numero di unità di risorsa della classe di risorsa  $R_k$  allocate a  $P_j$ , e  $Risorse_{totali\_allocate}_k$  indica quante unità della classe di risorsa  $R_k$  sono allocate ai processi al momento. L'algoritmo del banchiere evita i deadlock assicurando che il sistema sia in ogni momento in uno stato di allocazione tale che tutti i processi possano terminare il loro funzionamento senza possibilità di deadlock. È chiamato algoritmo del banchiere perché i banchieri hanno bisogno di un algoritmo simile quando ammettono prestiti che collettivamente superano i fondi bancari e rilasciano ogni prestito a rate.

| Notazione                      | Significato   |
|--------------------------------|---|
| $Risorse_{richieste}_{j,k}$    | Numero di unità della classe di risorsa $R_k$ attualmente richieste dal processo $P_j$                  |
| $Risorse_{massime}_{j,k}$      | Massimo numero di unità della classe di risorsa $R_k$ di cui può aver bisogno il processo $P_j$         |
| $Risorse_{allocate}_{j,k}$     | Numero di unità della classe di risorsa $R_k$ allocate al processo $P_j$                                |
| $Risorse_{totali\_allocate}_k$ | Numero totale di unità allocate della classe di risorsa $R_k$ , ossia $\sum_j Risorse_{allocate}_{j,k}$ |
| $Risorse_{totali}_k$           | Numero totale di unità della classe di risorsa $R_k$ appartenenti al sistema                            |

**Tabella 8.4** Notazione usata nell'algoritmo del banchiere.

L'algoritmo del banchiere usa la nozione di *stato di allocazione sicuro* per assicurare che l'ammissione di una richiesta di risorsa *non possa* portare a un deadlock né immediatamente né successivamente.

**Definizione 8.2 Stato di allocazione sicuro**

È uno stato di allocazione in cui è possibile costruire una sequenza di completamento di processo, rilascio risorse, ed eventi di allocazione risorse attraverso cui ogni processo  $P_j$  nel sistema può ottenere  $Risorse_{massime}_{j,k}$  risorse per ogni classe di risorse  $R_k$  e terminare il suo funzionamento.

Le tecniche per evitare i deadlock sono implementate trasferendo il sistema da uno stato di allocazione sicuro a un altro stato di allocazione sicuro come descritto di seguito:

1. quando un processo effettua una richiesta, si calcola il nuovo stato di allocazione in cui si troverebbe il sistema se la richiesta fosse ammessa. Chiameremo questo stato: *stato di allocazione proiettato*;
2. se lo stato di allocazione proiettato è uno stato di allocazione sicuro, si ammette la richiesta aggiornando le matrici  $Risorse_{allocate}$  e  $Risorse_{totali}$ ; altrimenti, si tiene la richiesta in attesa;
3. quando un processo rilascia una o più risorse o termina la propria esecuzione, si esaminano tutte le richieste in attesa e si allocano quelle che porterebbero il sistema in un nuovo stato di allocazione sicuro.

L'algoritmo del banchiere determina la sicurezza dello stato di allocazione di una risorsa

tentando di costruire una sequenza di completamento di processo, rilascio risorse ed eventi di allocazione risorse attraverso cui tutti i processi possano terminare. Ciò può essere realizzato tramite la simulazione come indicato nel Paragrafo 8.4, a meno di una modifica: il completamento di un processo  $P_l$ , se è in stato *Running* o *Blocked*, può richiedere ( $Risorse\_massime_{l,k} - Risorse\_allocate_{l,k}$ ) ulteriori unità di risorse di ogni classe di risorsa  $R_k$ , quindi l'algoritmo controlla se

$$\begin{aligned} \text{per tutti } R_k: \quad & Risorse\_totali_k - Risorse\_totali\_allocate_k \geq \\ & Risorse\_massime_{l,k} - Risorse\_allocate_{l,k} \end{aligned} \quad (8.3)$$

Quando questa condizione è soddisfatta, viene simulato il completamento del processo  $P_l$  e il rilascio di tutte le risorse a esso allocate aggiornando  $Risorse\_totali\_allocate_k$  per ogni  $R_k$ . Viene, poi, controllato se qualche altro processo può soddisfare l'Equazione 8.3, e così via. Il prossimo esempio illustra questo metodo in un sistema che ha una sola classe di risorse. Notare che, come nell'individuazione dei deadlock, la determinazione della sicurezza di uno stato di allocazione non dipende dall'ordine in cui si assume che i processi completino il loro funzionamento.

#### **Esempio 8.10 - L'Algoritmo del banchiere per un'unica classe di risorse**

Un sistema contiene 10 unità di una classe di risorsa  $R_k$ . Le richieste massime di risorse effettuate da parte dei tre processi  $P_1$ ,  $P_2$  e  $P_3$  sono, rispettivamente, 8, 7 e 5 unità di risorsa, e le loro allocazioni attuali sono, rispettivamente, 3, 1 e 3 unità di risorsa. La [Figura 8.7](#) rappresenta lo stato attuale di allocazione del sistema. Il processo  $P_1$  effettua ora una richiesta di un'unità di risorsa. Nello stato di allocazione proiettato,  $Risorse\_totali\_allocate = 8$ , quindi, nel sistema, ci saranno due unità libere della classe di risorsa  $R_k$ .

La sicurezza dello stato proiettato è garantita dal fatto che  $P_3$  soddisfa la Condizione 8.3 poiché è esattamente sotto di due unità rispetto alle sue richieste massime. Di conseguenza le due unità di risorsa disponibili possono essere allocate a  $P_3$  se successivamente le richiede e può terminare. A questo punto saranno disponibili cinque unità di risorsa per l'allocazione, quindi le richieste di  $P_1$  di quattro unità di risorse possono essere soddisfatte e può terminare. Ora, sono disponibili tutte le unità di risorsa nel sistema per  $P_2$ , così anch'esso può terminare. Quindi, lo stato di allocazione proiettato è sicuro e di conseguenza, l'algoritmo accetterà la richiesta effettuata da parte di  $P_1$ .

La nuova allocazione per i processi è 4, 1 e 3 unità di risorsa e  $Risorse\_totali\_allocate_k = 8$ . Consideriamo ora le seguenti richieste:

1.  $P_1$  effettua una richiesta di 2 unità di risorsa;
2.  $P_2$  effettua una richiesta di 2 unità di risorsa;
3.  $P_3$  effettua una richiesta di 2 unità di risorsa.

Le richieste di  $P_1$  e  $P_2$  non mettono il sistema in stati di allocazione sicuri perché la Condizione 8.3 non è soddisfatta da tutti i processi, quindi queste richieste non saranno accettate. Tuttavia, la richiesta di  $P_3$  sarà accettata.

L'Algoritmo 8.2 è l'algoritmo del banchiere. Quando un processo effettua una nuova richiesta, viene inserita nella matrice  $Risorse\_richieste$ , che memorizza le richieste in attesa di tutti i processi e viene invocato l'algoritmo con l'identificativo del processo richiedente. Quando un processo rilascia alcune risorse allocate per sé oppure termina la sua attività, viene invocato l'algoritmo per ogni processo, la cui richiesta è in attesa. L'algoritmo può essere descritto come segue: dopo alcune inizializzazioni al Passo 1, l'algoritmo simula l'accettazione della richiesta al Passo 2 calcolando lo stato di allocazione proiettato. Il Passo 3 controlla se lo stato di allocazione proiettato è fattibile, ossia se esistono sufficienti risorse libere per permettere l'accettazione della richiesta.

Per controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro, si controlla se il massimo numero di risorse di cui necessita ogni processo attivo, cioè ogni processo negli insiemi *Running* o *Blocked*, può essere soddisfatto allocando alcune delle risorse libere. Se esiste un processo di questo tipo, l'algoritmo simula il suo completamento cancellandolo dall'insieme degli *Active* (elenco processi attivi) e

rilasciando le risorse allocate per esso. Questa azione viene ripetuta finché non è possibile cancellare più processi dall'insieme degli *Active*. Se al termine di questo passo l'insieme *Active* è vuoto, lo stato proiettato è uno stato di allocazione sicuro, quindi l'algoritmo cancella la richiesta dalla lista delle richieste in attesa e alloca le risorse richieste. Questa azione non sarà eseguita se lo stato di allocazione proiettato non è né fattibile né sicuro, per cui la richiesta rimane in attesa.

Notare la similitudine del Passo 4 con l'algoritmo di individuazione dei deadlock (Algoritmo 8.1). Di conseguenza, l'algoritmo ha una complessità dell'ordine di  $n^2 \times r$ .

### Algoritmo 8.2 Algoritmo del banchiere

**Input**

|   |  |
|---|--|
| <i>n</i>                                | : numero di processi;  |
| <i>r</i>                                | : numero di classi di risorse;                                   |
| <i>Blocked</i>                          | : <b>insieme di</b> processi;                                    |
| <i>Running</i>                          | : <b>insieme di</b> processi;                                    |
| <i>P<sub>processo_richiedente</sub></i> | : Processo che effettua la nuova richiesta di risorsa;           |
| <i>Risorse_maxime</i>                   | : <b>array</b> [1.. <i>n</i> , 1.. <i>r</i> ] <b>di</b> integer; |
| <i>Risorse_allocate</i>                 | : <b>array</b> [1.. <i>n</i> , 1.. <i>r</i> ] <b>di</b> integer; |
| <i>Risorse_richieste</i>                | : <b>array</b> [1.. <i>n</i> , 1.. <i>r</i> ] <b>di</b> integer; |
| <i>Risorse_totali_allocate</i>          | : <b>array</b> [1.. <i>r</i> ] <b>di</b> integer;                |
| <i>Risorse_totali</i>                   | : <b>array</b> [1.. <i>r</i> ] <b>di</b> integer;                |

**Strutture dati**

|   |  |
|---|--|
| <i>Active</i>                           | : <b>insieme di</b> processi;                                    |
| <i>fattibile</i>                        | : boolean;   |
| <i>Nuova_richiesta</i>                  | : <b>array</b> [1.. <i>r</i> ] <b>di</b> integer;                |
| <i>Allocazione_simulata</i>             | : <b>array</b> [1.. <i>n</i> , 1.. <i>r</i> ] <b>di</b> integer; |
| <i>Risorse_totali_allocate_simulate</i> | : <b>array</b> [1.. <i>r</i> ] <b>di</b> integer;                |

1.  $Active = Running \cup Blocked;$   
**for** *k* = 1..*r*  
 $Nuova\_richiesta[k] = Risorse\_richieste[processo\_richiedente, k];$
2.  $Allocazione\_simulata := Risorse\_allocate;$   
**for** *k* = 1..*r* /\* Calcolare lo stato di allocazione proiettato\*/  
 $Allocazione\_simulata[processo\_richiedente, k] :=$   
 $Allocazione\_simulata[processo\_richiedente, k] + Nuova\_richiesta[k];$   
 $Risorse\_totali\_allocate\_simulate[k] := Risorse\_totali\_allocate[k] + Nuova\_richiesta[k];$
3.  $fattibile = true;$   
**for** *k* = 1..*r* /\* Controllare se lo stato di allocazione proiettato è fattibile \*/  
**if** *Risorse\_totali*[*k*] < *Risorse\_totali\_allocate\_simulate*[*k*] **then** *fattibile* = false;
4. **if** *fattibile* = true  
**then** /\* Controllare se lo stato di allocazione proiettato è uno stato di allocazione sicuro \*/  
**while** l'insieme *Active* contiene un processo *P<sub>j</sub>* tale che  
Per ogni *k*,  $Risorse\_totali[k] - Risorse\_totali\_allocate\_simulate[k] \geq Risorse\_maxime[l, k] - Allocazione\_simulata[l, k]$   
Rimuovi *P<sub>j</sub>* da *Active*;  
**for** *k* = 1..*r*  
 $Risorse\_totali\_allocate\_simulate[k] := Risorse\_totali\_allocate\_simulate[k] - Allocazione\_simulata[l, k];$
5. **if** l'insieme *Active* è vuoto  
**then** /\* Lo stato di allocazione proiettato è uno stato di allocazione sicuro \*/  
**for** *k* = 1..*r* /\* Cancellare la lista dalle richieste in attesa \*/  
 $Risorse\_richieste[processo\_richiedente, k] := 0;$   
**for** *k* = 1..*r* /\* Accettare la richiesta \*/  
 $Risorse\_allocate[processo\_richiedente, k] :=$   
 $Risorse\_allocate[processo\_richiedente, k] + Nuova\_richiesta[k];$   
 $Risorse\_totali\_allocate[k] := Risorse\_totali\_allocate[k] + Nuova\_richiesta[k];$

### Esempio 8.11 - Algoritmo del banchiere per classi di risorsa multiple

La Figura 8.8 illustra il funzionamento dell'algoritmo del banchiere in un sistema che contiene quattro processi *P<sub>1</sub>*, *P<sub>2</sub>*, *P<sub>3</sub>* e *P<sub>4</sub>*. Quattro classi di risorsa che contengono 6, 4, 8 e 5 unità di risorsa, di cui 5, 3, 5 e 4 unità di risorsa sono attualmente allocate. Il processo *P<sub>2</sub>* ha effettuato una richiesta (0, 1, 1, 0), che sta per essere elaborata.

L'algoritmo simula l'accettazione di questa richiesta nel Passo 2, e controlla se lo stato di allocazione proiettato è sicuro nel Passo 4. La Figura 8.8(b) mostra le strutture dati dell'algoritmo del banchiere all'inizio di questo controllo. In questo stato, sono

disponibili 1, 0, 2 e 1 unità di risorsa, quindi solo il processo  $P_1$  può terminare. Di conseguenza, l'algoritmo simula il suo completamento. La [Figura 8.8\(c\)](#) mostra le strutture dati dopo che  $P_1$  ha terminato. Le risorse allocate per  $P_1$  sono state rilasciate e quindi vengono eliminate da *Risorse\_allocate\_simulate* e  $P_1$  è cancellato dall'insieme *Active*. Il processo  $P_4$  ha bisogno di 0, 1, 3 e 4 unità di risorsa per soddisfare la sua richiesta massima di risorse necessarie, quindi, ora può avere queste risorse allocate, e può terminare. I processi rimanenti possono terminare nell'ordine  $P_2, P_3$ . Di conseguenza, la richiesta effettuata dal processo  $P_2$  viene accettata.

**(a)** Stato dopo il passo 1

|                 | $R_1$ | $R_2$ | $R_3$ | $R_4$            |  | $R_1$ | $R_2$ | $R_3$             | $R_4$ |   | $R_1$ | $R_2$                    | $R_3$ | $R_4$ |   |  |
|-----------------|-------|-------|-------|------------------|--|-------|-------|-------------------|-------|---|-------|--------------------------|-------|-------|---|--|
| $P_1$           | 2     | 1     | 2     | 1                |  | $P_1$ | 1     | 1                 | 1     | 1 | $P_1$ | 0                        | 0     | 0     | 0 |  |
| $P_2$           | 2     | 4     | 3     | 2                |  | $P_2$ | 2     | 0                 | 1     | 0 | $P_2$ | 0                        | 1     | 1     | 0 |  |
| $P_3$           | 5     | 4     | 2     | 2                |  | $P_3$ | 2     | 0                 | 2     | 2 | $P_3$ | 0                        | 0     | 0     | 0 |  |
| $P_4$           | 0     | 3     | 4     | 1                |  | $P_4$ | 0     | 2                 | 1     | 1 | $P_4$ | 0                        | 0     | 0     | 0 |  |
| Risorse massime |       |       |       | Risorse allocate |  |       |       | Risorse richieste |       |   |       | Totali allocate          |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Totali esistenti         |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Attivi                   |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | { $P_1, P_2, P_3, P_4$ } |       |       |   |  |

**(b)** Stato dopo il ciclo while del passo 4

|                 | $R_1$ | $R_2$ | $R_3$ | $R_4$            |  | $R_1$ | $R_2$ | $R_3$             | $R_4$ |   | $R_1$ | $R_2$                    | $R_3$ | $R_4$ |   |  |
|-----------------|-------|-------|-------|------------------|--|-------|-------|-------------------|-------|---|-------|--------------------------|-------|-------|---|--|
| $P_1$           | 2     | 1     | 2     | 1                |  | $P_1$ | 1     | 1                 | 1     | 1 | $P_1$ | 0                        | 0     | 0     | 0 |  |
| $P_2$           | 2     | 4     | 3     | 2                |  | $P_2$ | 2     | 1                 | 2     | 0 | $P_2$ | 0                        | 1     | 1     | 0 |  |
| $P_3$           | 5     | 4     | 2     | 2                |  | $P_3$ | 2     | 0                 | 2     | 2 | $P_3$ | 0                        | 0     | 0     | 0 |  |
| $P_4$           | 0     | 3     | 4     | 1                |  | $P_4$ | 0     | 2                 | 1     | 1 | $P_4$ | 0                        | 0     | 0     | 0 |  |
| Risorse massime |       |       |       | Risorse allocate |  |       |       | Risorse richieste |       |   |       | Totali allocate simulate |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Totali esistenti         |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Attivi                   |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | { $P_1, P_2, P_3, P_4$ } |       |       |   |  |

**(c)** Stato dopo aver simulato il completamento del processo  $P_1$

|                 | $R_1$ | $R_2$ | $R_3$ | $R_4$            |  | $R_1$ | $R_2$ | $R_3$             | $R_4$ |   | $R_1$ | $R_2$                    | $R_3$ | $R_4$ |   |  |
|-----------------|-------|-------|-------|------------------|--|-------|-------|-------------------|-------|---|-------|--------------------------|-------|-------|---|--|
| $P_1$           | 2     | 1     | 2     | 1                |  | $P_1$ | 1     | 1                 | 1     | 1 | $P_1$ | 0                        | 0     | 0     | 0 |  |
| $P_2$           | 2     | 4     | 3     | 2                |  | $P_2$ | 2     | 1                 | 2     | 0 | $P_2$ | 0                        | 1     | 1     | 0 |  |
| $P_3$           | 5     | 4     | 2     | 2                |  | $P_3$ | 2     | 0                 | 2     | 2 | $P_3$ | 0                        | 0     | 0     | 0 |  |
| $P_4$           | 0     | 3     | 4     | 1                |  | $P_4$ | 0     | 2                 | 1     | 1 | $P_4$ | 0                        | 0     | 0     | 0 |  |
| Risorse massime |       |       |       | Risorse allocate |  |       |       | Risorse richieste |       |   |       | Totali allocate simulate |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Totali esistenti         |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Attivi                   |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | { $P_2, P_3, P_4$ }      |       |       |   |  |

**(d)** Stato dopo aver simulato il completamento del processo  $P_4$

|                 | $R_1$ | $R_2$ | $R_3$ | $R_4$            |  | $R_1$ | $R_2$ | $R_3$             | $R_4$ |   | $R_1$ | $R_2$                    | $R_3$ | $R_4$ |   |  |
|-----------------|-------|-------|-------|------------------|--|-------|-------|-------------------|-------|---|-------|--------------------------|-------|-------|---|--|
| $P_1$           | 2     | 1     | 2     | 1                |  | $P_1$ | 1     | 1                 | 1     | 1 | $P_1$ | 0                        | 0     | 0     | 0 |  |
| $P_2$           | 2     | 4     | 3     | 2                |  | $P_2$ | 2     | 1                 | 2     | 0 | $P_2$ | 0                        | 1     | 1     | 0 |  |
| $P_3$           | 5     | 4     | 2     | 2                |  | $P_3$ | 2     | 0                 | 2     | 2 | $P_3$ | 0                        | 0     | 0     | 0 |  |
| $P_4$           | 0     | 3     | 4     | 1                |  | $P_4$ | 0     | 2                 | 1     | 1 | $P_4$ | 0                        | 0     | 0     | 0 |  |
| Risorse massime |       |       |       | Risorse allocate |  |       |       | Risorse richieste |       |   |       | Totali allocate simulate |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Totali esistenti         |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Attivi                   |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | { $P_2, P_3$ }           |       |       |   |  |

**(e)** Stato dopo aver simulato il completamento del processo  $P_2$

|                 | $R_1$ | $R_2$ | $R_3$ | $R_4$            |  | $R_1$ | $R_2$ | $R_3$             | $R_4$ |   | $R_1$ | $R_2$                    | $R_3$ | $R_4$ |   |  |
|-----------------|-------|-------|-------|------------------|--|-------|-------|-------------------|-------|---|-------|--------------------------|-------|-------|---|--|
| $P_1$           | 2     | 1     | 2     | 1                |  | $P_1$ | 1     | 1                 | 1     | 1 | $P_1$ | 0                        | 0     | 0     | 0 |  |
| $P_2$           | 2     | 4     | 3     | 2                |  | $P_2$ | 2     | 1                 | 2     | 0 | $P_2$ | 0                        | 1     | 1     | 0 |  |
| $P_3$           | 5     | 4     | 2     | 2                |  | $P_3$ | 2     | 0                 | 2     | 2 | $P_3$ | 0                        | 0     | 0     | 0 |  |
| $P_4$           | 0     | 3     | 4     | 1                |  | $P_4$ | 0     | 2                 | 1     | 1 | $P_4$ | 0                        | 0     | 0     | 0 |  |
| Risorse massime |       |       |       | Risorse allocate |  |       |       | Risorse richieste |       |   |       | Totali allocate simulate |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Totali esistenti         |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | Attivi                   |       |       |   |  |
|                 |       |       |       |                  |  |       |       |                   |       |   |       | { $P_3$ }                |       |       |   |  |

**Figura 8.8** Funzionamento dell'algoritmo del banchiere per l'Esempio 8.11.

## 8.7 Algoritmo del banchiere con allocazioni variabili

Si supponga che in un sistema multiprocessore a 3 processori siano presenti cinque processi  $P_0, P_1$  (allocati sul processore 0),  $P_2$  (allocato sul processore 1) e  $P_3, P_4$  (allocati sul processore 2) e un insieme di risorse di quattro tipi diversi A, B, C, D, e di trovarsi nella seguente configurazione:

|       | A | B       | C       | D |
|-------|---|---------|---------|---|
| $P_0$ | 4 | $X - 1$ | 3       | 2 |
| $P_1$ | 8 | 0       | $Y - 2$ | 2 |
| $P_2$ | 4 | 0       | 0       | 0 |
| $P_3$ | 0 | 0       | 3       | 2 |
| $P_4$ | 2 | 1       | $Z + 1$ | 4 |

Risorse allocate

|       | A  | B | C | D |
|-------|----|---|---|---|
| $P_0$ | 6  | 4 | 5 | 6 |
| $P_1$ | 10 | 7 | 6 | 8 |
| $P_2$ | 6  | 2 | 0 | 8 |
| $P_3$ | 0  | 3 | 4 | 2 |
| $P_4$ | 9  | 1 | 6 | 9 |

Risorse massime

|  | A | B | C  | D |
|--|---|---|----|---|
|  | 2 | 2 | 10 | 4 |

Risorse disponibili

Determinare gli intervalli dei valori interi di X, Y e Z per i quali:

- il sistema si trova in uno stato sicuro, elencando eventuali sequenze sicure;
- la richiesta attuale di  $P_2$  (2, 0, 0, 2) può essere soddisfatta.

### 8.7.1 Soluzione

Bisogna determinare una sequenza sicura in funzione dei parametri X, Y e Z:

- l'unico processo che può essere soddisfatto è  $P_0$  perché:

$$\begin{cases} 5 - x \geq 0 \\ 5 - x \leq 2 \end{cases} \Rightarrow \begin{cases} x \leq 5 \\ x \geq 3 \end{cases} \Rightarrow 3 \leq x \leq 5$$

quindi:

$$P_0[6, x + 1, 13, 6]$$

- solo la richiesta di  $P_3$  può essere soddisfatta:

$$P_3[6, x + 1, 16, 8]$$

- si può soddisfare la richiesta di  $P_2$ :

$$P_2[10, x + 1, 16, 8]$$

- si può soddisfare la richiesta di  $P_4$  se:

$$\begin{cases} 5 - z \geq 0 \\ 5 - z \leq 16 \end{cases} \Rightarrow \begin{cases} z \leq 5 \\ z \geq 5 - 16 = -11 \end{cases} \Rightarrow -11 \leq z \leq 5$$

quindi:

$$P_4[12, x + 2, 17 + z, 12]$$

- resta solo  $P_1$  la cui richiesta può essere soddisfatta solo se:

$$\begin{cases} 8 - y \geq 0 \\ 8 - y \leq 17 + z \end{cases} \Rightarrow \begin{cases} y \leq 8 \\ y \geq -5 - z \end{cases} \Rightarrow -14 \leq y \leq 8$$

La sequenza sicura è dunque:

$$< P_0, P_3, P_2, P_4, P_1 > \text{ con } 3 \leq x \leq 5, -11 \leq z \leq 5, -14 \leq y \leq 8$$

La richiesta di  $P_2[2, 0, 0, 2]$  non può essere immediatamente soddisfatta in quanto porterebbe il sistema in uno stato non sicuro. Può essere soddisfatta dopo che  $P_0$  e  $P_3$  hanno rilasciato le loro risorse. Nota: il fatto che il sistema sia multiprocessore non influenza l'esecuzione dei processi in quanto non ci sono processi che possono ottenere le risorse contemporaneamente.

### 8.8 Caratterizzazione dei deadlock delle risorse tramite i

## modelli a grafi

La *caratterizzazione del deadlock* è una dichiarazione delle caratteristiche essenziali di un deadlock. Nel Paragrafo 8.4 abbiamo presentato un algoritmo di individuazione dei deadlock che adotta il modello basato su matrice dello stato di allocazione di un sistema. Seguendo tale algoritmo, possiamo caratterizzare un deadlock come una situazione in cui non è possibile costruire una sequenza di completamento di processo, rilascio delle risorse ed eventi di allocazione delle risorse in base ai quali tutti i processi in un sistema possono terminare.

In questo paragrafo, discutiamo la caratterizzazione dei deadlock usando i modelli basati sul grafo dello stato di allocazione ed elementi della teoria dei grafi. Come visto nel Paragrafo 8.2.1, una relazione di attesa circolare tra processi è una condizione *necessaria* per un deadlock. Essa si manifesta in un ciclo in un RRAG o WFG. Un ciclo è una condizione *sufficiente* per un deadlock in alcuni sistemi (Esempio 8.3), ma non in altri (Esempio 8.4). Questa differenza è causata dalla natura delle classi di risorsa e delle richieste di risorsa nel sistema. Di conseguenza, classifichiamo i sistemi in base alle classi di risorsa e alle richieste di risorsa usate nei sistemi e forniamo caratterizzazioni separate di deadlock per le diverse classi di sistemi. Successivamente, esaminiamo la caratterizzazione di deadlock applicabile a tutti i sistemi. Utilizziamo nel seguito un RRAG per raffigurare lo stato di allocazione di un sistema.

### **I modelli per classi di risorse e per richieste di risorse**

Una classe di risorse  $R_i$  può contenere un'unica istanza della sua risorsa, oppure può contenere molte istanze. Ci stiamo riferendo, rispettivamente, ai due tipi di classi: classi di risorse a *singola-istanza* (SI) e classi di risorse a *istanza-multipla* (MI). Definiamo due tipi di richieste di risorsa. In una *richiesta-singola* (SR), un processo può richiedere una unità di una sola classe di risorsa, mentre una *richiesta-multipla* (MR) si ha quando un processo può richiedere una unità per ognuna delle diverse classi di risorse. Il kernel non alloca mai parzialmente una richiesta multipla; ossia, o alloca *tutte* le risorse richieste in una richiesta multipla o *nessuna* di esse. Nel secondo caso, il processo che fa la richiesta è bloccato finché possono essere allocate tutte le risorse.

Usando i modelli per le classi di risorse e per le richieste di risorse, possiamo definire quattro tipi di sistemi come mostrato in [Figura 8.9](#). Indichiamo questi sistemi combinando il nome del modello della classe di risorsa e il modello della richiesta di risorsa usato. Per esempio, il sistema SISR è quello che contiene le classi di risorsa SI e le richieste SR.

|                               |                                 | Modelli di richiesta delle risorse         |   |
|-------------------------------|---------------------------------|--|---|
|                               |                                 | Modello a singola richiesta (SR)           | Modello a richiesta multipla (MR)           |
| Modelli di istanza di risorse | Modello a istanza multipla (MI) | Istanza multipla, richiesta singola (MISR) | Istanza multipla, richiesta multipla (MIMR) |
|                               | Modello a istanza singola (SI)  | Istanza singola, richiesta singola (SISR)  | Istanza singola, richiesta multipla (SIMR)  |

**Figura 8.9** Classificazione dei sistemi in base ai modelli della classe di risorsa e della richiesta di risorsa.

### **8.8.1 Sistemi Singola-Istanza, Singola-Richiesta (SISR)**

In un sistema SISR, ogni classe di risorsa contiene una sola istanza di risorsa e ogni richiesta è una richiesta singola. Come discusso nel Paragrafo 8.2.2, l'esistenza di un ciclo in un RRAG implica una relazione di attesa reciproca per un insieme di processi. Poiché ogni classe di risorsa contiene una sola unità di risorsa, ogni processo bloccato  $P_i$  nel ciclo attende che esattamente un altro processo, per esempio  $P_k$ , rilasci la risorsa richiesta. Di conseguenza, un ciclo che comprende il processo  $P_i$  coinvolge anche il processo  $P_k$ . Ciò soddisfa la Condizione 8.2 per tutti i processi nel ciclo. Un ciclo è,

quindi, una condizione necessaria e sufficiente per concludere che esiste un deadlock nel sistema.

### 8.8.2 Sistemi Istanza-Multipla, Singola-Richiesta (MISR)

Un ciclo non è una condizione sufficiente per un deadlock nei sistemi MISR perché le classi di risorsa possono contenere diverse unità di risorsa. Si analizzi il sistema dell'Esempio 8.4, che illustrava questa proprietà, per comprendere quali condizioni debbano sussistere affinché esista un deadlock in un sistema MISR. Il RRAG del sistema contiene un ciclo comprendente i processi  $P_i$  e  $P_j$ , con  $P_j$  che richiede un'unità nastro e  $P_i$  che possiede un'unità nastro (Figura 8.3). Anche il processo  $P_k$ , che non appartiene al ciclo, possiede un'unità nastro, così la relazione di attesa reciproca tra  $P_i$  e  $P_j$ , cessa di esistere quando  $P_k$  rilascia l'unità nastro. Il processo  $P_i$  sarebbe stato in deadlock solo se i processi  $P_j$  e  $P_k$  fossero andati incontro entrambi ad attese indefinite. Quindi, affinché un processo sia in deadlock, è essenziale che tutti i processi che posseggono unità di una risorsa da esso richiesta, siano anch'essi in deadlock. Usiamo i concetti della teoria dei grafi per introdurre questo aspetto nella caratterizzazione dei deadlock nei sistemi MISR.

Un grafo  $G$  è una coppia ordinata  $G \equiv (N, E)$  dove  $N \equiv$  un insieme di *nodi* ed  $E$  è un insieme di *archi*. Un grafo  $G' \equiv (N', E')$  è un *sottografo* di un grafo  $G \equiv (N, E)$  se  $N' \subseteq N$  e  $E' \subseteq E$ , ossia se tutti i nodi e gli archi contenuti in  $G'$  sono contenuti anche in  $G$ .  $G'$  è un *sottografo* non banale di  $G$  se  $E' \neq \emptyset$ , ossia se contiene almeno un arco. Definiamo ora un *taglio* per caratterizzare un deadlock nei sistemi MISR.

**Definizione 8.3 Taglio** Un sottografo non banale  $G' \equiv (N', E')$  di un RRAG in cui ogni nodo  $n_i \in N'$  soddisfa le seguenti condizioni:

1. per ogni arco della forma  $(n_i, n_j)$  in  $E$ :  $(n_i n_j)$  è incluso in  $E'$  e  $n_j$  è incluso in  $N'$ ;
2. se un percorso  $n_i - \dots - n_j$  esiste in  $G'$ , anche il percorso  $n_j - \dots - n_i$  esiste in  $G'$ .

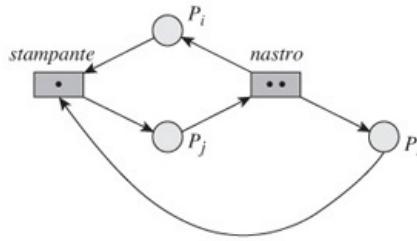
La **Parte 1** della Definizione 8.3 assicura che se un nodo è incluso in un taglio, tutti i suoi archi uscenti, ossia tutti gli archi provenienti da esso, sono inclusi anche essi nel taglio. La **Parte 2** della Definizione 8.3 assicura che ogni arco uscente di ogni nodo è incluso in un ciclo. Questo fatto assicura che ogni processo nel taglio è necessariamente in stato *blocked*. Le Parti 1 e 2 insieme implicano che tutti i processi che possono rilasciare una risorsa necessaria per qualche processo nel taglio sono essi stessi inclusi nel taglio, il che soddisfa la Condizione 8.2. Così possiamo concludere che la presenza di un taglio in un RRAG è una condizione necessaria e sufficiente per l'esistenza di un deadlock in un sistema MISR.

#### Esempio 8.12 - Deadlock in un sistema MISR

Il RRAG di Figura 8.3 raffigura lo stato di allocazione dell'Esempio 8.4 dopo che sono state effettuate le richieste 1-5. Esso non contiene un taglio perché il percorso  $P_i, \dots, P_k$  appartiene al taglio ma il percorso  $P_k, \dots, P_i$  non vi appartiene. Consideriamo ora la situazione dopo che è stata fatta la seguente richiesta:

6.  $P_k$  richiede una stampante.

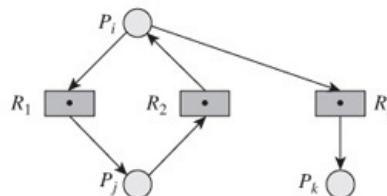
Il processo  $P_k$  si blocca ora sulla sesta richiesta. Il RRAG risultante è mostrato in Figura 8.10. Il RRAG completo è un taglio perché la **Parte 1** della Definizione 8.3 è banalmente soddisfatta, e ogni arco uscente da ogni nodo è coinvolto in un ciclo, il che soddisfa la **Parte 2** della Definizione 8.3. È facile verificare che i processi  $\{P_i, P_j, P_k\}$  sono in un deadlock poiché  $Blocked_P = \{P_i, P_j, P_k\}$ ,  $WF_i = \{P_j\}$ ,  $WF_i = \{P_j, P_k\}$ , e  $WF_k = \{P_j\}$  soddisfano le Condizioni 8.1 e 8.2.



**Figura 8.10** Un taglio nel RRAG di un sistema MISR implica un deadlock.

### 8.8.3 Sistemi Singola-Istanza, Richiesta-Multipla (SIMR)

Ogni classe di risorsa nei sistemi SIMR contiene solo un'unità di risorsa, e quindi ha esattamente un arco uscente nel RRAG. Un processo può effettuare una richiesta multipla, nel qual caso avrà più di un arco uscente. Un processo rimane bloccato se anche una delle sue risorse richieste non è disponibile. Questa condizione è soddisfatta quando il processo è coinvolto in un ciclo, quindi un ciclo è una condizione necessaria e sufficiente per un deadlock in un sistema SIMR. Questa proprietà è illustrata dal sistema di [Figura 8.11](#). Il nodo del processo  $P_i$  ha un arco uscente  $(P_i, R_1)$  che è una parte di un ciclo, e un arco uscente  $(P_i, R_3)$  che non è parte di alcun ciclo.



**Figura 8.11** Un ciclo è una condizione necessaria e sufficiente per un deadlock in un sistema SIMR.

Il processo  $P_i$  rimane bloccato finché non ottiene un'unità di risorsa di  $R_1$ . Poiché l'arco uscente  $(P_i, R_1)$  è coinvolto in un ciclo,  $P_i$  va incontro a un'attesa indefinita. Anche  $P_j$  va incontro a un'attesa indefinita. Di conseguenza  $\{P_i, P_j\}$  sono coinvolti in un deadlock.

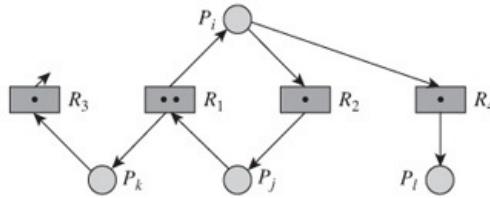
### 8.8.4 Sistemi Istanza-Multipla, Richiesta-Multipla (MIMR)

Nel modello MIMR, le classi di risorsa contengono diverse unità di risorse e i processi possono effettuare richieste multiple, di conseguenza sia i nodi processo che i nodi risorsa di un RRAG possono avere più archi uscenti. Se nessuno dei nodi risorsa, coinvolto in un ciclo nel RRAG, ha più archi uscenti, il ciclo è simile a un ciclo del RRAG di un sistema SIMR, e quindi è una condizione sufficiente per l'esistenza di un deadlock. Comunque, se un nodo risorsa in un ciclo ha più archi uscenti, un ciclo è una condizione necessaria ma non sufficiente per un deadlock. In tali casi ogni arco uscente dal nodo risorsa deve essere coinvolto in un ciclo; questo requisito è simile a quello dei sistemi MISR. L'Esempio 8.13 illustra questo aspetto.

#### Esempio 8.13 - Deadlock in un sistema MIMR

Il RRAG della [Figura 8.12](#) contiene il ciclo  $R_1 - P_i - R_2 - P_j - R_1$ . Il nodo risorsa  $R_1$  contiene un arco uscente  $(R_1, P_k)$  che non è incluso in un ciclo, di conseguenza il processo  $P_k$  può ottenere la risorsa  $R_3$  ed eventualmente rilasciare un'istanza della classe di risorsa  $R_1$ , che potrebbe essere allocata al processo  $P_j$ . Questo interromperà il ciclo nel RRAG, di conseguenza non c'è deadlock nel sistema. Se l'arco di allocazione di  $R_3$  fosse stato  $(R_3, P_i)$ , entrambi gli archi uscenti di  $R_1$  sarebbero stati coinvolti in cicli. La richiesta del processo  $P_j$  per  $R_1$  andrebbe ora incontro a un'attesa indefinita, e quindi avremmo una situazione di deadlock. Si noti che l'arco uscente  $(P_i, R_4)$  di  $P_i$  non

è coinvolto in un ciclo; tuttavia, esiste un deadlock perché  $P_i$  ha effettuato una richiesta multipla e la sua richiesta per la classe di risorsa  $R_2$  causa un'attesa indefinita.



**Figura 8.12** RRAG per un sistema MIMR.

Dalla discussione precedente e dall'Esempio 8.13 è chiaro che dobbiamo distinguere tra nodi processo e nodi risorsa nel RRAG di un sistema MIMR: tutti gli archi uscenti di un nodo risorsa devono essere coinvolti in cicli perché si verifichi un deadlock, mentre un nodo processo ha bisogno di avere solo un arco uscente coinvolto in un ciclo. Definiamo un *taglio di risorse* per incorporare questo requisito, dove un taglio di risorse differisce da un taglio solo nel fatto che la *Parte 1* della Definizione 8.4 si applica soltanto ai nodi risorsa.

**Definizione 8.4 Taglio di risorse** Un sottografo non banale  $G' = (N', E')$  di un RRAG in cui ogni nodo  $n_i \in N'$  soddisfa le seguenti condizioni:

1. se  $n_i$  è un nodo risorsa, per ogni arco della forma  $(n_i, n_j)$  in  $E'$ :  $(n_i, n_j)$  è incluso in  $E'$  e  $n_j$  è incluso in  $N'$ ;
2. se un percorso  $n_i - \dots - n_j$  esiste in  $G'$ , anche il percorso  $n_j - \dots - n_i$  esiste in  $G'$ .

#### Esempio 8.14 - Taglio di risorse

I nodi  $P_i, P_j, P_k, R_2, R_2$  ed  $R_3$  della Figura 8.12 dovrebbero essere coinvolti in un taglio di risorse se l'arco di allocazione della classe di risorsa  $R_3$  fosse  $(R_3, P_i)$ . Notare che l'arco uscente  $(P_i, R_4)$  del processo  $P_i$  non è incluso nel taglio di risorse.

Chiaramente, un taglio di risorse è una condizione necessaria e sufficiente per l'esistenza di un deadlock in un sistema MIMR. Infatti, stiamo affermando senza dimostrazione che un taglio di risorse è una condizione necessaria e sufficiente per l'esistenza di un deadlock in tutte le classi di sistemi discussi in questo paragrafo (Problema 8.17).

### 8.8.5 Processi in deadlock

Identifichiamo con  $D$  l'insieme dei processi in deadlock, che contiene i processi rappresentati dai nodi processo nei tagli di risorse. Inoltre, contiene altri processi che vanno incontro ad attese indefinite. Usiamo la seguente notazione per identificare tutti i processi in  $D$ .

|        |   |
|--------|---|
| $RR_i$ | l'insieme delle classi di risorsa richieste dal processo $P_i$ .  |
| $HS_k$ | l'insieme <i>holder</i> (o possessore) della classe di risorsa $R_k$ , ossia l'insieme dei processi ai quali sono allocate le unità della classe di risorsa $R_k$ . |
| $KS$   | l'insieme dei nodi processo nel taglio delle risorse (detto <i>insieme taglio</i> del RRAG).  |
| $AS$   | un <i>insieme ausiliario</i> di nodi processo nel RRAG che vanno incontro ad attese indefinite. Questi nodi non sono inclusi in un taglio delle risorse.            |

$KS$  è l'insieme dei nodi processo inclusi nel taglio delle risorse. Ora un processo  $P_i \notin KS$  va incontro a un'attesa indefinita se tutti i possessori di una classe di risorsa  $R_k$  da esso

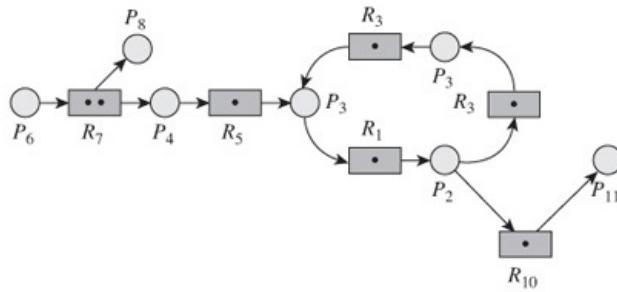
richiesta sono inclusi in  $KS$ . Le classi di risorsa i cui possessori sono inclusi in  $\{P_i\} \cup KS$  causano in modo analogo attese indefinite per i loro richiedenti. Perciò possiamo identificare  $D$ , l'insieme dei processi in deadlock, come segue:

$$AS = \{P_i \mid RR_i \text{ contiene } R_k \text{ tale che } HS_k \subseteq (KS \cup AS)\} \quad (8.4)$$

$$D = KS \cup AS \quad (8.5)$$

### Esempio 8.15 - Processi in deadlock

La Figura 8.13 mostra un RRAG di un sistema MIMR. Il ciclo  $P_1 - R_1 - P_2 - R_2 - P_3 - R_3 - P_1$  nel RRAG forma un taglio delle risorse perché nessuno dei nodi  $R_1$ ,  $R_2$ ,  $R_3$  hanno un arco uscente che esce dal ciclo. Di conseguenza, esiste un deadlock nel sistema. Identifichiamo i processi in  $D$  come segue:



**Figura 8.13** Processi in deadlock.

$$KS = \{P_1, P_2, P_3\}$$

$$AS = \{P_4\} \text{ poiché } RR_4 = \{R_5\}, HS_5 = \{P_1\} \text{ e } \{P_1\} \subseteq \{P_1, P_2, P_3\}$$

$$D = KS \cup AS = \{P_1, P_2, P_3, P_4\}.$$

Il processo  $P_6$  non è incluso in  $AS$  poiché  $RR_6 = \{R_7\}$ ,  $HS_7 = \{P_4, P_8\}$  e  $HS_7 \not\subseteq (KS \cup AS)$ .

## 8.9 Casi di studio

Un sistema operativo gestisce numerose risorse di differenti tipi, quali risorse hardware come la memoria e i dispositivi di I/O, risorse software come i file contenenti programmi o dati e messaggi tra processi e risorse del kernel come strutture dati e blocchi di controllo utilizzati dal kernel. L'overhead per l'individuazione e la risoluzione dei deadlock e delle tecniche per evitare i deadlock rende tali politiche di gestione dei deadlock poco interessanti nella pratica. Di conseguenza, un SO può utilizzare l'approccio di prevenzione dei deadlock, creando una situazione in cui non sono necessarie azioni esplicite di gestione dei deadlock, oppure semplicemente non si preoccupa della possibilità di deadlock. Inoltre, poiché la prevenzione dei deadlock vincola l'ordine in cui i processi richiedono le loro risorse, i sistemi operativi tendono a gestire i deadlock separatamente per ogni tipo di risorsa come la memoria, i dispositivi di I/O, i file e le risorse del kernel. Discutiamo nel seguito questi approcci.

### Memoria

La memoria è una risorsa prelazionabile, quindi il suo uso da parte dei processi non può causare un deadlock. La gestione esplicita dei deadlock non è perciò necessaria. La memoria allocata a un processo viene rilasciata riversando il processo sul disco se la memoria è necessaria a un altro processo.

### Dispositivi di I/O

Tra le politiche di prevenzione dei deadlock, la politica di allocazione globale richiede che tutti i processi facciano una richiesta multipla per *tutte* le loro richieste di risorse. Questa politica genera il minimo overhead di CPU, ma ha lo svantaggio menzionato nel

Paragrafo 8.5.1; ovvero comporta uno scarso utilizzo dei dispositivi di I/O che vengono allocati molto prima che un processo ne abbia effettivamente bisogno. Il ranking delle risorse, dall'altra parte, non è una politica pratica per controllare l'uso dei dispositivi di I/O perché qualsiasi assegnazione dei rank di risorse causa inconvenienti a qualche gruppo di utenti. Questo problema è collegato al fatto che i dispositivi di I/O generalmente non sono prelazionabili. I sistemi operativi superano questo problema creando dispositivi virtuali. Per esempio, il sistema crea una stampante virtuale usando un'area del disco per memorizzare il file che deve essere stampato. La stampa effettiva sarà eseguita appena una stampante diventa disponibile. Poiché i dispositivi virtuali vengono creati su richiesta, non è necessario preallocare come nella politica di allocazione globale senza che il sistema vada incontro a una riduzione di spazio su disco.

### **File e messaggi tra processi**

Un file è una risorsa creata dall'utente e un SO contiene un elevato numero di file. Le politiche di prevenzione dei deadlock, come il ranking delle risorse, potrebbero generare un elevato overhead e inconvenienti agli utenti. Di conseguenza, i sistemi operativi non estendono ai file le azioni di gestione dei deadlock; i processi stessi, che accedono a uno stesso insieme di file, dovranno prevedere soluzioni per evitare i deadlock. Per analoghe ragioni, i sistemi operativi non gestiscono i deadlock causati da messaggi tra processi.

### **Blocchi di controllo**

Il kernel alloca i blocchi di controllo come i process control block (PCB) e gli event control block (ECB) per i processi in un determinato ordine. In particolare, un PCB è allocato quando un processo viene creato e un ECB è allocato quando un processo è bloccato su di un evento. Di conseguenza, una soluzione può essere il ranking delle risorse. Se si vuole una politica più semplice, tutti i control block per un job o un processo possono essere allocati insieme sin dall'inizio.

## **8.9.1 Gestione dei deadlock in Unix**

La maggior parte dei sistemi operativi semplicemente non considera la possibilità di deadlock che coinvolgono i processi utente e Unix ne è un esempio. Tuttavia, Unix gestisce i deadlock dovuti alla condivisione delle strutture dati del kernel tra i processi utente. Nel Paragrafo 5.4.1 abbiamo visto che un processo Unix, in esecuzione da parte della CPU, esegue codice kernel quando si verifica un'interrupt o una chiamata di sistema, per cui i processi utente potrebbero eseguire concorrentemente codice kernel. Il kernel utilizza l'approccio del ranking delle risorse per la prevenzione dei deadlock (Paragrafo 8.5.2) richiedendo ai processi di impostare dei lock sulle strutture dati del kernel in ordine standard; tuttavia, ci sono eccezioni a questa regola, e quindi si potrebbero verificare dei deadlock. Presentiamo delle semplificazioni delle due soluzioni usate per prevenire i deadlock.

Il kernel Unix utilizza un buffer cache (Paragrafo 14.13.1) per velocizzare gli accessi ai blocchi del disco usati più frequentemente. Esso consiste di un pool di buffer in memoria e una struttura dati hash per controllare se uno specifico blocco del disco è presente in un buffer. Per facilitare il riutilizzo dei buffer, la lista di buffer è mantenuta in ordine least recently used (LRU), cioè il primo buffer nella lista è il buffer usato meno di recente e l'ultimo buffer è il buffer usato più di recente. Il normale ordine di accesso a un blocco del disco è quello di usare la struttura dati hash per localizzare un blocco se esiste in un buffer, impostare un lock sul buffer contenente il blocco del disco e poi porre un lock sulla lista dei buffer per aggiornare lo stato LRU del buffer. Comunque, se un processo richiede soltanto un buffer per caricare un nuovo blocco del disco, accederà direttamente alla lista dei buffer e prenderà il primo buffer non in uso in quel momento. Per eseguire questa azione, il processo mette un lock sulla lista. Successivamente, prova a impostare il lock sul primo buffer nella lista. C'è possibilità di deadlock perché quest'ordine di locking della lista e del buffer è diverso dall'ordine standard di impostazione di questi lock.

Unix utilizza un approccio innovativo per evitare tali deadlock. Prevede una particolare operazione che tenta di impostare un lock e restituisce un codice di errore se il lock è già impostato. Il processo che cerca un buffer libero sfrutta questa operazione per controllare se un buffer è libero. Se viene restituito un codice di errore, il processo tenterà semplicemente di impostare il lock al buffer successivo e, così via, finché trova un buffer che può usare. Questo approccio evita i deadlock evitando attese circolari.

Un'altra situazione in cui i lock non possono essere impostati nell'ordine standard è la funzione del file system che consente di creare un link (Paragrafo 13.4.2). Un comando per la creazione di un link fornisce il path name per un file e la directory che contiene il link al file. Questo comando può essere implementato bloccando le directory che contengono il file e il link. Tuttavia, non può essere definito un ordine standard per l'impostazione dei lock di queste directory. Di conseguenza, due processi, che tentano di bloccare concorrentemente le stesse directory, possono andare in deadlock. Per evitare tali deadlock, la funzione del file system non tenta di acquisire entrambi i lock allo stesso tempo. Prima blocca una directory, la aggiorna nella maniera opportuna e rilascia il lock. Poi blocca l'altra directory e l'aggiorna. Così, richiede solo un lock alla volta. Questo approccio previene i deadlock perché questi processi non soddisfano la condizione hold-and-wait.

## 8.9.2 Gestione dei deadlock in Windows

Windows Vista prevede una funzionalità chiamata *wait chain traversal* (WCT), che supporta le applicazioni e i debugger nell'individuazione dei deadlock. Una wait chain comincia su un thread ed è analoga a un percorso nel grafo di richiesta e allocazione risorse (RRAG). Così, un thread punta a un object o a un lock per il quale è in attesa, e l'object o il lock punta al thread che ne è in possesso. Un debugger può acquisire informazioni sul motivo di una sospensione o blocco di un'applicazione invocando la funzione *getthreadwaitchain* con l'identificativo del thread per ottenere una catena che parte da quel thread. La funzione restituisce un array che contiene gli identificativi dei thread trovati su una catena di attesa che comincia dal thread designato, e un valore boolean che indica se qualsiasi sottoinsieme di thread trovato sulla wait chain forma un ciclo.

## Riepilogo

Un *deadlock* è una situazione in cui un insieme di processi attende indefinitamente degli eventi (che potrebbero non verificarsi mai), ciascuno dei quali può essere generato solo da altri processi dell'insieme. Un deadlock pregiudica il servizio utente, il throughput e l'efficienza delle risorse. In questo capitolo, abbiamo discusso le tecniche adottate dai SO per la gestione dei deadlock.

Un deadlock di risorsa si verifica quando si verificano quattro condizioni contemporaneamente: le risorse sono non condivisibili e non prelazionabili, un processo possiede delle risorse mentre è in attesa delle risorse in uso da parte di altri processi, chiamata condizione *hold-and-wait*, ed esistono attese circolari tra processi. Un SO può scoprire un deadlock analizzando lo *stato di allocazione* di un sistema, che consiste di informazioni relative alle risorse allocate e alle richieste di risorse sulle quali i processi sono bloccati. Nei sistemi in cui un processo non può richiedere più di una unità di risorsa di risorse, può essere adottato un *modello basato su grafi* dello stato di allocazione. Un *grafo di allocazione e richiesta di risorse* (RRAG) rappresenta l'allocazione delle risorse e le richieste di risorse in attesa nel SO, mentre un *grafo di attesa* (WFG) descrive una relazione di attesa tra processi. In entrambi i modelli, una condizione di attesa circolare si traduce in un percorso circolare nel grafo. Un *modello basato su matrici* rappresenta lo stato di allocazione tramite un insieme di matrici.

Quando un processo termina, esso rilascia le sue risorse e il kernel può allocarle ad altri processi che ne hanno fatto richiesta. Quando viene usato un modello basato su matrici dello stato di allocazione, si può individuare un deadlock verificando se ogni processo, attualmente bloccato su una richiesta di risorse, può ottenere l'allocazione della risorsa richiesta attraverso una sequenza di completamento del processo, rilascio risorse ed eventi di allocazione risorse. Poiché l'individuazione dei deadlock comporta un elevato overhead dovuto a questo controllo, sono stati presentati vari approcci che assicurano l'assenza di deadlock. Nell'approccio *prevenzione dei deadlock*, la politica di allocazione delle risorse impone alcuni vincoli sulle richieste di risorse che facciano in modo che le quattro condizioni per i deadlock non siano soddisfatte contemporaneamente. Nell'approccio per *evitare i deadlock*, chi alloca le risorse conosce il numero massimo di risorse di cui un processo ha bisogno. A ogni richiesta di risorsa, controlla se si può trovare una sequenza di completamento del processo, rilascio risorse ed eventi di allocazione risorse attraverso la quale tutti i processi

possano ottenere il loro massimo necessario e terminare il loro funzionamento. La richiesta di risorsa viene accettata solo se viene soddisfatto tale controllo.

Quando viene usato un modello basato su grafi dello stato di allocazione, i deadlock si possono caratterizzare in termini di percorsi nel grafo. Tuttavia, la caratterizzazione diventa complessa quando una classe di risorsa può contenere molte unità di risorsa.

Per ragioni di convenienza ed efficienza, un SO può usare differenti politiche di gestione dei deadlock per diversi tipi di risorse. Tipicamente, un SO usa approcci di prevenzione dei deadlock per le risorse del kernel, e crea risorse virtuali per evitare i deadlock sui dispositivi di I/O; comunque, non gestisce i deadlock che coinvolgono le risorse utente come file e messaggi tra processi.

## Domande

- 8.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
- Un cido nel grafo di richiesta e allocazione delle risorse (RRAG) è una condizione necessaria e sufficiente per un deadlock se ogni classe di risorsa contiene solo un'unità di risorsa.
  - La risoluzione di deadlock garantisce che i deadlock non si verifichino.
  - La politica di allocazione globale di prevenzione dei deadlock assicura che la condizione di attesa circolare non si verificherà mai nel sistema.
  - La politica ranking delle risorse di prevenzione dei deadlock assicura che la condizione di hold-and-wait non si verificherà mai nel sistema.
  - Se un insieme di processi  $D$  è in deadlock, l'insieme *Blocked* dell'Algoritmo 8.4.1 conterrà alcuni di questi processi quando termina l'esecuzione dell'algoritmo; comunque, *Blocked* può non contenerli tutti.
  - Se un processo  $P_i$  richiede  $r$  unità di una classe di risorsa  $R_j$  e un numero di unità di  $R_j \geq r$  sono libere, allora l'algoritmo del banchiere allocherà definitivamente  $r$  unità per  $P_i$ .
  - L'algoritmo del banchiere non garantisce che i deadlock non si verifichino.
  - Un SO ha una sola classe di risorsa controllata dall'algoritmo del banchiere. 12 unità della risorsa sono state attualmente allocate ai processi, e di tali risorse 2 sono state allocate al processo  $P_i$ . Se  $P_i$  necessita di al più 5 risorse, il sistema contiene minimo 15 unità di risorse.
  - Un SO che usa l'algoritmo del banchiere per più risorse è stato in funzione per qualche tempo con quattro processi. Un nuovo processo arriva nel sistema. Inizialmente non ha risorse allocate. Il nuovo stato di allocazione del sistema è sicuro?
  - Se ogni classe di risorsa in un sistema ha una sola unità di risorsa, ogni ciclo nel RRAG del sistema è anche un taglio di risorse.
- 8.2. Un SO ha  $n$  unità di risorsa di una classe di risorse. Tre processi usano questa classe di risorse e ognuno di essi ha necessitato di al più 3 unità di risorse. Il modo e l'ordine in cui i processi richiedono le unità della classe di risorse non è noto. Qual è il minimo valore di  $n$  affinché il funzionamento del sistema sia privo di deadlock?
- 3, b. 7, c. 6, d. 9
- 8.3. Un SO impiega l'algoritmo del banchiere per controllare l'allocazione di 7 unità nastro. I tre processi  $P_1$ ,  $P_2$  e  $P_3$  hanno necessità di al più 7, 3 e 5, rispettivamente. Quante unità il SO può allocare in modo sicuro al processo  $P_1$ , se l'attuale stato di allocazione è il seguente.
- 2, 1 e 1 unità nastro sono allocate ai processi  $P_1$ ,  $P_2$  e  $P_3$ , rispettivamente.
    - 0, ii. 1, iii. 2, iv. 3
  - 1, 2 e 1 unità nastro sono allocate ai processi  $P_1$ ,  $P_2$  e  $P_3$ , rispettivamente.
    - 0, ii. 1, iii. 2, iv. 3

## Problemi

- 8.1. Giustificare in maniera chiara perché i deadlock possono verificarsi in un sistema

produttore-consumatore a buffer limitato.

- 8.2. Quando viene utilizzato il ranking delle risorse come politica di prevenzione dei deadlock, un processo può richiedere un'unità della classe di risorsa  $R_k$  solo se  $rank_k > rank_i$  per ogni classe di risorsa  $R_i$  le cui risorse sono allocate a esso. Spiegare se si possono verificare deadlock se la condizione diventa  $rank_k \geq rank_i$ .
- 8.3. Un sistema che contiene risorse prelazionabili usa la seguente politica di allocazione delle risorse: quando una risorsa richiesta da un processo  $P_i$  non è disponibile,
- la risorsa è prelazionabile da uno dei processi possessori  $P_j$  se  $P_j$  è più giovane di  $P_i$ . La risorsa è ora allocata a  $P_i$ . Viene allocata nuovamente a  $P_j$  quando  $P_i$  termina. (Un processo è considerato più giovane se è iniziato più tardi.)
  - se la condizione (a) non è soddisfatta,  $P_i$  è bloccato per la risorsa.
- Una risorsa rilasciata è sempre allocata al suo richiedente più vecchio. Mostrare che in questo sistema non si possono verificare deadlock. Mostrare, inoltre, che anche la starvation non si verifica.
- 8.4. Sviluppare un modello basato su matrice per lo stato di allocazione del sistema di [Figura 8.13](#). Applicare l'Algoritmo 8.4.1 per trovare i processi coinvolti nel deadlock.
- 8.5. Modifichiamo il sistema di [Figura 8.13](#) in modo che il processo  $P_6$  abbia effettuato una richiesta multipla per le risorse  $R_7$  e  $R_1$ . Quali sono i processi coinvolti nel deadlock? Il processo  $P_1$  viene interrotto e il processo  $P_3$  effettua una richiesta per la risorsa  $R_5$ . Il sistema ora è in un deadlock?
- 8.6. Un sistema usa la politica di individuazione e risoluzione dei deadlock. Il costo d'interruzione di un processo è considerato di una unità. Discutere come identificare il(i) processo(i) vittima così da minimizzare il costo della risoluzione del deadlock in ognuno dei seguenti sistemi: (a) sistemi SISR, (b) sistemi SIMR, (c) sistemi MISR e (d) sistemi MIMR.
- 8.7. Lo stato di allocazione in cui vengono allocate 6, 1 e 2 unità di risorsa ai processi  $P_1$ ,  $P_2$  e  $P_3$  nel sistema dell'Esempio 8.10 è sicuro? Sarebbe sicuro lo stato di allocazione se venissero allocate 3, 2 e 3 unità di risorsa?
- 8.8. Le seguenti richieste sarebbero accettate nello stato attuale dall'algoritmo del banchiere?

|                       | $R_1 R_2$ | $R_1 R_2$           | $R_1 R_2$ |
|-----------------------|-----------|---------------------|-----------|
| $P_1$                 | 2 5       | 1 3                 | 3 4       |
| $P_2$                 | 3 2       | 2 1                 |           |
| Massimo<br>necessario |           | Risorse<br>allocate |           |
|                       |           | Totale<br>esist.    | 4 5       |

- Processo  $P_2$  richiede (1, 0)
- Processo  $P_2$  richiede (0, 1)
- Processo  $P_2$  richiede (1, 1)
- Processo  $P_1$  richiede (1, 0)
- Processo  $P_1$  richiede (0, 1)

- 8.9. Nel seguente sistema:

|                       | $R_1 R_2 R_3$ | $R_1 R_2 R_3$       | $R_1 R_2 R_3$ |
|-----------------------|---------------|---------------------|---------------|
| $P_1$                 | 3 6 8         | 2 2 3               | 5 4 10        |
| $P_2$                 | 4 3 3         | 2 0 3               |               |
| $P_3$                 | 3 4 4         | 1 2 4               |               |
| Massimo<br>necessario |               | Risorse<br>allocate |               |
|                       |               | Totale<br>esist.    | 7 7 10        |

- È sicuro l'attuale stato di allocazione?
- Sarebbero accettate le seguenti richieste nello stato attuale dall'algoritmo del banchiere?
  - Processo  $P_1$  richiede (1, 1, 0)
  - Processo  $P_3$  richiede (0, 1, 0)
  - Processo  $P_2$  richiede (0, 1, 0)

- 8.10. Tre processi  $P_1$ ,  $P_2$  e  $P_3$  usano una risorsa controllata dall'algoritmo del banchiere.

Esistono nell'attuale stato di allocazione due unità di risorsa non allocate. Quando  $P_1$  e  $P_2$  richiedono un'unità di risorsa ciascuno, rimangono bloccati sulle loro richieste; mentre, quando  $P_3$  richiede due unità di risorsa, la sua richiesta è immediatamente accettata. Spiegare perché.

8.11. Un sistema che usa l'algoritmo del banchiere per l'allocazione delle risorse contiene  $n_1$  e  $n_2$  unità di risorsa delle classi di risorsa  $R_1$  e  $R_2$  e i tre processi  $P_1$ ,  $P_2$  e  $P_3$ . Le risorse non allocate nel sistema sono (1, 1). Vengono fatte le seguenti osservazioni sul funzionamento del sistema.

- a. Se il processo  $P_1$  effettua una richiesta (1, 0) seguita da una richiesta (0, 1), la richiesta (1, 0) sarà concessa mentre la richiesta (0, 1) non lo sarà.
- b. Se, invece delle richieste del punto (a), il processo  $P_1$  effettuasse una richiesta (0, 1), questa sarebbe concessa.

Trovare un possibile insieme di valori per le attuali allocazioni e per le richieste massime di risorse dei processi affinché l'algoritmo del banchiere produca decisioni in linea con le osservazioni precedenti.

8.12. Mostrare che quando viene applicato l'algoritmo del banchiere a un insieme finito di processi, ognuno con un tempo di esecuzione finito, ogni richiesta di risorsa alla fine sarà accettata.

8.13. I processi, in un particolare SO, effettuano richieste multiple. Questo SO usa un algoritmo del banchiere progettato affinché una singola classe di risorsa implementi la tecnica per evitare i deadlock come segue: quando un processo richiede unità di risorse di  $n$  classi di risorsa, la richiesta è vista come un insieme di  $n$  richieste singole; per esempio, una richiesta multipla (2, 1, 3) sarebbe vista come tre richieste singole (2, 0, 0), (0, 1, 0) e (0, 0, 3). La richiesta multipla è accettata solo se ogni richiesta singola è stata accettata nell'attuale stato di allocazione del sistema. È un approccio coerente con la tecnica per evitare i deadlock? Giustificare la risposta fornendo elementi riguardo la sua correttezza oppure fornendo un controc esempio.

8.14. Un sistema a risorsa singola contiene  $Risorse\_totali_s$  unità della classe di risorsa  $R_s$ . Se il sistema contiene  $n$  processi, mostrare che non si può verificare un deadlock se sono soddisfatte alcune delle seguenti condizioni (vedere la notazione usata nell'Algoritmo 8.2).

- a. Per ogni  $i : Risorse\_necessarie_{i,s} \leq Risorse\_totali_s/n$
- b.  $\sum_i Risorse\_necessarie_{i,s} \leq Risorse\_totali_s$
- c.  $\sum_i Risorse\_necessarie_{i,s} \leq Risorse\_totali_s + n - 1$  e per ogni  $i$ ,  $1 \leq Risorse\_necessarie_{i,s} \leq Risorse\_totali_s$

8.15. In un sistema a risorsa singola che contiene  $Risorse\_totali_s$  unità della classe di risorsa  $R_s$ ,  $PA$  è definito come segue:

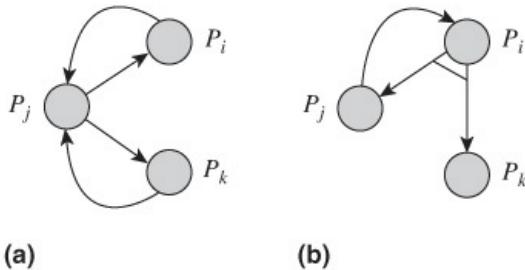
$PA = \{P_i \mid P_i \text{ è stato allocato con qualche risorsa ma non sono state raggiunte le sue richieste}\}$

Quale delle seguenti affermazioni sono vere (vedere la notazione usata nell'Algoritmo 8.2)? Giustificare la risposta.

- a. I "processi in  $PA$  saranno definitivamente in deadlock se  $\sum_i Risorse\_necessarie_{i,s} > Risorse\_totali_s$ ."
- b. I "processi in  $PA$  possono essere in deadlock solo se esiste qualche intero  $k$  tale che  $num\_proc(k) > Risorse\_totali_s/k$ , dove  $num\_proc(k)$  è il numero di processi in  $PA$  le cui richieste massime di unità della classe di risorsa  $R_s$  supera  $k$ ."
- 8.16. Il nuovo stato di allocazione di un sistema dopo aver accettato una richiesta di risorsa non è uno stato di allocazione sicuro secondo l'algoritmo del banchiere.
  - a. Ciò implica che si verificherà sicuramente un deadlock?
  - b. Il sistema riesce a effettuare una transizione in uno stato di allocazione sicuro? Se sì, dare un esempio che mostra una tale transizione.

- 8.17. Mostrare che un taglio delle risorse in un RRAG è una condizione necessaria e sufficiente per i deadlock nei sistemi SISR, MISR, SIMR e MIMR.
- 8.18. Un WFG viene utilizzato per rappresentare lo stato di allocazione di un sistema in cui le classi di risorsa possono contenere unità multiple e i processi possono effettuare richieste multiple di risorse (un sistema MIMR). Sviluppare una caratterizzazione di deadlock usando il WFG. (Suggerimento: un nodo nel WFG ha più

di un arco uscente in due casi: quando un processo richiede una unità di risorsa di una classe di risorsa a istanza-multipla, e quando un processo effettua una richiesta multipla. Questi sono detti archi uscenti OR e archi uscenti AND, rispettivamente. Per differenziare i due tipi di archi uscenti, gli archi uscenti AND di un processo sono uniti da una linea retta come mostrato in [Figura 8.14\(b\)](#). La [Figura 8.14\(a\)](#) mostra gli archi uscenti per il RRAG di [Figura 8.10](#), mentre la [Figura 8.14\(b\)](#) mostra gli archi uscenti per il RRAG di [Figura 8.11](#). (Tali archi uscenti hanno differenti implicazioni per l'individuazione dei deadlock!)



**Figura 8.14** WFG con archi uscenti multipli: (a) archi OR; (b) archi AND.

- 8.19. Un SO usa una semplice politica per gestire le situazioni di deadlock. Quando verifica che un insieme di processi è in deadlock, li interrompe tutti e li riavvia immediatamente. In quali condizioni il deadlock si riverificherà?
- 8.20. Un SO ha un solo disco, che usa (a) per creare file utente e (b) per creare una stampante virtuale per ogni processo. Lo spazio è allocato per entrambi gli usi in base alla domanda, e un processo è bloccato se le sue richieste di spazio su disco non possono essere accettate. Le richieste di stampa dirette a una stampante virtuale sono inviate a una stampante reale quando un processo termina. C'è possibilità di deadlock in questo sistema? Se sì, sotto quali condizioni? Suggerire una soluzione al problema del deadlock.
- 8.21. Un *deadlock inesistente* (*phantom deadlock*) è una situazione in cui un algoritmo di gestione deadlock dichiara un deadlock che non esiste effettivamente. Se i processi possono ritirare le loro richieste di risorse, mostrare che l'Algoritmo 8.1 può individuare i phantom deadlock. L'individuazione dei phantom deadlock può essere impedita?
- 8.22. Una strada attraversa un insieme di binari ferroviari in due punti. Alcune barriere sono poste sulla strada a ogni incrocio per fermare il traffico quando un treno sta per passare. Il traffico ferroviario viene fermato se un'auto blocca un binario. È consentita la marcia delle auto in entrambi i sensi di marcia ed è consentito il traffico ferroviario nei due sensi sui binari.
- Discutiamo se si possono verificare deadlock nel traffico stradale e ferroviario. Non ci sarebbero deadlock se sia il traffico stradale che quello ferroviario fossero a una sola direzione?
  - Progettare un insieme di semplici regole per evitare i deadlock nel traffico stradale e ferroviario.
- 8.23. Viene proposto l'uso dell'approccio di prevenzione dei deadlock per il problema dei filosofi a cena (Paragrafo 6.7.3) come segue: i posti al tavolo della cena sono numerati da 1 a  $n$ , e anche le forchette sono numerate da 1 a  $n$ , in modo che la forchetta alla sinistra del posto  $i$  possieda il numero  $i$ . I filosofi devono seguire la seguente regola: un filosofo deve prima prendere la forchetta con numerazione inferiore, poi prendere quella con numerazione maggiore. Mostrare che non si possono verificare deadlock in questo sistema.
- 8.24. Un insieme di processi  $D$  è in deadlock.
- Si osserva che:
- se un processo  $P_j \in D$  è interrotto, un insieme di processi  $D'$  c'è ancora in deadlock;
  - se un processo  $P_i \in D$  è interrotto, non esiste alcun deadlock nel sistema.

Illustrare alcune possibili ragioni per questa differenza e spiegarle con l'aiuto di un esempio. (Suggerimento: riferirsi alle Equazioni 8.4 e 8.5).

8.25. Dopo che l'Algoritmo 8.1 abbia determinato che un insieme di processi D è in deadlock, uno dei processi in D viene interrotto. Qual è il modo più efficiente per determinare se esiste un deadlock nel nuovo stato?

## Problemi avanzati

8.1. Si supponga che in un sistema siano presenti i processi P0, P1, P2, P3, P4 e un insieme di risorse di quattro tipi diversi A, B, C e D e di trovarsi nella seguente configurazione:

|                  | A | B       | C       | D |                 | A  | B | C | D |                     |  |
|------------------|---|---------|---------|---|-----------------|----|---|---|---|---------------------|--|
| $P_0$            | 4 | $X - 1$ | 3       | 2 |                 | 6  | 4 | 5 | 6 |                     |  |
| $P_1$            | 8 | 0       | $Y - 2$ | 2 |                 | 10 | 7 | 6 | 8 |                     |  |
| $P_2$            | 4 | 0       | 0       | 0 |                 | 6  | 2 | 0 | 8 |                     |  |
| $P_3$            | 0 | 0       | 3       | 2 |                 | 0  | 3 | 4 | 2 |                     |  |
| $P_4$            | 2 | 1       | $Z + 1$ | 4 |                 | 9  | 1 | 6 | 9 |                     |  |
| Risorse allocate |   |         |         |   | Risorse massime |    |   |   |   | Risorse disponibili |  |

Determinare gli intervalli dei valori interi di X, Y e Z per i quali il sistema si trova in uno stato sicuro.

## Note bibliografiche

Di Dijkstra (1965), Havender (1968) e Habermann (1969) sono i primi lavori sulla gestione dei deadlock. Dijkstra (1965) e Habermann (1969) hanno trattato l'algoritmo del banchiere. Coffman et al. (1971) hanno trattato l'algoritmo di individuazione dei deadlock per un sistema contenente istanze multiple di risorse. Holt (1972) ha fornito un grafo per la caratterizzazione teorica dei deadlock. Isloor e Marsland (1980) hanno realizzato una buona sintesi di questo argomento. A Zobel (1983) si deve un'ampia bibliografia. Howard (1973) ha trattato l'approccio pratico al deadlock descritto nel Paragrafo 8.9. Tay e Loke (1995) e Levine (2003) hanno trattato la caratterizzazione dei deadlock.

Bach (1986) descrive la gestione dei deadlock in Unix.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
2. Coffman, E.G., M.S. Elphick, and A. Shoshani (1971): "System deadlock," *Computing Surveys*, **3** (2), 67-78.
3. Dijkstra, E.W. (1965) : "Cooperating sequential processes," Technical report EWD-123, Technlogical University, Eindhoven.
4. Habermann, A.N. (1969): "Prevention of System deadlock," *Communications of the ACM*, **12** (7), 373-377.
5. Habermann, A.N. (1973): "A new approach to avoidance of system deadlock," in *Lecture notes in Computer Science*, Vol. 16, Springer-Verlag.
6. Havender, J.W. (1968): "Avoiding deadlock in multitasking systems," *IBM Systems Journal*, **7** (2), 74-84.
7. Holt, R.C. (1972): "Some deadlock properties of computer systems," *Computing Surveys*, **4** (3), 179-196.
8. Howard, J.H. (1973): "Mixed solutions to the deadlock problem," *Communications of the ACM*, **6** (3), 427-430.
9. Isloor, S.S., and T.A. Marsland (1980): "The deadlock problem - an overview," *Computer*, **13** (9), 58-70.
10. Levine, G. (2003): "Defining deadlock," *Operating Systems Review*, **37**, 1.
11. Rypka, D.J., and A.P. Lucido (1979): "Deadlock detection and avoidance for shared logical resources," *IEEE Transactions on Software Engineering*, **5** (5), 465-471.
12. Tay, Y.C., and W.T. Loke (1995): "On Deadlock of exclusive AND-requests for resources," *Distributed Computing*, Springer Verlag, **9**, 2, 77-94.
13. Zobel, D. (1983): "The deadlock problem - a classifying bibliography," *Operating*



---

# CAPITOLO 9

## Sincronizzazione dei processi: message passing

---

### Obiettivi di apprendimento

- Definizione ed implementazione del message passing
- Mailbox
- Message passing nei protocolli di alto livello
- Message passing nei sistemi operativi: Unix, Windows

Lo scambio di messaggi (message passing) si adatta a diverse situazioni in cui lo scambio di informazioni tra processi gioca un ruolo fondamentale. Uno dei suoi usi più importanti è nel paradigma *client-server*, in cui un processo *server* fornisce un servizio, e altri processi, chiamati *client*, gli inviano messaggi per utilizzare tale servizio. Questo paradigma è molto utilizzato. Per esempio, un SO basato su microkernel implementa come server alcune funzionalità come lo scheduling, un SO convenzionale offre servizi come la stampa mediante l'uso di server, e, su Internet, una varietà di servizi sono offerti dai Web server. Un altro importante utilizzo dello scambio di messaggi è nei protocolli di alto livello per lo scambio di e-mail e per la comunicazione tra i task nei programmi paralleli o distribuiti. In questo caso, lo scambio di messaggi è utilizzato per scambiare informazioni, mentre altre parti del protocollo vengono utilizzate per garantire l'affidabilità.

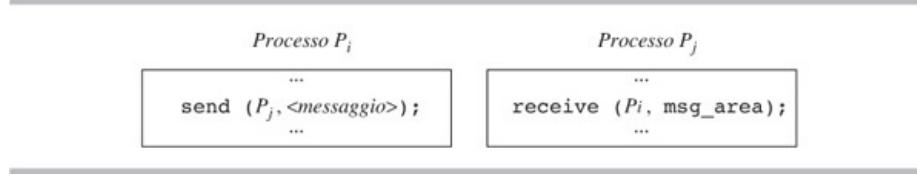
Le problematiche chiave nel message passing riguardano l'identificazione reciproca dei processi che inviano e ricevono i messaggi, l'esecuzione da parte del kernel delle azioni relative alla consegna dei messaggi, le modalità di memorizzazione, consegna dei messaggi e la politica di sospensione dei processi in attesa della consegna dei messaggi. Queste caratteristiche sono specifiche di ogni sistema operativo.

Descriveremo differenti configurazioni per lo scambio dei messaggi adottate nei sistemi operativi e discuteremo il loro impatto sui processi utente e sul kernel. Inoltre, descriveremo l'implementazione del message passing nei sistemi operativi Unix e Windows.

### 9.1 Panoramica sul message passing

Nel Paragrafo 5.2.5 abbiamo elencato quattro modalità di interazione tra processi – *condivisione dei dati*, *message passing*, *sincronizzazione* e *segnali* ([Tabella 5.7](#)). Di questi, abbiamo affrontato la condivisione dei dati e la sincronizzazione nel [Capitolo 6](#) e i segnali nel [Capitolo 5](#). La condivisione dei dati fornisce un modo per accedere ai valori dei dati condivisi in maniera mutuamente esclusiva. La sincronizzazione dei processi viene eseguita bloccando un processo finché gli altri processi non hanno eseguito alcune azioni specifiche. Le funzionalità del message passing coincidono con quelle della condivisione dei dati e della sincronizzazione; tuttavia, ogni forma di interazione tra processi ha una sua specifica area di applicazione. Discuteremo tale aspetto dopo aver fatto una panoramica sullo scambio di messaggi.

La [Figura 9.1](#) mostra un esempio di message passing. Il processo  $P_i$  invia un messaggio al processo  $P_j$  eseguendo l'istruzione `send` ( $P_j, <\text{messaggio}>$ ). Il codice compilato dell'istruzione `send` richiama la funzione di libreria `send`. La `send` effettua una chiamata di sistema `send`, con due parametri  $P_j$  e il messaggio. L'esecuzione dell'istruzione `receive` ( $P_i, \text{msg\_area}$ ), dove `msg_area` è un'area dello spazio di indirizzamento di  $P_j$ , consiste in una chiamata di sistema `receive`.



**Figura 9.1** Message passing.

La semantica dello scambio di messaggi è la seguente: al momento di una chiamata *send* da parte di  $P_i$ , il kernel controlla se il processo  $P_j$  è bloccato su una chiamata *receive*. In questo caso, copia il messaggio in `msg_area` e attiva  $P_j$ . Se il processo  $P_j$  non ha ancora effettuato un chiamata *receive*, il kernel imposta la consegna del messaggio appena  $P_j$  effettua la chiamata *receive*. Quando il processo  $P_j$  riceve il messaggio, lo interpreta ed esegue l'azione appropriata.

I messaggi possono essere scambiati tra processi in esecuzione sullo stesso computer o su computer differenti connessi tramite una rete. Inoltre, i processi che partecipano allo scambio dei messaggi possono decidere il significato di uno specifico messaggio e le azioni che il processo destinatario deve intraprendere alla ricezione. Grazie a questa flessibilità, il message passing è utilizzato nelle seguenti applicazioni:

- nel paradigma *client-server*, per le comunicazioni tra le componenti di un sistema operativo basato su micro-kernel, per fornire i servizi di stampa ai processi o per fornire servizi Web ai processi client in esecuzione su altri computer;
- come supporto dei protocolli di alto livello usati per le comunicazioni tra computer o per fornire la funzione di posta elettronica;
- per implementare la comunicazione tra task in un programma parallelo o distribuito.

In linea di principio, lo scambio di messaggi può essere eseguito utilizzando alcune variabili condivise. Per esempio, `msg_area` in [Figura 9.1](#) potrebbe essere una variabile condivisa.  $P_i$  potrebbe memorizzare un valore o un messaggio nella variabile e  $P_j$  potrebbe leggerlo da quella posizione. Tuttavia, questo approccio non è efficiente poiché in coppia di processi dovrebbe creare una variabile condivisa della dimensione corretta e condividerla. Inoltre, dovrebbero utilizzare un meccanismo di sincronizzazione analogo a quello del problema produttore-consamatore (Paragrafo 6.7.1) per garantire che un processo destinatario acceda a un messaggio memorizzato in una variabile condivisa solo dopo che un processo mittente lo abbia memorizzato in quella posizione. Il message passing è molto più semplice in questa situazione. Inoltre è più generale, poiché può essere usato in un ambiente distribuito, dove l'approccio mediante variabile condivisa non è utilizzabile.

Il problema produttore-consamatore con buffer singolo, un solo processo produttore e un singolo processo consumatore, può essere implementato con il message passing come mostrato in [Figura 9.2](#). La soluzione non usa alcuna variabile condivisa. Piuttosto, il processo  $P_i$ , ovvero il processo produttore, ha una variabile chiamata *buffer* e il processo  $P_j$ , ovvero il processo consumatore, ha una variabile chiamata *message\_area*. Il processo produttore produce, inserisce nel *buffer* e invia il contenuto del *buffer* in un messaggio al consumatore. Il consumatore riceve il messaggio nella *message\_area* e da lì lo consuma. La chiamata di sistema *send* blocca il processo produttore finché il messaggio non viene consegnato al consumatore, e la chiamata di sistema *receive* blocca il consumatore finché il messaggio non gli viene inviato.

---

```

begin
  Parbegin
    var buffer : . . . ;
    repeat
      { produci elemento nel buffer }
      send (Pj, buffer);
      { altre istruzioni del ciclo }
    forever;
  Parend;
end.

```

*Processo P<sub>i</sub>*

---

```

var area_messaggio : . . . ;
repeat
  receive (Pi, area_messaggi);
  { consuma nell'area_messaggi }
  { altre istruzioni del ciclo }
forever;

```

*Processo P<sub>j</sub>*

---

**Figura 9.2** Soluzione al problema produttore-consumatore utilizzando lo scambio dei messaggi.

La soluzione al problema produttore-consumatore di [Figura 9.2](#) è molto più semplice rispetto alle soluzioni discusse nel [Capitolo 6](#); tuttavia, è restrittiva poiché consente di avere un singolo processo produttore e un singolo consumatore. Nel caso generale, è efficiente utilizzare le tecniche di sincronizzazione dei processi discusse nel [Capitolo 6](#) per implementare un sistema in cui sono presenti più produttori e consumatori.

### **Problematiche del message passing**

Due sono le problematiche importanti dello scambio dei messaggi:

- *identificazione per nome (Naming) dei processi*: quando i nomi dei processi mittente e destinatario sono indicati esplicitamente nelle istruzioni `send` e `receive`, o se le rispettive identità sono dedotte dal kernel in un altro modo;
- *consegna dei messaggi*: quando un processo mittente resta bloccato finché il messaggio inviato non viene consegnato, l'ordine con il quale i messaggi vengono consegnati al processo destinatario, e come vengono gestite le condizioni eccezionali.

Queste problematiche guidano le scelte implementative e inoltre influenzano la generalità dello scambio dei messaggi. Per esempio, se è richiesto che un processo mittente conosca l'identità di un processo destinatario, l'ambito di applicazione del message passing sarebbe limitato ai processi all'interno della stessa applicazione. Un requisito meno vincolante consentirebbe di estendere lo scambio dei messaggi a processi in differenti applicazioni e a processi in esecuzione su sistemi differenti. In maniera simile, l'utilizzo della consegna dei messaggi in ordine FCFS può essere alquanto restrittivo; i processi possono voler ricevere i messaggi in qualche altro ordine.

#### **9.1.1 Denominazione diretta e indiretta**

Nella tecnica di *denominazione diretta*, i processi mittente e destinatario dichiarano il proprio nome. Per esempio, le istruzioni `send` e `receive` potrebbero avere la seguente sintassi:

```

send (<processo_destinazione>, <lunghezza_messaggio>,
      <indirizzo_messaggio>);
receive (<processo_sorgente>, <message_area>);

```

dove `<processo_destinazione>` e `<processo_sorgente>` sono i nomi dei processi (tipicamente, sono gli id di processo assegnati dal kernel), `<indirizzo_messaggio>` è l'indirizzo dell'area di memoria nello spazio di indirizzamento del processo mittente che contiene la forma testuale del messaggio da inviare, e `<message_area>` è l'area di memoria nello spazio di indirizzamento del processo destinatario in cui il messaggio deve essere consegnato. I processi di [Figura 9.2](#) utilizzano la denominazione diretta.

Tale tecnica può essere utilizzata in due modi. Nella tecnica basata sui *nomi simmetrici*, sia il processo mittente che quello destinatario specificano i rispettivi nomi. In questo modo, un processo può decidere da quale processo ricevere un messaggio. Tuttavia, deve conoscere il nome di ogni processo che vuole inviargli messaggi; cosa difficoltosa quando i processi di applicazioni differenti vogliono comunicare, o quando un server vuole ricevere una richiesta da ogni processo in un insieme di client. Con l'uso dei

*nomi asimmetrici*, il destinatario non fornisce il nome del processo da cui vuole ricevere un messaggio; il kernel inoltra un messaggio inviatogli da *qualche* processo.

Nella tecnica di *denominazione indiretta*, i processi non specificano i rispettivi nomi nelle istruzioni *send* e *receive*. Discuteremo la tecnica dei nomi indiretti nel Paragrafo 9.3.

### 9.1.2 Send bloccanti e non-bloccanti

Una *send* bloccante blocca il processo mittente finché il messaggio da inviare non viene consegnato al processo destinatario. Questa metodologia di message passing è chiamata *scambio di messaggi sincrono*. Una chiamata *send* non bloccante consente a un mittente di proseguire la propria esecuzione dopo aver effettuato una chiamata *send*, senza preoccuparsi dell'immediata consegna del messaggio; tale scambio di messaggi è chiamato message passing *asincrono*. In entrambi i casi, la primitiva *receive* è tipicamente bloccante.

Il message passing sincrono fornisce alcune proprietà per i processi utente e semplifica le azioni del kernel. Un processo mittente ha la garanzia che il messaggio inviato venga consegnato prima di poter continuare la propria esecuzione. Questa caratteristica semplifica la progettazione dei processi concorrenti. Il kernel consegna il messaggio immediatamente se il processo destinatario ha già effettuato una chiamata *receive* per ricevere un messaggio; altrimenti, blocca il processo mittente finché il processo ricevente non effettua una chiamata *receive*. Il kernel può semplicemente lasciare il messaggio nell'area di memoria del mittente finché non viene consegnato. Tuttavia, l'uso delle *send* bloccanti presenta una controindicazione, può cioè ritardare un processo mittente in alcune situazioni, quale per esempio, durante la comunicazione con un server di stampa sovraccarico.

Il message passing asincrono migliora la concorrenza tra i processi mittente e destinatario consentendo al processo mittente di continuare la propria esecuzione. Tuttavia, causa anche un problema di sincronizzazione poiché il mittente non dovrebbe alterare il contenuto dell'area di memoria che contiene il testo del messaggio finché il messaggio non viene consegnato. Per superare questo problema, il kernel esegue il *buffering del messaggio*: quando un processo effettua una chiamata *send*, il kernel alloca un buffer nell'area di sistema e copia il messaggio nel buffer. In questo modo, il processo mittente è libero di accedere all'area di memoria che conteneva il testo del messaggio. Tuttavia, questa organizzazione coinvolge un sostanziale impiego di memoria per i buffer quando molti messaggi sono in attesa di essere consegnati. Inoltre, consuma tempo di CPU, dal momento che un messaggio deve essere copiato due volte: una volta nel buffer di sistema quando viene effettuata un chiamata *send* e, successivamente, nell'area del messaggio del destinatario al momento della consegna.

### 9.1.3 Eccezioni nel message passing

Per facilitare la gestione delle eccezioni, le chiamate *send* e *receive* prevedono due ulteriori parametri. Il primo parametro è un insieme di flag che indica come il processo vuole che vengano gestite le eccezioni: chiameremo questo parametro *flag*. Il secondo parametro è l'indirizzo di un'area di memoria in cui il kernel scrive un codice che descrive l'esito della chiamata *send* o *receive*; chiameremo quest'area *status\_area*.

Quando un processo effettua una chiamata *send* o *receive*, il kernel scrive un codice di stato nella *status\_area*. Successivamente controlla i *flag* per decidere se deve gestire qualche eccezione ed eventualmente esegue le azioni necessarie. Infine restituisce il controllo al processo. Il processo controlla il codice di stato fornito dal kernel e gestisce ogni eccezione che vuole gestire autonomamente.

Seguono alcune eccezioni con le rispettive azioni sono le seguenti.

1. Il processo destinatario specificato in una chiamata *send* non esiste.
2. Nella tecnica a denominazione simmetrica, il processo sorgente specificato in una chiamata *receive* non esiste.
3. Una chiamata *send* non può essere elaborata perché il kernel non ha memoria sufficiente per i buffer.
4. Non esiste alcun messaggio per un processo che effettua una chiamata *receive*.
5. Un insieme di processi entra in una situazione di deadlock quando un processo è bloccato su una chiamata *receive*.

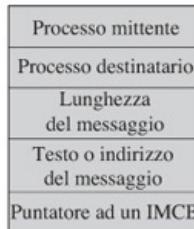
Nei casi 1 e 2, il kernel può terminare il processo che ha effettuato la chiamata *send* o *receive* e imposta il codice di terminazione in modo da descrivere la condizione di eccezione. Nel caso 3, il processo mittente può essere bloccato finché non si libera spazio per il buffer. Il caso 4 non è un'eccezione effettiva se le *receive* sono bloccanti (generalmente lo sono!), ma può essere trattato come un'eccezione in modo che il processo destinatario abbia un'opportunità di gestire la condizione se lo vuole. Un processo può preferire l'azione standard, ovvero che il kernel blocchi il processo finché non arriva un messaggio a esso indirizzato, o può preferire un'azione propria, come l'attesa di una quantità di tempo specificata prima di rinunciare.

Eccezioni più importanti appartengono all'ambito delle politiche del SO. La situazione di stallo (deadlock) del caso 5 ne è un esempio. Molti sistemi operativi non gestiscono questa particolare eccezione poiché crea overhead per il rilevamento del deadlock. Anche le situazioni di difficile gestione, come un processo in attesa per un lungo periodo su una chiamata *receive*, appartengono all'ambito delle politiche del SO.

## 9.2 Implementazione del message passing

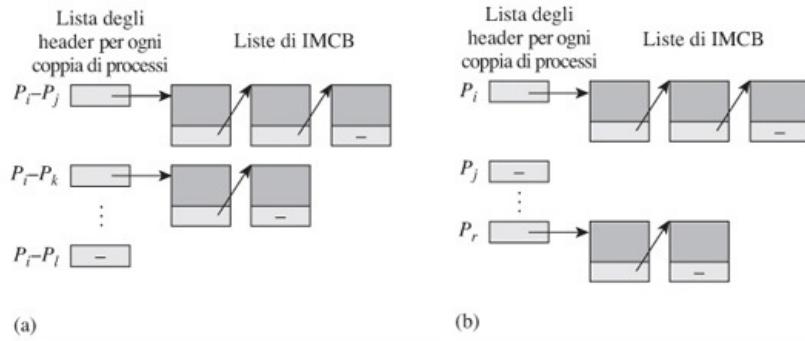
### 9.2.1 Buffering dei messaggi tra processi

Quando un processo  $P_i$  invia un messaggio a qualche processo  $P_j$  utilizzando una *send* non bloccante, il kernel costruisce un *interprocess message control block* (IMCB) per memorizzare tutte le informazioni necessarie per consegnare il messaggio (Figura 9.3). Il control block contiene i nomi dei processi mittente e destinatario, la lunghezza del messaggio e il testo del messaggio. Per il control block è allocato un buffer nell'area kernel. Quando un processo  $P_j$  effettua una chiamata *receive*, il kernel copia il messaggio dall'IMCB appropriato nell'area messaggi fornita da  $P_j$ .



**Figura 9.3** Interprocess message control block (IMCB).

I campi puntatore degli IMCB vengono usati per creare liste di IMCB al fine di semplificare la consegna dei messaggi. La Figura 9.4 mostra l'organizzazione delle liste di IMCB quando si utilizzano le *send* bloccanti e la consegna dei messaggi in ordine FCFS. Nella tecnica a denominazione simmetrica per ogni coppia di processi in comunicazione viene utilizzata una lista separata. Quando il processo  $P_i$  effettua una chiamata *receive* per ricevere un messaggio dal processo  $P_j$ , la lista degli IMCB per la coppia  $P_i$ - $P_j$  viene utilizzata per consegnare il messaggio. Nella tecnica a denominazione asimmetrica può essere mantenuta una singola lista di IMCB per ogni processo. Quando un processo effettua una *receive*, viene elaborato il primo IMCB nella sua lista per consegnare un messaggio.



**Figura 9.4** Liste di IMCB per *send* bloccanti con denominazione (a) simmetrica e (b) asimmetrica.

Se si utilizzano le *send* bloccanti, a ogni istante di tempo può essere non consegnato al massimo un messaggio inviato da un processo. Il processo viene bloccato finché il messaggio non viene consegnato. Dunque non è necessario copiare il messaggio in un IMCB. Il kernel può semplicemente annotare l'indirizzo del testo del messaggio nell'area di memoria del mittente, e utilizzare questa informazione per la consegna del messaggio. Questa organizzazione consente di risparmiare un'operazione di copia del messaggio. Tuttavia, crea delle difficoltà se il mittente è riversato sul disco prima che il messaggio venga consegnato, per cui può essere preferibile utilizzare un IMCB. Sarebbe necessario un numero inferiore di IMCB rispetto all'utilizzo delle *send* non bloccanti, poiché a ogni istante di tempo al più un messaggio inviato da ogni processo può essere in un IMCB.

Il kernel può dover riservare una considerevole quantità di memoria per i messaggi tra processi, particolarmente se si utilizzano le *send* non bloccanti. In questi casi, può memorizzare il testo dei messaggi sul disco. Pertanto un IMCB conterebbe l'indirizzo del blocco del disco dove è memorizzato il messaggio, piuttosto che il testo del messaggio.

### 9.2.2 Consegnare dei messaggi tra processi

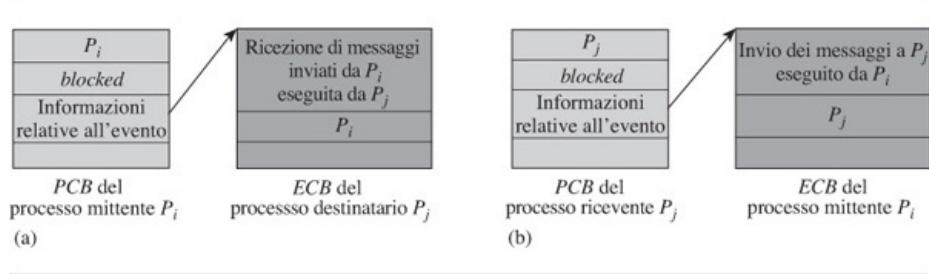
Quando un processo  $P_i$  invia un messaggio al processo  $P_j$ , il kernel consegna il messaggio a  $P_j$  immediatamente se  $P_j$  è attualmente bloccato su una *receive* per un messaggio da  $P_i$ , o da qualsiasi altro processo. Dopo aver consegnato il messaggio, il kernel deve anche cambiare lo stato di  $P_j$  a *ready*. Se il processo  $P_j$  non ha ancora effettuato una chiamata *receive*, il kernel deve organizzare la consegna del messaggio quando  $P_j$  effettua la chiamata *receive*. In questo modo, le azioni per la consegna dei messaggi si verificano sia al momento della *send* che della *receive*.

Si ricordi dal Paragrafo 5.2.4 che il kernel utilizza un *event control block* (ECB) per annotare le azioni che dovrebbero essere effettuate quando si verifica un evento anticipato. L'ECB contiene tre campi:

- la descrizione dell'evento anticipato;
- l'identificativo del processo in attesa dell'evento;
- un puntatore a un ECB per la creazione delle liste di ECB.

La [Figura 9.5](#) mostra l'uso degli ECB per l'implementazione del message passing mediante la tecnica a denominazione simmetrica e *send* bloccanti. Quando  $P_i$  effettua una chiamata *send*, il kernel controlla se esiste un ECB per la chiamata *send* da parte di  $P_i$ , ovvero, se  $P_j$  ha effettuato una chiamata *receive* ed è in attesa dell'invio del messaggio da parte di  $P_i$ . Se non è questo il caso, il kernel sa che la chiamata *receive* potrebbe verificarsi in qualche momento futuro, per cui crea un ECB per l'evento "ricevi da  $P_i$  per  $P_j$ " e specifica  $P_i$  come processo che sarà influenzato dall'evento. Lo stato del processo  $P_i$  viene impostato a *blocked* e l'indirizzo dell'ECB è inserito nel campo relativo alle informazioni sull'evento del suo PCB [[Figura 9.5\(a\)](#)]. La [Figura 9.5\(b\)](#) illustra il caso in cui il processo  $P_j$  effettua una chiamata *receive* prima che  $P_i$  effettui una chiamata *send*. Viene creato un ECB per un evento "invia a  $P_j$  da  $P_i$ ". L'identificativo di  $P_j$  è inserito nell'ECB per indicare che lo stato di  $P_j$  sarà influenzato quando si verificherà l'evento

relativo alla *send*.



**Figura 9.5** Utilizzo degli ECB per implementare la tecnica a denominazione simmetrica con chiamate bloccanti: (a) *send*; (b) *receive*.

La [Figura 9.6](#) mostra i dettagli completi delle azioni eseguite dal kernel per implementare lo scambio dei messaggi utilizzando la tecnica a denominazione simmetrica e le *send* bloccanti. Per le ragioni menzionate in precedenza, il kernel crea un IMCB anche se il processo mittente è bloccato fino alla consegna del messaggio. Quando un processo  $P_i$  invia un messaggio al processo  $P_j$ , il kernel prima controlla se la *send* è stata anticipata, ovvero, se era stato creato un ECB per l'evento *send*. Questo dovrebbe accadere se il processo  $P_j$  ha già effettuato una chiamata *receive* per ricevere un messaggio da  $P_i$ . In questo caso, l'azione  $S_3$  consiste nella consegna immediata del messaggio a  $P_j$  e modifica il suo stato da *blocked* a *ready*. L'ECB e l'IMCB vengono distrutti. Se non esiste un ECB per la *send*, al passo  $S_4$  viene creato un ECB per una chiamata *receive* da parte del processo  $P_j$ , che ora è anticipato, il processo mittente è bloccato, e l'IMCB è inserito nella lista degli IMCB del processo  $P_j$ . Le azioni inverse sono eseguite al momento di una chiamata *receive*: se si è già verificata una *send* corrispondente, un messaggio viene consegnato al processo  $P_j$  e  $P_i$  viene attivato; altrimenti, viene creato un ECB per una chiamata *send* e  $P_j$  viene bloccato.

---

| Durante l'invio di un messaggio verso $P_j$ da parte di $P_i$ : |   |
|---|---|
| Passo   | Descrizione   |
| $S_1$   | <i>Crea un IMCB e inizializza i suoi campi;</i>   |
| $S_2$   | <i>Se esiste un ECB per l'evento "invia un messaggio da <math>P_i</math> a <math>P_j</math>"</i>  |
| $S_3$   | <b>allora</b> <ul style="list-style-type: none"> <li>(a) <i>Spedisci il messaggio a <math>P_j</math>;</i></li> <li>(b) <i>Attiva <math>P_j</math>;</i></li> <li>(c) <i>Distruggi ECB e IMCB;</i></li> <li>(d) <i>Ritorna a <math>P_i</math>;</i></li> </ul>   |
| $S_4$   | <b>altrimenti</b> <ul style="list-style-type: none"> <li>(a) <i>Crea un ECB per l'evento "ricevi un messaggio per <math>P_j</math> da <math>P_i</math>" e identifica l'id di <math>P_i</math> come il processo in attesa dell'evento;</i></li> <li>(b) <i>Poni <math>P_i</math> nello stato bloccato e inserisci l'indirizzo dell'ECB nel PCB di <math>P_i</math>;</i></li> <li>(c) <i>Aggiungi l'IMCB nella lista degli IMCB di <math>P_j</math>;</i></li> </ul> |
| Durante la ricezione di un messaggio per $P_j$ spedito da $P_i$ |   |
| Passo   | Descrizione   |
| $R_1$   | <i>Se esiste un ECB per l'evento "ricevi un messaggio per <math>P_j</math> da <math>P_i</math>"</i>   |
| $R_2$   | <b>allora</b> <ul style="list-style-type: none"> <li>(a) <i>Spedisci il messaggio dall'opportuno IMCB nella lista di <math>P_j</math>;</i></li> <li>(b) <i>Attiva <math>P_j</math>;</i></li> <li>(c) <i>Distruggi ECB e IMCB;</i></li> <li>(d) <i>Ritorna a <math>P_j</math>;</i></li> </ul>  |
| $R_3$   | <b>altrimenti</b> <ul style="list-style-type: none"> <li>(a) <i>Crea un ECB per l'evento "invia un messaggio da <math>P_i</math> per <math>P_j</math>" e identifica l'id di <math>P_j</math> come il processo in attesa dell'evento;</i></li> <li>(b) <i>Poni <math>P_j</math> nello stato bloccato e inserisci l'indirizzo dell'ECB nel PCB di <math>P_j</math>;</i></li> </ul>  |

---

**Figura 9.6** Azioni del kernel nel message passing utilizzando la tecnica a denominazione simmetrica e le send bloccanti.

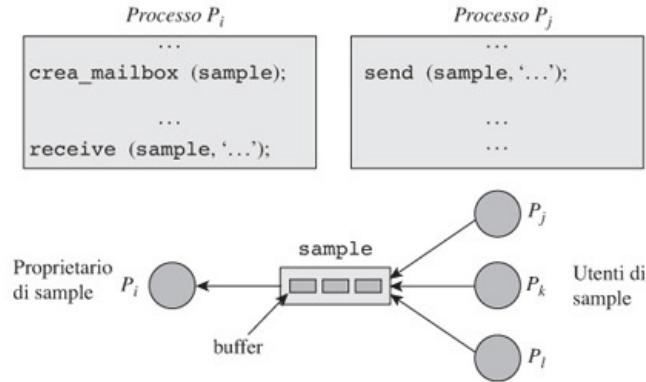
Le azioni intraprese quando si utilizzano le *send* non bloccanti sono più semplici. Non è necessario bloccare e attivare il mittente [Passi  $S_4(b)$  e  $R_2(b)$  in [Figura 9.6](#)]. Quando un messaggio da inviare non può essere consegnato immediatamente [Passo  $S_4(a)$ ] non è necessaria la creazione di un ECB poiché il mittente non è bloccato finché il messaggio non viene consegnato.

### 9.3 Mailbox

Una mailbox (cassetta postale) è un contenitore dei messaggi scambiati tra i processi e possiede le seguenti caratteristiche.

1. Ha un nome unico.
2. Il proprietario della mailbox è tipicamente il processo che l'ha creata. Solo il proprietario del processo può ricevere i messaggi da una mailbox.
3. Ciascun processo che è a conoscenza del nome della mailbox gli può inviare messaggi (al processo *utente* della mailbox). In questo modo, i processi mittenti e destinatari utilizzano il nome di una mailbox, piuttosto che i rispettivi nomi, nelle istruzioni *send* e *receive*; è un'istanza a *denominazione indiretta* (Paragrafo 9.1.1).

La [Figura 9.7](#) illustra lo scambio di messaggi mediante l'utilizzo di una mailbox *campione*. Il processo  $P_j$  invia un messaggio alla mailbox, utilizzando il nome della mailbox nell'istruzione *send*. Se  $P_i$  non ha già eseguito un'istruzione *receive*, il kernel conserva il messaggio in un buffer. Il kernel può associare un insieme limitato di buffer a ogni mailbox, o può allocare i buffer da un insieme condiviso di buffer al momento dell'invio di un messaggio. Entrambe le istruzioni *crea\_mailbox* e *send* restituiscono codici di stato.



**Figura 9.7** Creazione e uso della mailbox *sample*.

Il kernel può fornire un insieme limitato di nomi di mailbox o può consentire ai processi utente di assegnare i nomi alle mailbox a scelta. Nel primo caso, non può essere garantita la confidenzialità della comunicazione tra una coppia di processi poiché ogni processo può utilizzare qualsiasi mailbox. La confidenzialità aumenta quando i processi possono scegliere il nome da assegnare alle mailbox.

Per controllare la creazione e la distruzione delle mailbox, il kernel può richiedere che un processo si “connetta” esplicitamente a una mailbox prima di utilizzarla, e si disconnetta quando ha finito di utilizzarla. In questo modo il kernel può distruggere la mailbox se non vi sono processi connessi. Alternativamente, può consentire al proprietario di una mailbox di distruggerla. In questo caso, ha la responsabilità di comunicare la distruzione a tutti i processi “connessi” alla mailbox. Il kernel può consentire al proprietario di una mailbox di trasferire la proprietà a un altro processo.

L’uso di una mailbox ha i seguenti vantaggi:

- *anonimato del destinatario*: un processo che invia un messaggio per richiedere un servizio può non avere interesse all’identità del processo destinatario, fintanto che il processo destinatario non può eseguire la funzione richiesta. Una mailbox consente al processo mittente di non conoscere l’identità del destinatario. Inoltre, se il SO permette di cambiare la proprietà di una mailbox dinamicamente, un processo può prontamente rilevare un altro servizio;
- *classificazione dei messaggi*: un processo può creare diverse mailbox e usare ogni mailbox per ricevere i messaggi di un tipo specifico. Questa organizzazione consente una semplice classificazione dei messaggi (Esempio 9.3).

L’anonimato di un processo destinatario, come abbiamo appena visto, può offrire l’opportunità di trasferire una funzione da un processo a un altro. Si consideri un SO il cui kernel è strutturato sotto forma di processi multipli che comunicano mediante messaggi. Gli interrupt rilevanti per la funzione di scheduling dei processi possono essere modellati come messaggi inviati a una mailbox chiamata *scheduling*. Se il SO vuole utilizzare diversi criteri di scheduling per diversi periodi del giorno, può implementare diversi scheduler come processi e trasferire la proprietà di appartenenza della mailbox *scheduling* a questi processi. In questo modo, il processo scheduler che attualmente possiede la mailbox *scheduling* può ricevere tutti i messaggi relativi allo scheduling. Le funzionalità dei server di un SO possono essere trasferite in maniera analoga. Per esempio, tutte le richieste di stampa possono essere dirette a una stampante laser piuttosto che a una stampante a matrice di punti semplicemente cambiando la proprietà di una mailbox *stampa*.

Sebbene un processo possa anche rimanere anonimo quando invia un messaggio a una mailbox, l’identità del mittente spesso deve essere nota. Per esempio, un server può essere programmato per restituire le informazioni di stato di ogni richiesta. Questo obiettivo può essere ottenuto passando l’id del mittente con il testo del messaggio. Il mittente del messaggio, d’altra parte, potrebbe non conoscere l’identità del server; quindi, dovrebbe ricevere la risposta del server mediante una *receive* asimmetrica. Come alternativa, il compilatore può implementare la chiamata *send* come chiamata bloccante che richiede una risposta contenente le informazioni di stato; in tal modo la restituzione

delle informazioni di stato sarebbe una responsabilità del kernel.

### Esempio 9.1 - Uso delle mailbox

Un sistema di prenotazioni di una compagnia aerea si compone di un database centralizzato e di un insieme di processi di prenotazione; ogni processo rappresenta un agente di prenotazione. La [Figura 9.8](#) mostra lo pseudocodice per il server di prenotazione. Utilizza tre mailbox chiamate *richiesta*, *prenotazione*, e *cancellazione* e si aspetta che un processo di prenotazione invii messaggi di richiesta, prenotazione e cancellazione, rispettivamente, a queste mailbox. I valori dei flag nelle chiamate *receive* sono scelti in modo tale che una chiamata *receive* restituisca un codice di errore se non esistono messaggi. Per migliorare l'efficacia, il server elabora tutti messaggi di cancellazione pendenti prima di elaborare una prenotazione o una richiesta, ed effettua le prenotazioni prima delle richieste.

```
repeat
    while receive (prenota, flags1, msg_area1) ritorna al messaggio
        while receive (cancella, flags2, msg_area2) ritorna al messaggio
            esegui la cancellazione;
            esegui la prenotazione,
        if receive (richiedi_informazioni, flags3, msg_area3) ritorna al messaggio then
            while receive (cancella, flags2, msg_area2) ritorna al messaggio
                esegui la cancellazione;
                elabora la richiesta di informazioni;
    forever
```

**Figura 9.8** Un server di prenotazioni di una compagnia aerea che utilizza tre mailbox: *richiesta*, *prenotazione* e *cancellazione*.

## 9.4 Protocolli di alto livello che utilizzano il message passing

In questo paragrafo, discutiamo tre protocolli che utilizzano il paradigma del message passing per fornire diversi servizi. Il *simple mail transfer protocol* (SMTP) consegna le e-mail. Le *remote procedure call* (RPC) sono una caratteristica dei linguaggi di programmazione per l'*elaborazione distribuita*; viene usata per richiamare una parte di un programma presente in un computer diverso. *Parallel virtual machine* (PVM) e *message passing interface* (MPI) sono standard per lo scambio di messaggi per la programmazione parallela.

### 9.4.1 Il Simple Mail Transfer Protocol (SMTP)

Il protocollo SMTP è utilizzato per consegnare e-mail a uno o più utenti in modo affidabile ed efficiente. Utilizza la tecnica a denominazione asimmetrica (Paragrafo 9.1.1). Una mail viene consegnata al terminale di un utente se l'utente è attivo altrimenti viene depositata nella mailbox dell'utente. Il protocollo SMTP può consegnare mail attraverso un certo numero di ambienti di comunicazione tra processi (IPCE), dove un IPCE può coprire una parte di una rete, un'intera rete o diverse reti. SMTP è un protocollo di livello applicazione. Utilizza il TCP come protocollo di trasporto e IP come protocollo di routing. I dettagli di questi livelli di rete e i dettagli della consegna affidabile vanno, comunque, al di là dello scopo di questo capitolo; sono discussi successivamente nel [Capitolo 16](#).

SMTP è composto da diversi semplici comandi. Quelli rilevanti per i nostri scopi sono i seguenti: il comando MAIL indica chi sta inviando una mail. Contiene un percorso inverso lungo la rete, rappresentato da una lista opzionale di host e il nome della mailbox del mittente. Il comando RCPT indica chi deve ricevere la mail. Contiene un percorso rappresentato da una lista opzionale di host e una mailbox di destinazione. Un comando MAIL può essere eseguito da uno o più comandi RCPT. Il comando DATA contiene i dati da inviare ai destinatari. Dopo aver elaborato il comando DATA, l'host mittente inizia a elaborare il comando MAIL per inviare i dati alla destinazione. Quando un host accetta i dati per l'inoltro o per la consegna alla mailbox di destinazione, il protocollo genera un timestamp che indica quando i dati sono stati consegnati all'host e lo inserisce all'inizio dei dati. Quando i dati raggiungono l'host contenente la mailbox di destinazione, il percorso inverso specificato nel comando MAIL viene inserito all'inizio dei dati. Il

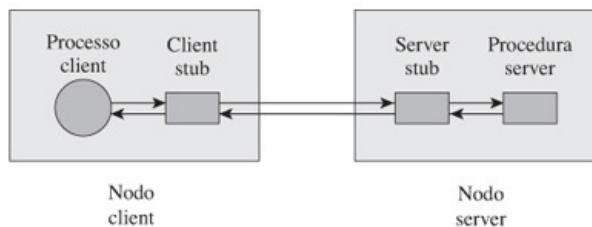
protocollo fornisce altri comandi per consegnare una mail al terminale di un utente, al terminale e alla mailbox dell'utente e al terminale o alla mailbox dell'utente. SMTP non fornisce una funzionalità di mailbox in ricezione, quindi viene tipicamente utilizzato con il protocollo Internet Message Access Protocol (IMAP) oppure con il protocollo Post Office Protocol (POP); questi protocolli consentono agli utenti di salvare i messaggi nelle mailbox.

### 9.4.2 Remote Procedure Call (RPC)

Le porzioni di un *programma distribuito* sono eseguite su computer differenti. La *remote procedure call* (chiamata a procedura remota o RPC) è una caratteristica dei linguaggi di programmazione utilizzata per richiamare queste porzioni. La semantica somiglia a quella di una chiamata a procedura convenzionale. La sintassi tipica è *call <proc\_id> (<messaggio>)*; dove *<proc\_id>* è l'id di una procedura remota e *<messaggio>* è una lista di parametri. La chiamata consiste nell'invio di *<messaggio>* alla procedura remota *<proc\_id>*. Il risultato della chiamata consiste nella risposta restituita dalla procedura *<proc\_id>*. L'implementazione di una RPC prevede l'utilizzo di un protocollo bloccante. Possiamo vedere la relazione chiamante-chiamato come una relazione client-server. In questo modo, la procedura remota è il server e un processo che la chiama è un client. Chiameremo i sistemi sui quali sono in esecuzione i processi client e server, rispettivamente, *nodo client* e *nodo server*.

I parametri possono essere passati per valore o per riferimento. Se l'architettura del nodo server è differente da quella del nodo client, il meccanismo RPC esegue l'appropriata conversione dei valori dei parametri. Per i parametri passati per riferimento, il chiamante deve costruire funzionalità a livello di sistema per i parametri ([Capitolo 15](#)). Queste funzionalità vengono trasmesse alla procedura remota nel messaggio. Possono essere eseguiti controlli sui parametri al momento della compilazione se il chiamante e il chiamato sono integrati durante la compilazione; altrimenti, i controlli sui tipi devono essere eseguiti dinamicamente quando viene effettuata una chiamata a procedura remota.

Il diagramma di [Figura 9.9](#) mostra l'organizzazione utilizzata per implementare una chiamata a procedura remota. La procedura server è la procedura remota che deve essere invocata. Il processo client chiama la procedura *client stub*, presente sullo stesso nodo client. Il client stub assembla i parametri, cioè colleziona i parametri, li converte in un formato indipendente dall'architettura, e prepara un messaggio contenente questa rappresentazione dei parametri. A questo punto chiama il *server stub*, presente sul nodo server che contiene la procedura remota. Il server stub converte i parametri in una forma specifica per l'architettura e invoca la procedura remota. I risultati della chiamata a procedura sono restituiti al processo client attraverso il server stub e il client stub. I dettagli riguardanti i nomi della procedura remota e l'affidabilità della chiamata a procedura remota sono discussi successivamente nel [Capitolo 16](#).



**Figura 9.9** Panoramica di una chiamata a procedura remota (RPC).

Due sono gli standard per le RPC che si sono affermati e sono ampiamente utilizzati: SunRPC e SOF/DCE. Il loro utilizzo semplifica l'implementazione delle RPC, e rende i programmi che utilizzano le RPC portabili tra i sistemi e i rispettivi sistemi operativi. Questi standard specificano una *rappresentazione esterna* dei dati per il passaggio dei parametri e dei risultati tra client e server, e un'interfaccia per il compilatore che gestisce l'operazione di assemblaggio dei parametri.

La caratteristica *remote method invocation* (invocazione di un metodo remoto - RMI) di Java è un'implementazione della chiamata a procedura remota integrata nel linguaggio

Java. Il metodo remoto da invocare diventa un metodo di qualche oggetto. I parametri che sono oggetti locali sono passati per valore, mentre gli oggetti non locali sono passati per riferimento. L'integrazione con il linguaggio Java semplifica l'assegnazione dei nomi del metodo remoto e l'affidabilità del passaggio dei parametri e dei risultati tra client e server.

### 9.4.3 Standard del message passing per la programmazione parallela

Un *programma parallelo* si compone di un insieme di task che possono essere eseguiti in parallelo. Tali programmi possono essere eseguiti su un insieme eterogeneo di computer o su un *massively parallel processor* (MPP). I programmi paralleli utilizzano le librerie per lo scambio di messaggi che consentono alle attività parallele di comunicare attraverso i messaggi. La *macchina virtuale parallela* (PVM) e l'interfaccia per lo scambio di messaggi - *message passing interface* (MPI) sono due standard usati per la codifica delle librerie per lo scambio di messaggi. Entrambi gli standard forniscono le seguenti funzionalità:

- comunicazione punto-punto tra due processi, mediante l'uso delle tecniche a denominazione simmetrica e asimmetrica, e comunicazione collettiva tra processi, che include la possibilità di inviare un messaggio in broadcast a un insieme di processi;
- *sincronizzazione a barriera* tra un insieme di processi in cui un processo che richiama una funzione di sincronizzazione a barriera resta bloccato finché *tutti* i processi nell'insieme dei processi non hanno richiamato la funzione di sincronizzazione a barriera;
- operazioni globali per distribuire le porzioni disgiunte di dati in un messaggio a processi differenti, per raccogliere i dati provenienti da processi differenti ed eseguire operazioni globali di riduzione sui dati ricevuti.

Nello standard PVM, un insieme eterogeneo di computer collegati in rete funzionano come un macchina virtuale parallela, ovvero un unico grande computer parallelo. I singoli sistemi possono essere workstation, sistemi multiprocessori o supercomputer vettoriali. Quindi il message passing deve risolvere la problematica della rappresentazione eterogenea dei dati sui diversi computer che costituiscono la macchina virtuale parallela. Alla ricezione di un messaggio, può essere eseguita una sequenza di chiamate alle routine di libreria che spacchettano e convertono i dati in una forma adatta all'elaborazione da parte del processo ricevente. PVM fornisce anche segnali che possono essere utilizzati per notificare specifici eventi.

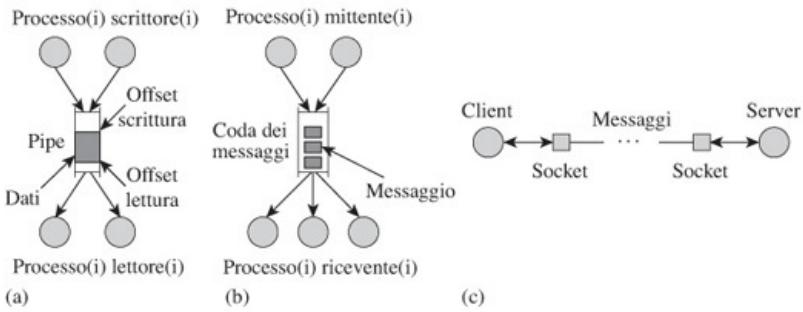
MPI è uno standard per processori altamente paralleli. Fornisce una send non bloccante, implementata come segue: il messaggio da inviare viene copiato in un buffer, in modo che il processo che ha invocato la send possa continuare l'esecuzione. Tuttavia, il processo non deve riutilizzare il buffer prima che sia completata la send precedente sul buffer. A tal fine, un *request handle* è associato a ogni send non bloccante. Inoltre, vengono fornite alcune chiamate di libreria per controllare il completamento di un'operazione di send esaminando il relativo request handle e per bloccare un processo finché una specifica operazione di send, o una di più operazioni send, non sia completata.

## 9.5 Casi di studio

### 9.5.1 Message passing in Unix

Unix supporta tre funzioni di comunicazione tra processi denominate *pipe*, coda dei messaggi (*message queue*) e *socket*. Una pipe è una funzione per il trasferimento dei dati, mentre le code di messaggi e i socket sono usati per lo scambio di messaggi. Queste funzioni hanno una caratteristica comune, cioè i processi possono comunicare senza conoscere le identità degli altri. Queste tre funzioni differiscono in dipendenza dell'applicazione. Le unnamed pipe (pipe senza nome) possono essere usate solo da processi che appartengono allo stesso albero dei processi, mentre la named pipe (pipe con nome) possono essere usate anche da altri processi. Le code di messaggi possono essere usate solo dai processi presenti nell'"Unix system domain", ovvero nel dominio di Unix in esecuzione su un computer. I socket possono essere usati dai processi nel dominio Unix e da quelli presenti in alcuni domini Internet. La **Figura 9.10** illustra i

concetti di pipe, message queue e socket.



**Figura 9.10** Comunicazione interprocesso in Unix: (a) pipe; (b) coda dei messaggi; (c) socket.

### Pipe

Una pipe è un meccanismo first-in, first-out (FIFO) per il trasferimento dei dati tra processi lettori e scrittori. Una pipe è implementata nel file system in molte versioni di Unix; tuttavia, differisce da un file in un aspetto importante, in quanto i dati inseriti in una pipe possono essere letti solo una volta. Unix fornisce due tipi di pipe: pipe con nome e senza nome (named e unnamed). Entrambi i tipi di pipe sono creati mediante la chiamata di sistema *pipe*. Le semantiche sono identiche fatta eccezione per la seguente differenza: una pipe con nome corrisponde a un elemento all'interno di una directory e può quindi essere utilizzata da qualsiasi processo, soggetta ai permessi sui file, mediante la system call *open*. Viene mantenuta nel sistema finché non è rimossa attraverso la system call *unlink*. Una pipe senza nome non corrisponde ad alcun elemento all'interno di una directory; può essere usata solo dal suo creatore e dai suoi discendenti nell'albero dei processi. Il kernel cancella una pipe senza nome quando non esistono più lettori o scrittori.

Una pipe è implementata come un file, fatta eccezione per due differenze (riferirsi al Paragrafo 13.14.1 per una discussione sull'implementazione dei file in Unix). La dimensione di una pipe è limitata in modo tale che i dati in una pipe siano memorizzati nei blocchi a indirizzamento diretto dell'inode. Il kernel tratta una pipe come una coda mantenendo due offset: un offset è usato per scrivere i dati nella pipe e l'altro per leggere i dati dalla pipe [Figura 9.10(a)]. Gli offset di lettura e scrittura sono mantenuti nell'inode invece che nella struttura file. Questa organizzazione fa in modo che un processo non possa modificare gli offset in alcun modo, se non mediante la lettura o la scrittura dei dati. I dati scritti vengono inseriti nella pipe utilizzando l'offset di scrittura che viene incrementato del numero di byte scritti. I dati scritti da più scrittori sono mischiati se vengono scritti in maniera interfogliata. Se una pipe è piena, un processo che vuole scrivere dati al suo interno viene posto in sleep. Un'operazione di lettura viene eseguita utilizzando l'offset di lettura che viene incrementato del numero di byte letti. Un processo che legge dati da una pipe vuota viene posto in sleep.

### Coda dei messaggi

Una message queue (coda di messaggi) in Unix è analoga a una mailbox. Viene creata e posseduta da un processo. Altri processi possono inviare o ricevere messaggi verso oppure da una coda in accordo ai permessi di accesso specificati dal creatore della coda [Figura 9.10(b)]. Questi permessi sono specificati utilizzando le stesse convenzioni dei permessi dei file in Unix (Paragrafo 15.6.3). La dimensione di una coda di messaggi, in termini di numero di byte che può contenere, è specificata al momento della creazione.

Una coda di messaggi è creata mediante la system call *msgget* (*key, flag*) dove *key* specifica il nome della coda dei messaggi e *flag* indica alcune opzioni. Il kernel mantiene un array di code di messaggi e le rispettive chiavi. La posizione di una coda di messaggi in questo array è utilizzata come identificativo della coda; viene restituita dalla chiamata *msgget*, e il processo che esegue la chiamata la usa per inviare e ricevere messaggi. La problematica dei nomi è affrontata come segue: se un processo effettua una chiamata *msgget* con una chiave che corrisponde al nome di una coda di messaggi esistente, il kernel restituisce l'identificativo della coda. In questo modo, una coda può essere usata

da qualsiasi processo nel sistema. Se la chiave in una chiamata *msgget* non corrisponde al nome di una coda di messaggi esistente, il kernel crea una nuova coda, imposta la chiave come suo nome e restituisce l'identificativo della coda. Il processo che effettua la chiamata diventa il proprietario della coda.

Ogni messaggio consiste di un tipo di messaggio, espresso come intero e testo del messaggio. Il kernel copia ogni messaggio in un buffer e crea un header in cui sono indicati la dimensione del messaggio, il tipo e un puntatore all'area di memoria in cui è memorizzato il testo. Inoltre, mantiene una lista di header per ogni coda di messaggi per rappresentare i messaggi inviati ma non ancora ricevuti.

I messaggi sono inviati e ricevuti utilizzando le seguenti chiamate di sistema:

```
msgsnd (msgqid, msg_struct_ptr, count, flag)
msgrcv (msgqid, msg_struct_ptr, maxcount, type, flag)
```

I parametri *count* e *flag* di una chiamata *msgsnd* specificano il numero di byte in un messaggio e le azioni da intraprendere se c'è spazio sufficiente nella coda; per esempio, se bloccare il mittente o restituire un codice di errore. *msg\_struct\_ptr* è l'indirizzo di una struttura che contiene il tipo di un messaggio, un intero e il testo del messaggio, *maxcount* è la lunghezza massima del messaggio e *type* indica il tipo del messaggio da ricevere.

Quando un processo effettua una chiamata *msgrcv*, il parametro *type*, un intero, indica il tipo di messaggio che si vuole ricevere. Quando il parametro *type* ha valore positivo, la chiamata restituisce il primo messaggio nella coda con un tipo corrispondente. Se il valore di *type* è negativo, restituisce il messaggio con il numero più piccolo il cui tipo è più piccolo del valore assoluto di *type*. Se il valore di *type* è zero, restituisce il primo messaggio nella coda, senza considerare il tipo. Il processo viene bloccato se la coda non contiene alcun messaggio da consegnare.

Quando un processo effettua una chiamata *msgsnd*, viene bloccato se la coda dei messaggi non ha spazio a sufficienza per contenere il messaggio. Il kernel lo riattiva quando qualche processo riceve un messaggio dalla coda, e il processo ripete il controllo per verificare se il messaggio può essere inserito nella coda. Se il controllo fallisce, il processo viene bloccato di nuovo. Quando a un certo punto inserisce il messaggio nella coda dei messaggi, il kernel attiva tutti i processi bloccati su una receive sulla coda. Quando ognuno di questi processi viene schedulato, controlla se è presente nella coda un messaggio del tipo desiderato. Se il controllo fallisce, viene bloccato nuovamente.

L'Esempio 9.2 mostra come queste caratteristiche possono essere usate per implementare il server per le prenotazioni dell'Esempio 9.1.

### Esempio 9.2 - Coda dei messaggi in Unix

La Figura 9.11 mostra il server per le prenotazioni implementato utilizzando le system call di Unix 5.4. Ai messaggi di cancellazione, prenotazione e richiesta sono assegnati i tipi, rispettivamente, 1, 2 e 3. La chiamata *msgrcv* con *type* = -4 e *flag* = "no wait" restituiscono un messaggio di cancellazione, se presente. Se non sono presenti messaggi di cancellazione, restituisce un messaggio di prenotazione, se presente, oppure un messaggio di richiesta. Questa organizzazione comporta l'elaborazione delle cancellazioni prima delle prenotazioni e delle prenotazioni prima delle richieste, come desiderato. Inoltre non sono necessarie tre mailbox come in Figura 9.8.

```
server_prenotazioni()
{
    msgqid = msgget (prenotazione, flags);
    ...
    repeat
        msgrcv (msgqid, &msg_struct, 200, -4, "no wait");
        if ...      /* esiste un messaggio */
        then ...    /* elabora il messaggio */
    while(true);
}
```

**Figura 9.11** Un server per le prenotazioni in Unix 5.4.

Un socket è semplicemente un terminale di un canale di comunicazione. I socket possono essere utilizzati per la comunicazione tra processi all'interno del dominio Unix e nel dominio Internet; limiteremo la discussione al dominio Unix. Un canale di comunicazione tra un client e un server viene impostato come segue: i processi client e server creano ciascuna un socket. Questi due socket sono connessi per creare un canale di comunicazione al fine di inviare e ricevere messaggi [Figura 9.10(c)]. Il server può impostare canali di comunicazione con più client simultaneamente.

Il problema dei nomi viene affrontato come segue: il server collega il suo socket a un indirizzo valido nel dominio in cui il socket sarà utilizzato. L'indirizzo a questo punto viene reso noto nel dominio. Un processo client utilizza l'indirizzo per eseguire una *connect* tra il proprio socket e quello del server. Questo metodo evita l'uso degli identificativi dei processi nella comunicazione; è piuttosto un'istanza della tecnica a denominazione indiretta (Paragrafo 9.1.1).

Un server crea un socket *s* mediante la chiamata di sistema:

```
s = socket (domain, type, protocol)
```

dove *type* e *protocol* non sono rilevanti nel dominio Unix. La chiamata *socket* restituisce un identificatore di socket al processo. Il processo server effettua una chiamata *bind* (*s*, *addr*, ...), dove *s* è l'identificativo del socket restituito dalla chiamata *socket* e *addr* è l'indirizzo per il socket. Questa chiamata collega il socket all'indirizzo *addr*; *addr* a questo punto diventa il 'nome' del socket, che viene reso noto nel dominio per essere usato dai client. Il server esegue la chiamata di sistema *listen* (*s*, ...) per indicare che è disponibile ad accettare delle connessioni al proprio socket *s*.

Un client crea un socket mediante la chiamata *socket*, per esempio, *cs = socket (...)*, e tenta di connetterlo al socket di un server utilizzando la chiamata di sistema:

```
connect (cs, server_socket_addr, server_socket_addrlen)
```

Il server viene attivato quando un client cerca di connettersi al proprio socket. A questo punto effettua la chiamata *new\_soc = accept* (*s*, *client\_addr*, *client\_addrlen*). Il kernel crea un nuovo socket, lo connette al socket del client e restituisce l'identificativo di questo nuovo socket. Il server utilizza questo socket per implementare la comunicazione client-server. Il socket utilizzato dal server nella chiamata *listen* serve solo per impostare le connessioni. Tipicamente, dopo la chiamata *connect* il server crea un nuovo processo per gestire la nuova connessione. Questo metodo consente al socket creato dal server di accettare e gestire in modo concorrente nuove connessioni mediante le chiamate *listen* e *connect*. La comunicazione tra un client e un server è implementata mediante le chiamate *read* e *write* o *send* e *receive*. Una chiamata *send* assume il seguente formato:

```
count = send (s, message, message_length, flags)
```

e restituisce il numero di byte inviati. Un connessione via socket viene chiusa utilizzando la chiamata *close* (*s*) o *shutdown* (*s*, *mode*).

### 9.5.2 Message passing in Windows

Windows fornisce diverse funzioni per lo scambio di messaggi sicuro nell'ambito di un host e all'interno di un dominio Windows, che consiste di un gruppo di host. Una pipe con nome (*pipe named*) è utilizzata per la comunicazione bidirezionale affidabile in modalità byte o messaggio tra un server e i suoi client. È implementata attraverso l'interfaccia del file system e supporta sia lo scambio di messaggi sincrono che asincrono. Il nome della pipe segue la convenzione per l'attribuzione dei nomi universale (universal naming convention - UNC) di Windows, che garantisce la presenza di nomi unici all'interno di una rete Windows. La prima chiamata *createnamedpipe* per una pipe con nome è eseguita da un server, che ne specifica il nome, un descrittore di sicurezza, e il numero di connessioni simultanee che deve garantire. Il kernel annota queste informazioni e crea una connessione alla pipe. Il server effettua una chiamata *connectnamedpipe*, che lo blocca finché un client non si connette alla pipe. Un client si connette a una pipe tramite la funzione *createfile* o *callnamedpipe* con il nome della pipe come argomento. La chiamata ha successo se il tipo di accesso richiesto è conforme al descrittore di sicurezza della pipe. A questo punto il client può usare le funzioni *readfile* e *writefile* per accedere alla pipe. Il server può eseguire ulteriori chiamate a *createnamedpipe* per creare nuove connessioni alla pipe. Windows fornisce una *mailslot* per la comunicazione unidirezionale

non affidabile. Può essere utilizzata sia per lo scambio di messaggi punto a punto che per l'invio in broadcast di brevi messaggi in un dominio Windows.

### **Chiamata a procedura locale - Local Procedure Call (LPC)**

La funzionalità LPC esegue lo scambio di messaggi tra processi presenti nello stesso host. Viene utilizzata dalle componenti di Windows per alcuni scopi come l'invocazione del security authentication server, e dai processi nelle elaborazioni degli utenti per comunicare con i processi del sottosistema di ambiente. Inoltre viene utilizzata nella chiamata a procedura remota quando i processi mittente e destinatario sono presenti nello stesso host.

LPC fornisce una scelta fra tre metodi di message passing che si adattano ai messaggi piccoli e grandi, e ai messaggi speciali utilizzati dalla GUI Win32. I primi due tipi di LPC utilizzano gli oggetti porta (*port*) per implementare lo scambio di messaggi. Ogni oggetto port è come una mailbox. Contiene un insieme di messaggi in una struttura dati chiamata coda dei messaggi (*message queue*). Per impostare una comunicazione con i client, un server crea un oggetto port, pubblica il suo nome nell'host, e attende le richieste di connessione dai client. Il server viene attivato quando un client invia una richiesta di connessione all'oggetto port e fornisce al client un riferimento (*handle*) per l'oggetto. Il client usa questo riferimento per inviare un messaggio. Il server può comunicare con molti client mediante lo stesso oggetto port. Per i messaggi brevi, la coda dei messaggi contiene il testo del messaggio. Come discusso nel Paragrafo 9.1.2, questi messaggi vengono copiati due volte durante lo scambio dei messaggi. Quando un processo invia un messaggio, questo viene copiato nella coda dei messaggi dell'oggetto port. Da qui, viene copiato nello spazio di indirizzamento del destinatario. Per controllare il sovraccarico dello scambio di messaggi, la lunghezza di un messaggio viene limitata a 256 byte.

Il secondo metodo di message passing è usato per messaggi grandi. I processi client e server associano un oggetto sezione (*section*) nei propri spazi di indirizzamento. Quando il client vuole inviare un messaggio, scrive il testo del messaggio nell'oggetto section e invia un breve messaggio contenente il suo indirizzo e la dimensione nell'oggetto port. Alla ricezione di questo messaggio, il server vede il testo del messaggio nell'oggetto section. In questo modo, il messaggio viene copiato solo una volta.

Il terzo metodo di LPC è chiamato *LPC veloce*. Utilizza un oggetto section per passare i messaggi e un oggetto *event pair* per eseguire la sincronizzazione tra i processi client e server. Il server crea un oggetto event pair per ogni client, che consiste di due oggetti event. Inoltre crea un thread per ogni client, il cui scopo è esclusivamente quello di gestire le richieste effettuate dai client. Lo scambio di messaggi avviene in questo modo: il processo client inserisce un messaggio nell'oggetto section, invia un segnale all'oggetto event su cui il thread server è in attesa e si mette in attesa sull'altro oggetto event della coppia. Il thread server elabora il messaggio, invia un segnale all'oggetto event su cui il client è in attesa, e si mette in attesa sull'altro oggetto event. Per facilitare lo scambio di messaggi, il kernel fornisce una funzione che segnala automaticamente un oggetto event della coppia ed effettua una wait sull'altro oggetto event.

### **Socket e chiamata a procedura remota**

I *socket* di Windows furono originariamente modellati sui socket Unix BSD ma successivamente furono incluse molte estensioni. Le loro caratteristiche e implementazione sono analoghe a quelle dei socket Unix descritti nel Paragrafo 9.5. Winsock è integrato con il message passing di Windows. Quindi un programma può effettuare un'operazione asincrona su socket e ricevere una notifica di completamento dell'operazione mediante un messaggio di callback di Windows.

La funzionalità RPC di Windows è compatibile con lo standard SOF/DCE. È implementata utilizzando LPC se la procedura richiamata si trova sullo stesso host del client; altrimenti, viene implementata seguendo le linee discusse nel Paragrafo 9.4.2. È supportata anche una *RPC asincrona*, in cui la procedura remota opera concorrentemente con il suo client e, al suo completamento, viene inviata al client una notifica secondo quanto specificato nella chiamata: mediante un oggetto evento di sincronizzazione, mediante una chiamata a procedura asincrona, mediante una porta di I/O o mediante un'informazione di stato, che il client può esaminare.

## **Riepilogo**

Il paradigma del message passing realizza lo scambio di informazioni tra processi senza utilizzare la memoria condivisa. Questa caratteristica lo rende utile in diverse situazioni come la comunicazione tra le funzioni di un SO basato su microkernel, nell'elaborazione client-server, nei protocolli di comunicazione di alto livello e nella comunicazione tra task nei programmi paralleli o distribuiti. In questo capitolo, abbiamo studiato le funzionalità del message passing nei linguaggi di programmazione e nei sistemi operativi.

Le problematiche chiave nello scambio di messaggi riguardano la denominazione dei processi mittente e destinatario nelle chiamate *send* e *receive*, e la consegna dei messaggi. Nella tecnica a denominazione simmetrica, i processi mittente e destinatario specificano i rispettivi nomi nelle chiamate *send* e *receive*. Questo consente a un processo di gestire simultaneamente più conversazioni. Nella tecnica a denominazione asimmetrica, il processo destinatario non specifica un nome nella chiamata *receive*; il kernel considera i messaggi inviatigli da tutti i processi per la consegna. In quella a denominazione indiretta, i processi mittente e destinatario specificano il nome di una *mailbox*, piuttosto che i nomi, rispettivamente, del destinatario e del mittente. Questo consente al mittente e al destinatario di mantenere diverse conversazioni indipendenti attraverso diverse *mailbox*. Una *mailbox* contiene un insieme di buffer in cui i messaggi possono essere memorizzati per la consegna. Quando sono utilizzate le *mailbox*, il kernel utilizza i propri buffer per memorizzare i messaggi non consegnati.

Lo scambio di messaggi viene adottato nei protocolli di alto livello come il *simple mail transfer protocol* (SMTP), la *remote procedure call* (RPC), e negli standard *parallel virtual machine* (PVM) e *message passing interface* (MPI) per la programmazione parallela. Pertanto, i sistemi operativi forniscono molte funzionalità per lo scambio di messaggi da usare in diverse situazioni.

## Domande

- 9.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. Quando un processo invia un messaggio utilizzando una chiamata *send* bloccante, il kernel deve copiare il messaggio in un buffer.
  - b. Quando viene utilizzata una *send* non bloccante, un messaggio deve essere copiato due volte prima che il processo destinatario possa esaminarlo.
  - c. Nella tecnica a denominazione simmetrica, un processo che è stato bloccato su una chiamata *receive* sarà attivato se un qualsiasi processo gli invia un messaggio.
  - d. Quando si usa la tecnica a denominazione indiretta, un processo che invia un messaggio non deve conoscere l'identità del processo al quale il messaggio sarà consegnato.
- 9.2. Indicare la risposta esatta per ognuna delle seguenti domande.
  - a. Se un SO ha  $n$  processi e utilizza le chiamate *send* bloccanti e le chiamate *receive* asimmetriche,
    - i. Il SO può richiedere fino a  $n - 1$  buffer per ognuno degli  $n$  processi alla volta.
    - ii. Il SO può richiedere in ogni istante fino a  $\frac{n}{2} \times \frac{n}{2}$  buffer.
    - iii. Il SO può richiedere fino a  $n$  buffer in ogni momento.
    - iv. Nessuna delle precedenti.
  - b. Rispondere alla domanda 9.2(a) nel caso in cui i processi usano le *send* bloccanti e le chiamate *receive* simmetriche.

## Problemi

- 9.1. In [Figura 9.6](#) un processo può essere bloccato a causa della mancanza di memoria necessaria per creare un IMCB o un ECB. Si spieghi come queste condizioni possono essere gestite.
- 9.2. Modificare lo schema di [Figura 9.6](#) per implementare lo scambio di messaggi con la tecnica a denominazione simmetrica e le *send* bloccanti.
- 9.3. Il sistema di prenotazioni dell'Esempio 9.1 utilizza flag in una chiamata *receive* per controllare la presenza di un messaggio pendente. Un'ipotetica funzione di *mailbox*

non supporta i flag. Quindi un processo utilizza il seguente approccio per ottenere un effetto equivalente: quando un processo vuole controllare la presenza di messaggi in una mailbox, invia uno speciale messaggio con il testo "controllo messaggi" alla mailbox, ed esegue un *receive* dalla mailbox. Se il suo messaggio speciale viene consegnato, conclude che non ci sono altri messaggi nella mailbox. Riscrivere il sistema di prenotazioni utilizzando questo approccio. (*Suggerimento*: attenzione ai vecchi messaggi speciali!)

- 9.4. Modificare lo schema di [Figura 9.6](#) per implementare le code dei messaggi di Unix.
- 9.5. Viene proposto di introdurre una funzione di time-out nel message passing, tale che un processo che effettua una *receive* specifica la quantità di tempo che un messaggio può attendere. Alla scadenza di questo lasso di tempo, si verifica un time-out e il processo viene attivato. Progettare un'implementazione di questa funzione utilizzando il meccanismo della gestione degli eventi.
- 9.6. I processi in un SO utilizzano lo scambio di messaggi asimmetrico e asincrono. Il kernel riserva una quantità di memoria limitata da usare come buffer e non usa spazio su disco a questo scopo. Analizzare il sistema per eventuali deadlock ([Capitolo 8](#)). Il kernel come dovrebbe rilevare questi deadlock?
- 9.7. Progettare la send asincrona dello standard *message passing interface* (MPI) descritto nel Paragrafo 9.4.3.

## Note bibliografiche

La comunicazione tra processi nel sistema RC4000 è descritta in Brinch Hansen (1970). Accetta et al. (1986) discutono lo schema utilizzato in Mach. Bach (1986), McKusick et al. (1996), Vahalia (1996), e Stevens e Rago (2005) descrivono il message passing in Unix. Bovet e Cesati (2005) approfondiscono il message passing in Linux, mentre Russinovich e Solomon (2005) affrontano quello in Windows.

Geist et al. (1996) descrivono e confrontano gli standard per il message passing PVM e MPI per la programmazione parallela.

1. Accetta, M., R. Baron, W. Bolosky, D.B. Golub, R. Rashid, A. Tevanian, and M. Young (1986): "Mach: A new kernel foundation for Unix development," *Proceedings of the Summer 1986 USENIX Conference*, June 1986, 93-112.
2. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
3. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, Calif.
4. Brinch Hansen, P. (1970): "The nucleus of a multiprogramming system," *Communications of the ACM*, **13** (4), 238-241, 250.
5. Geist, G., J.A. Kohl, and P. M. Papadopoulos (1996): "PVM and MPI: a comparison of features," *Calculateurs Paralleles*, **8** (2).
6. McKusick, M.K., K. Bostic, M.J. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4 BSD Operating System*, Addison Wesley, Reading, Mass.
7. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
8. Stevens, W.R., and S.A. Rago (2005): *Advanced Programming in the Unix Environment*, 2nd ed., Addison Wesley Professional, Reading, Mass.
9. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
10. Vahalia, U. (1996): *Unix Internals - The New frontiers*, Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 10

## Sincronizzazione e scheduling nei sistemi operativi multiprocessore

---

### Obiettivi di apprendimento

- Classificazione delle architetture multiprocessore
- Problematiche nei sistemi multiprocessore
- Sincronizzazione e scheduling dei processi
- Sistemi operativi per architetture multiprocessore: Mach, Linux, SMP in Windows

Un sistema multiprocessore fornisce tre benefici: throughput elevato, aumento della velocità di elaborazione e il decadimento graduale (graceful degradation). Il *throughput elevato* può essere ottenuto utilizzando le CPU per eseguire molti processi simultaneamente. *L'aumento della velocità di elaborazione* di un'applicazione può essere ottenuto se molti dei suoi processi sono eseguiti dalle CPU allo stesso tempo. La *graceful degradation* è la caratteristica mediante la quale il sistema può continuare a funzionare seppur con un ridotto numero di CPU. In questo modo, il sistema può offrire continuità di funzionamento, sebbene con ridotte capacità.

Per realizzare i benefici di un sistema multiprocessore, il sistema operativo sfrutta la presenza di molte CPU mediante tre caratteristiche speciali: primo, un kernel *multiprocessore simmetrico - kernel SMP* in breve - consente a molte CPU di eseguire codice kernel in parallelo in modo che le funzioni di controllo del kernel non diventino un collo di bottiglia delle prestazioni. Secondo, speciali lock di sincronizzazione chiamati *spin lock* e *sleep lock* riducono i ritardi di sincronizzazione per i processi eseguiti in parallelo su diverse CPU. Terzo, le politiche di scheduling come lo *scheduling per affinità* e il *coscheduling* assicurano che i processi di un'applicazione possano essere eseguiti in maniera efficiente su diverse CPU.

Iniziamo con una panoramica delle architetture dei sistemi multiprocessore, che fornisce la base per la discussione delle tre caratteristiche dei SO descritti in precedenza.

### 10.1 Architettura dei sistemi multiprocessore

Le prestazioni di un sistema con singolo processore dipendono dalle prestazioni della CPU e della memoria, che possono essere migliorate mediante chip più veloci e con diversi livelli di cache. Tuttavia, la velocità dei chip non può essere incrementata oltre certi limiti tecnologici. Ulteriori miglioramenti delle prestazioni del sistema possono essere ottenuti solo utilizzando più CPU.

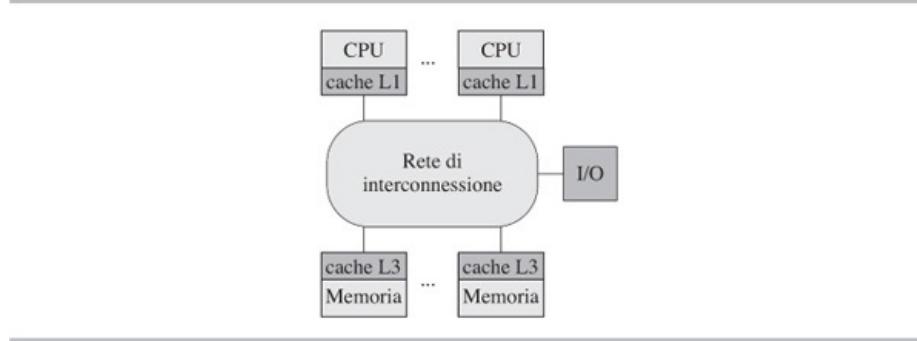
Come risultato della presenza di più CPU, le architetture multiprocessore possiedono la capacità di fornire i tre benefici riassunti nella [Tabella 10.1](#). Il *throughput elevato* è possibile perché il SO può schedulare diversi processi in parallelo e dunque diverse applicazioni possono progredire allo stesso tempo. L'aumento del throughput rispetto a un sistema con singolo processore può essere limitato dalla *contesa per la memoria* che si verifica quando più CPU tentano di accedere alla memoria allo stesso tempo, cosa che fa aumentare il tempo di accesso alla memoria per ogni processo. *L'aumento della velocità di elaborazione* si realizza quando i processi di un'applicazione sono schedulati in parallelo. Il fattore di aumento delle prestazioni può essere limitato dal grado di parallelismo all'interno dell'applicazione, ovvero, in che misura i processi di un'applicazione possono essere eseguiti senza richiedere frequenti sincronizzazioni. La *graceful degradation* fornisce la continuità di funzionamento anche in caso di malfunzionamento della CPU. Questa caratteristica è vitale per supportare applicazioni critiche come servizi online e applicazioni real-time.

| Beneficio                              | Descrizione  |
|--|--|
| Throughput elevato                     | Diversi processi possono essere eseguiti dalle CPU allo stesso tempo. Dunque viene eseguito più lavoro.  |
| Aumento della velocità di elaborazione | Diversi processi di un'applicazione possono essere eseguiti allo stesso tempo, con conseguente diminuzione della durata, ovvero del tempo di elaborazione, di una applicazione; fornisce migliori tempi di risposta. |
| Graceful degradation                   | Il malfunzionamento di una CPU non blocca il funzionamento del sistema; il sistema può continuare a funzionare con prestazioni ridotte.  |

**Tabella 10.1** Benefici dei multiprocessori.

### Un modello di sistema

La [Figura 10.1](#) mostra un modello di sistema multiprocessore. Le CPU, la memoria e il sottosistema di I/O sono connessi alla rete di interconessione. Ogni chip della CPU può contenere cache di livello 1 e di livello 2, ovvero le cache L1 e L2, che contengono blocchi di istruzioni e dati a cui si è fatto riferimento di recente da parte della CPU. Tuttavia, per semplicità, assumiamo che la CPU contenga solo una cache L1. La memoria si compone di diverse unità di memoria. Assumiamo che una cache L3 sia associata ad ogni unità di memoria e mantenga i blocchi di istruzioni e dati recentemente riferiti. Ogni volta che una CPU o un dispositivo di I/O vuole effettuare un accesso in memoria, la rete di interconnessione impone un percorso con l'unità di memoria che contiene il byte richiesto. Non tenendo conto dei ritardi della rete di interconnessione, il tempo effettivo di accesso alla memoria dipende dal hit ratio nelle cache L1, L2 ed L3 e dal tempo di accesso alla memoria (Paragrafo 2.2.3).



**Figura 10.1** Modello di sistema multiprocessore.

### Coerenza della cache e del TLB

Quando i processi usano dati condivisi, possono essere presenti nel sistema allo stesso momento diverse copie del dato  $d$ . Una di queste copie potrebbe essere nell'unità di memoria e una potrebbe trovarsi nella cache L3 associata all'unità di memoria, mentre la parte restante si troverebbe nella cache L1 delle CPU dove i processi vengono schedulati. Quando un processo in esecuzione su una CPU aggiorna una copia di  $d$ , le altre copie di  $d$  diventano obsolete. L'utilizzo da parte dei processi causerebbe problemi di correttezza e di consistenza dei dati, per cui il sistema utilizza un *protocollo di correttezza della cache* per assicurare che non venga mai utilizzata una copia obsoleta nell'elaborazione.

I protocolli di coerenza della cache sono basati su due approcci fondamentali, dei quali sono state sviluppate diverse varianti. L'approccio *snooping-based* può essere usato se la rete di interconnessione è un bus. Una CPU controlla il bus per rilevare i messaggi relativi al caching ed elimina le copie obsolete dalla propria cache L1. Nella variante *write-invalidate* di questo approccio, ogni processo che aggiorna una copia del dato condiviso  $d$  deve aggiornare la copia di  $d$  in memoria. Dunque la memoria non contiene mai una copia obsoleta. Una CPU che aggiorna  $d$  invia sul bus un messaggio "cache non valida" per  $d$ . Alla rilevazione di questo messaggio, ogni CPU che controlla il bus scarta la copia di  $d$ , se presente, dalla propria cache L1. La prossima volta che questa CPU

accede a  $d$ , viene copiato il nuovo valore nella cache L1 della CPU.

Un approccio per la coerenza della cache *directory-based* richiede di mantenere una directory di informazioni relative alle copie in cache dei dati nel sistema; la directory potrebbe indicare quale CPU contiene copie in cache di ogni dato. Nell'aggiornare un dato  $d$ , una CPU manda un segnale di invalidazione della cache a ognuna di queste CPU. Alternativamente, la directory potrebbe indicare il luogo della copia di ogni dato condiviso aggiornata più di recente. Quando una CPU  $C_1$  vuole accedere a un dato  $d$ , invia una richiesta "leggi  $d$ " alla directory. La directory invia a sua volta la richiesta all'unità di memoria o alla CPU che ha la copia più recente di  $d$  nella propria cache, la quale inoltra il valore di  $d$  a  $C_1$ . Dopo l'aggiornamento, il riferimento a  $d$  nella directory punta a  $C_1$ . La *coerenza del TLB* rappresenta un problema analogo, che si verifica quando l'informazione contenuta in qualche elemento del TLB di una CPU diventa obsoleta a causa del rimpiazzamento della pagina operato da altre CPU o a causa della modifica dei privilegi di accesso dei processi alle pagine condivise. Una pagina condivisa  $p_i$  di un processo ha elementi nei TLB di molte CPU. Se si verifica un fault di pagina in un processo in esecuzione su una delle CPU, per esempio la CPU  $C_1$ , e la pagina  $p_i$  è sostituita con una nuova pagina, l'elemento del TLB relativo a  $p_i$  in  $C_1$  viene cancellato (Paragrafo 12.2.2). Gli elementi del TLB relativi a  $p_i$  nelle altre CPU sono ora obsoleti, per cui devono essere cancellati. La cancellazione viene effettuata mediante un'azione di *TLB shootdown*, nella quale la CPU  $C_1$  invia degli interrupt interprocessore alle altre CPU con i dettagli dell'id di  $p_i$ , in modo che le altre CPU possano invalidare le copie di  $p_i$  nei propri TLB. Azioni simili sono effettuate quando vengono modificati i privilegi di accesso alle pagine condivise. L'overhead di uno shootdown del TLB è ridotto in due modi. L'elemento della tabella delle pagine relativo a  $p_i$  indica quali CPU hanno elementi di TLB relativi a  $p_i$ , in modo che  $C_1$  invii gli interrupt solo a queste CPU. Una CPU che riceve l'ordine di shootdown potrebbe implementarlo in modo *silenzioso*, ovvero secondo necessità. Se lo shootdown riguarda il processo attualmente in esecuzione, cancella immediatamente l'elemento del TLB; altrimenti accoda l'ordine e lo gestisce quando viene schedulato il processo cui si riferisce.

### **Classificazione dei sistemi multiprocessore**

I sistemi multiprocessore sono classificati in tre tipi di sistemi in base al modo in cui le CPU e le unità di memoria sono associate le une con le altre.

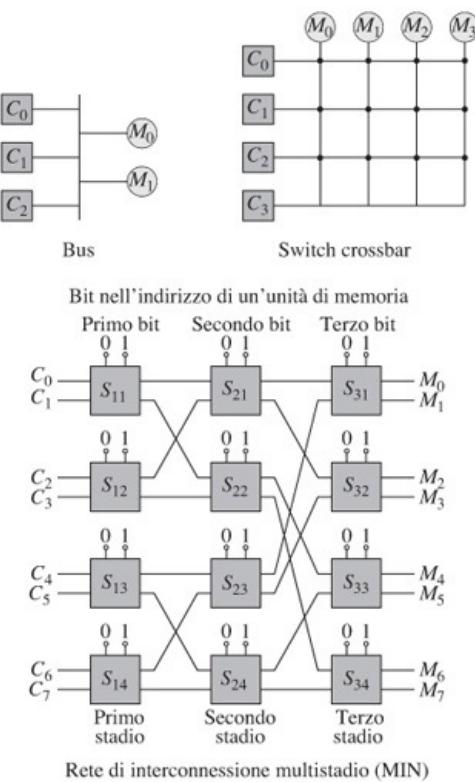
- **Architettura uniform memory access (architettura UMA):** tutte le CPU nel sistema possono accedere all'intera memoria in maniera identica, ovvero, con la stessa velocità di accesso. Alcuni esempi di architettura UMA sono il sistema Balance della Sequent e il VAX 8800 della Digital. L'architettura UMA viene chiamata, in letteratura, *architettura multiprocessore strettamente accoppiata*. Viene anche chiamata *architettura multiprocessore simmetrica (Symmetrical Multiprocessor - SMP)*.
- **Architettura nonuniform memory access (architettura NUMA):** il sistema si compone di un numero di *nodi* ed ogni nodo si compone di una o più CPU, un'unità di memoria e un sottosistema di I/O. L'unità di memoria di un nodo si dice *locale* alle CPU in quel nodo. Le altre unità di memoria vengono definite *non locali*. Tutte le unità di memoria insieme costituiscono un singolo spazio di indirizzamento. Ogni CPU può accedere all'intero spazio di indirizzamento; tuttavia, può accedere all'unità di memoria locale più velocemente rispetto all'accesso alle unità di memoria non locali. Alcuni esempi di architettura NUMA sono l'AlphaServer della HP e il NUMA-Q dell'IBM.
- **Architettura no-remote-memory-access (architettura NORMA):** ogni CPU ha la propria memoria locale. Le CPU possono accedere alle unità di memoria remote, ma questo accesso avviene tramite rete, per cui è molto lento se confrontato con l'accesso alla memoria locale. Il sistema Hypercube della Intel è un esempio di architettura NORMA. Un sistema NORMA è un sistema distribuito secondo la Definizione 3.8 e, per questo motivo, in questo capitolo non affronteremo i sistemi NORMA.

### **Reti di interconnessione**

Le CPU di un sistema multiprocessore accedono alle unità di memoria attraverso una rete di interconnessione. Due importanti attributi di una rete di interconnessione sono il costo e la velocità di accesso effettiva. La [Tabella 10.2](#) elenca le caratteristiche e i relativi vantaggi di tre famose reti di interconnessione. La [Figura 10.2](#) contiene uno schema di queste reti.

| Rete di interconnessione                 | Caratteristiche   |
|--|---|
| Bus                                      | Costo basso. Velocità di accesso ragionevole in condizione di bassa densità di traffico. Solo un collegamento CPU-memoria può essere attivo a ogni istante.   |
| Crossbar switch                          | Costo elevato. Scarsa espandibilità. Le CPU e le unità di memoria sono connesse allo switch. Un collegamento CPU-memoria viene implementato selezionando un percorso tra una CPU e un'unità di memoria. Consente molti collegamenti CPU-memoria in parallelo. |
| Multistage interconnection network (MIN) | Compromesso tra un bus e un crossbar switch. Si compone di molti stadi di switch crossbar $2 \times 2$ . Un collegamento CPU-memoria viene determinato selezionando un percorso attraverso ogni stadio. Consente alcuni collegamenti in parallelo.            |

**Tabella 10.2** Caratteristiche delle reti di interconnessione.



**Figura 10.2** Bus, crossbar switch e multistage interconnection network (MIN).

Un *bus* in un sistema multiprocessore è semplicemente un'estensione di un bus in un sistema con singolo processore. Tutte le unità di memoria e tutte le CPU sono connesse al bus. In questo modo il bus supporta il traffico dati tra ogni CPU e ogni unità di memoria. Tuttavia, solo un collegamento CPU-memoria può essere attivo alla volta. Il bus è semplice ed economico ma è lento a causa delle contese per il suo utilizzo quando più di una CPU vuole accedere alla memoria nello stesso momento. Il bus può diventare un collo di bottiglia quando aumenta il numero delle CPU.

Un *crossbar switch* riduce il problema della contesa fornendo molti percorsi per i collegamenti CPU-memoria. Utilizza una matrice di collegamenti in cui le CPU sono organizzate lungo una dimensione e le unità di memoria lungo l'altra dimensione (Figura 10.2). Ogni CPU e ogni unità di memoria ha il suo bus indipendente. Quando una CPU, per esempio la CPU  $C_1$ , vuole accedere a un byte situato in un'unità di memoria, per

esempio l'unità di memoria  $M_3$ , lo switch connette il bus di  $C_1$  con il bus di  $M_3$  e il collegamento CPU-memoria avviene lungo questo percorso. Questo collegamento non è soggetto a contese causate dai collegamenti tra le altre CPU e le altre unità di memoria poiché questi collegamenti utilizzerebbero percorsi differenti attraverso lo switch. In questo modo, lo switch può fornire un'ampia larghezza di banda per l'accesso alla memoria. La contesa si verificherebbe solo se due o più CPU volessero accedere alla stessa unità di memoria, evento che ha una bassa probabilità di occorrenza in condizioni di bassa densità di traffico tra le CPU e le unità di memoria. Tuttavia, un crossbar switch è costoso e, inoltre, soffre di scarsa espandibilità.

Un *multistage interconnection network* (MIN) è un compromesso tra un bus e un crossbar switch in termini di costo e parallelismo; è stato usato nel BBN Butterfly, che realizza un'architettura NUMA. La [Figura 10.2](#) mostra una rete di interconnessione Omega  $8 \times 8$ , che consente a 8 CPU di accedere a 8 unità di memoria con indirizzi binari da 000 a 111. Contiene tre stadi poiché le unità di memoria hanno indirizzi di 3 bit. Ogni colonna contiene gli switch crossbar  $2 \times 2$  di uno stadio della rete di interconnessione. Per ogni switch, una riga rappresenta una CPU e un colonna rappresenta il valore di un bit nell'indirizzo binario dell'unità di memoria cui accedere. Se il bit dell'indirizzo è 0, viene selezionato l'output superiore dello switch. Se il bit dell'indirizzo è 1, viene selezionato l'output inferiore dello switch. Questi output sono collegati agli switch dello stadio successivo.

Quando la CPU  $C_1$  vuole accedere all'unità di memoria  $M_4$ , la connessione avviene come segue: l'indirizzo dell'unità di memoria  $M_4$  è 100. Poiché il primo bit è 1, viene selezionato l'output inferiore dello switch  $S_{11}$ . Questo è collegato a  $S_{22}$ , il cui output superiore viene selezionato poiché il bit successivo dell'indirizzo è 0. Questo è collegato a  $S_{33}$ , il cui output superiore viene selezionato. L'ultimo collegamento porta a  $M_4$  come desiderato. Gli switch  $S_{13}$ ,  $S_{24}$  ed  $S_{34}$  sarebbero selezionati se la CPU  $C_4$  volesse accedere all'unità di memoria 7. La rete di interconnessione utilizza dodici switch  $2 \times 2$ . Il costo di questi switch è molto inferiore rispetto ai crossbar switch  $8 \times 8$ . In generale, una rete multistadio  $N \times N$  usa  $\log_2 N$  stadi e ogni stadio contiene  $(N/2)$  switch  $2 \times 2$ .

Altre reti di interconnessione usano combinazioni di queste tre reti di interconnessione fondamentali. Per esempio, la IEEE scalable coherent interface (SCI) utilizza una rete ad anello che fornisce servizi sul modello a bus, ma utilizza collegamenti veloci punto-punto unidirezionali per ottenere un elevato throughput. Un crossbar switch viene utilizzato per selezionare il link unidirezionale connesso a una CPU.

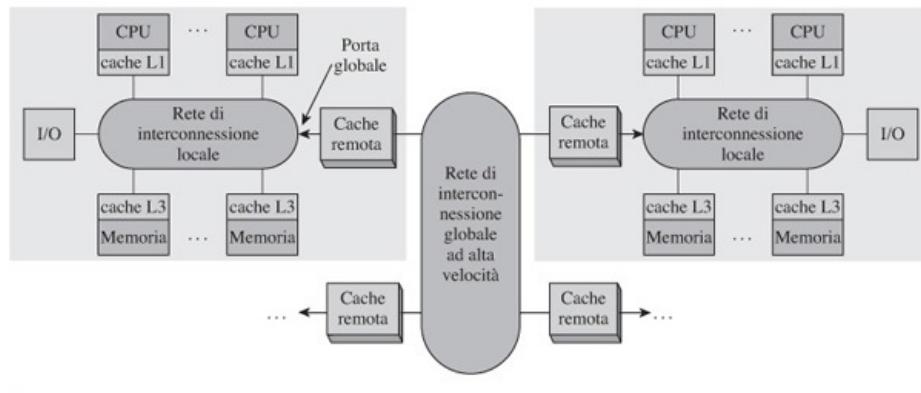
### 10.1.1 Architettura SMP

Le architetture SMP tipicamente utilizzano un bus o un crossbar switch come rete di interconnessione. Come discusso in precedenza, solo un collegamento può essere attivo sul bus in ogni istante; gli altri collegamenti vengono ritardati. Dunque le CPU sono soggette a ritardi imprevedibili durante l'accesso in memoria. Il bus può diventare un collo di bottiglia e limitare le prestazioni del sistema. Quando si utilizza un crossbar switch, le CPU e il sottosistema di I/O subiscono minori ritardi nell'accesso alla memoria, per cui le prestazioni del sistema sono migliori rispetto all'utilizzo del bus. I ritardi dovuti allo switch sono anche più prevedibili rispetto ai ritardi del bus. I protocolli di coerenza della cache si aggiungono ai ritardi per l'accesso alla memoria in entrambe queste varianti dell'architettura SMP. Dunque i sistemi SMP non scalano bene oltre un numero ridotto di CPU.

### 10.1.2 Architettura NUMA

La [Figura 10.3](#) illustra l'architettura di un sistema NUMA. Ogni contenitore tratteggiato racchiude un nodo del sistema. Un nodo potrebbe essere composto di un sistema con singola CPU; tuttavia, comunemente vengono utilizzati sistemi SMP come nodi. Dunque un nodo si compone di alcune CPU, le unità di memoria locale e un sottosistema di I/O connessi da una rete di interconnessione locale. Ogni rete di interconnessione locale ha anche una porta globale e le porte globali di tutti i nodi sono connesse a una rete di interconnessione globale ad alta velocità in grado di fornire tassi di trasferimento fino a 1 GB/s, ovvero 1 GB per secondo. Queste porte sono utilizzate per il traffico tra le CPU e le unità di memoria non locali. Una porta globale di un nodo può anche contenere una cache per memorizzare istruzioni e dati di memorie non locali cui le CPU del nodo hanno avuto accesso. La rete di interconnessione globale mostrata in [Figura 10.3](#) ricorda la

IEEE scalable coherent interface (SCI). Utilizza una rete ad anello che fornisce link punto-punto unidirezionali veloci tra i nodi.



**Figura 10.3** Architettura NUMA.

Come in un sistema SMP, l'hardware di un sistema NUMA deve assicurare la coerenza tra le cache delle CPU di un nodo. Inoltre deve assicurare la coerenza tra le cache non locali. Questa esigenza può rallentare gli accessi in memoria e consumare parte dell'ampiezza di banda delle reti di interconnessione. Ignorando i ritardi nelle reti di interconnessione locali e non locali, il tempo effettivo di accesso a una memoria locale dipende dall'hit ratio delle cache L1 e L3 e dal tempo di accesso alla memoria. Il tempo di accesso a una memoria non locale dipende dall'hit ratio della cache L1 e dalla cache remota della porta globale e dal tempo di accesso alla memoria.

I nodi di un sistema NUMA tipicamente sono sistemi SMP ad alte prestazioni che contengono 4 o 8 CPU. In conseguenza dell'alta velocità della rete di interconnessione non locale, le prestazioni di questa architettura NUMA sono scalabili all'aumentare del numero di nodi. Le prestazioni effettive di un sistema NUMA dipendono dagli accessi di memoria non locali effettuati dai processi durante la loro esecuzione. Questo è un problema del SO, di cui discuteremo nel prossimo paragrafo.

## 10.2 Problematiche nei sistemi operativi multiprocessore

Per realizzare i benefici di throughput elevato e aumento della velocità di elaborazione offerti da un sistema multiprocessore, le CPU devono essere utilizzate efficacemente e i processi di un'applicazione dovrebbero essere in grado di interagire in modo sinergico. Queste due considerazioni influenzano, naturalmente, lo scheduling dei processi e la sincronizzazione dei processi. Inoltre influenzano anche i metodi di funzionamento del sistema operativo in risposta agli interrupt e alle chiamate di sistema. La [Tabella 10.3](#) evidenzia le tre problematiche fondamentali sollevate da queste considerazioni.

| Problematica                  | Descrizione   |
|-------------------------------|---|
| Struttura del kernel          | Molte CPU dovrebbero essere in grado di eseguire il codice del kernel in parallelo, in modo che l'esecuzione delle funzioni del kernel non diventi un collo di bottiglia. |
| Sincronizzazione dei processi | La presenza di più CPU dovrebbe essere sfruttata per ridurre l'overhead dovuto alla commutazione dei processi e i ritardi di sincronizzazione.                            |
| Scheduling dei processi       | La politica di scheduling dovrebbe sfuggire la presenza di più CPU per determinare l'aumento della velocità di elaborazione per le applicazioni.                          |

**Tabella 10.3** Problematiche relative alla sincronizzazione e allo scheduling in un SO multiprocessore.

I primi sistemi operativi multiprocessore funzionavano in modalità *master-slave*. In questa modalità, una CPU è designata come master e tutte le altre CPU operano come

suoi slave. Solo la CPU master può eseguire il codice kernel. Gestisce gli interrupt e le system call ed esegue lo scheduling. Comunica le decisioni di schedulazione alle altre CPU mediante *interrupt interprocessore* (IPI). Il vantaggio principale della struttura del kernel master-slave è la sua semplicità. Quando un processo effettua una chiamata di sistema, la CPU sulla quale era in esecuzione resta idle finché il processo riprende la sua esecuzione o la CPU master assegna un nuovo lavoro alla CPU. Nessuna di queste situazioni si può verificare finché la CPU master gestisce la system call ed esegue lo scheduling. Dunque, l'esecuzione delle funzioni del kernel da parte del master è un collo di bottiglia che influisce sulle prestazioni del sistema. Questo problema può essere risolto strutturando il kernel in modo tale che molte CPU possano eseguire il suo codice in parallelo.

La presenza di più CPU può essere sfruttata per ridurre i ritardi di sincronizzazione. In un sistema con singolo processore, consentire a un processo di ciclare finché non si verifica una condizione di sincronizzazione nega la CPU agli altri processi e può portare all'inversione delle priorità (Paragrafo 6.5.1). Dunque, la sincronizzazione viene effettuata bloccando un processo finché la condizione di sincronizzazione non si è verificata. Tuttavia, in un sistema multiprocessore, la sincronizzazione mediante cicli non porta all'inversione delle priorità poiché il processo che detiene il lock può essere eseguito su un'altra CPU in parallelo con il processo che cicla. Tuttavia è preferibile lasciare ciclare il processo, piuttosto che bloccarlo, se la quantità di tempo trascorsa a ciclare fosse minore dell'overhead totale di CPU per bloccarlo, schedulare un altro processo, attivarlo e rischedularlo successivamente. Questa condizione sarebbe soddisfatta se un processo che cicla per entrare in una sezione critica e il detentore della sezione critica fossero schedulati in parallelo. I sistemi operativi multiprocessore mettono a disposizione speciali tecniche di sincronizzazione per sfruttare queste caratteristiche.

Lo scheduling dei processi è influenzato da due fattori: le prestazioni della cache durante l'esecuzione di un processo e i requisiti di sincronizzazione dei processi di un'applicazione. Schedulare un processo ogni volta sulla stessa CPU può portare a un elevato hit ratio della cache, che migliorerebbe le prestazioni del processo e inoltre contribuirebbe a migliorare le prestazioni del sistema. Se i processi di un'applicazione interagiscono frequentemente, schedularli allo stesso momento su differenti CPU fornirebbe loro un'opportunità di interazione in tempo reale, che porterebbe a un aumento della velocità dell'applicazione. Per esempio, un produttore e un consumatore in un sistema produttore-consumatore con singolo buffer possono essere in grado di eseguire diversi cicli di produzione e consumo di elementi in una time slice se i processi sono schedulati per l'esecuzione in parallelo.

In questo modo, la struttura del kernel e gli algoritmi utilizzati per la schedulazione e la sincronizzazione determinano insieme se un SO multiprocessore ottiene un elevato throughput. Tuttavia, i computer crescono di dimensione con le innovazioni tecnologiche o le richieste degli utenti, per cui un altro aspetto delle prestazioni, chiamato *scalabilità*, è ugualmente importante. La scalabilità di un sistema indica quanto si comporta bene il sistema al crescere della sua dimensione. La dimensione di un SO multiprocessore può crescere mediante l'aggiunta di più CPU, unità di memoria e altre risorse di sistema, o mediante la creazione di più processi nelle applicazioni. Quando un sistema cresce di dimensione si verificano due tipi di aspettative riguardanti le prestazioni: il throughput del sistema dovrebbe aumentare linearmente con il numero delle CPU e i ritardi dei singoli processi, dovuti alla sincronizzazione o allo scheduling, non dovrebbero aumentare all'aumentare del numero dei processi nel sistema.

La scalabilità è importante nella progettazione sia dell'hardware che del software. Le tecnologie di interconnessione che hanno buone prestazioni quando il sistema contiene un numero ridotto di CPU e di unità di memoria possono non ottenere prestazioni altrettanto soddisfacenti quando il loro numero aumenta. Per essere scalabile, l'ampiezza di banda effettiva di una rete di interconnessione dovrebbe aumentare linearmente con il numero delle CPU. Come discusso nel Paragrafo 10.1, il crossbar switch è più scalabile rispetto al bus come rete di interconnessione. Nel campo software, vengono utilizzate speciali tecniche per assicurare la scalabilità degli algoritmi. Discuteremo questo aspetto nei Paragrafi 10.4 e 10.5.

### 10.3 Struttura del kernel

Il kernel di un sistema operativo multiprocessore per un'architettura SMP viene chiamato *kernel SMP*. È strutturato in modo che ogni CPU possa eseguire il codice del

kernel in parallelo. Questa caratteristica si basa su due condizioni fondamentali: il codice del kernel SMP è *rientrante* (Paragrafo 11.3.3 per una discussione sul codice rientrante) e le CPU che lo eseguono in parallelo coordinano le loro attività mediante sincronizzazione e interrupt interprocessore.

### **Sincronizzazione**

Il kernel utilizza i semafori binari per assicurare la mutua esclusione sulle strutture dati del kernel (Paragrafo 6.9) – faremo riferimento a esse come *mutex lock*. Il lock si dice essere a *grana grossa* se un mutex lock controlla gli accessi a un gruppo di strutture dati e si dice essere a *grana fine* se un mutex lock controlla gli accessi a un singolo dato o a una singola struttura dati. Il lock a grana grossa fornisce semplicità; tuttavia, non si può accedere in parallelo a due o più strutture dati controllate dal lock, per cui l'esecuzione delle funzionalità del kernel può diventare un collo di bottiglia. Il lock a grana fine consente alle CPU di accedere in parallelo a differenti strutture dati. Tuttavia, il lock a grana fine può incrementare l'overhead del lock poiché una CPU che esegue codice del kernel dovrebbe impostare e rilasciare un maggior numero di lock. Inoltre, può causare deadlock se tutte le CPU non impostano i lock nello stesso ordine. Dunque, dovrebbero essere utilizzate delle politiche di prevenzione dei deadlock come la politica del ranking delle risorse (Paragrafo 8.8), cioè dovrebbe essere assegnato un punteggio a ogni lock in modo tale che una CPU potrebbe impostare i lock secondo l'ordine crescente di punteggio.

Buone prestazioni dei kernel SMP si ottengono assicurando il parallelismo senza incorrere nell'overhead dovuto al lock. Due sono i modi di operare:

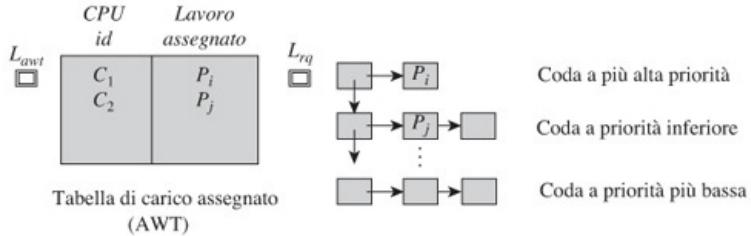
- *uso di lock separati per le funzionalità del kernel*: le CPU possono eseguire differenti funzionalità del kernel in parallelo senza incorrere in elevati overhead per il lock;
- *partizionamento delle strutture dati di una funzionalità del kernel*: le CPU possono eseguire la stessa funzione del kernel in parallelo impostando i lock per le diverse parti delle strutture dati. Il lock può essere evitato del tutto associando permanentemente una diversa parte ad ogni CPU.

### **Gestione dell'heap**

Il parallelismo nella gestione dell'heap può essere ottenuto mantenendo diverse *free list*, ovvero liste di aree di memoria libere nell'heap (Paragrafo 11.5.1). Il lock non è necessario se ogni CPU possiede la propria free list; tuttavia, questa organizzazione farebbe degradare le prestazioni poiché le decisioni relative all'allocazione non sarebbero ottimali. Creare free list separate per memorizzare le aree di memoria libere di dimensione differente e consentire a una CPU il lock di una free list appropriata fornirebbe il parallelismo tra le CPU che cercano di ottenere aree di memoria di dimensione differente. Inoltre, eviterebbe prestazioni subottimali causate dall'associazione permanente di una free list a una CPU.

### **Scheduling**

La [Figura 10.4](#) illustra alcune semplici strutture dati per lo scheduling utilizzate da un kernel SMP. Le CPU  $C_1$  e  $C_2$  sono impegnate, rispettivamente, nell'esecuzione dei processi  $P_i$  e  $P_j$ . Le code dei processi pronti sono organizzate come discusso nel Paragrafo 7.4.3: ogni coda dei processi pronti contiene i PCB dei processi *ready* con una specifica priorità. Il kernel mantiene una struttura dati addizionale chiamata *tabella del carico assegnato* (assigned workload table – AWT) nella quale memorizza il carico assegnato alle varie CPU. I mutex lock chiamati  $L_{rq}$  e  $L_{awt}$  controllano, rispettivamente, le strutture dati dei processi pronti e l'AWT. Assumiamo che le CPU impostino questi lock nell'ordine  $L_{rq}$  seguito da  $L_{awt}$ .



**Figura 10.4** Strutture dati per lo scheduling in un kernel SMP.

Tuttavia, l'uso delle strutture dati per la schedulazione mostrato in [Figura 10.4](#) crea il problema di una forte contesa per l'acquisizione dei mutex lock  $L_{rq}$  e  $L_{awt}$  poiché ogni CPU deve impostare e rilasciare questi lock durante la schedulazione. Per ridurre questo overhead, alcuni sistemi operativi partizionano l'insieme dei processi in più sottoinsiemi di processi e assegnano ogni sottoinsieme a una differente CPU per la schedulazione. In questa organizzazione, le code dei processi pronti e la tabella di carico assegnato vengono partizionate in base alle CPU. Ogni CPU accederebbe alla struttura dati contenente la coda dei processi *pronti* a essa assegnati. In un kernel prelazionabile, i mutex lock sarebbero ancora necessari per evitare race condition su ogni struttura dati poiché la CPU potrebbe essere interrotta a causa degli interrupt. Tuttavia, questi lock raramente sarebbero contesi, per cui l'overhead di sincronizzazione sarebbe basso. Il prezzo per questa riduzione dell'overhead di sincronizzazione viene pagato o in termini di prestazioni del sistema, poiché alcune CPU potrebbero essere idle mentre altre sarebbero molto utilizzate, o in termini di overhead per il bilanciamento del carico tra le CPU, a causa dei trasferimenti di processi da una CPU con carico di lavoro elevato a una con poco carico.

Un kernel SMP fornisce la *graceful degradation* poiché continua a operare anche in caso di malfunzionamenti, sebbene l'efficienza potrebbe essere ridotta. Per esempio, il malfunzionamento di una CPU quando non sta eseguendo codice del kernel non interferisce con il funzionamento delle altre CPU del sistema. Dunque queste continuerebbero a funzionare normalmente. La non disponibilità della CPU danneggiata colpirebbe il processo in esecuzione sulla CPU al momento del malfunzionamento. Inoltre in qualche misura avrebbe effetto sul throughput e sui tempi di risposta del sistema, poiché un minor numero di processi possono essere schedulati in parallelo.

### Kernel NUMA

Le CPU in un sistema NUMA hanno differenti tempi di accesso alla memoria locale e non locale. Un processo funzionerebbe in maniera più efficiente se le istruzioni e gli operandi cui fa riferimento si trovassero per la maggior parte nella memoria locale. Per tener fede a questo principio, ogni nodo in un sistema NUMA ha un proprio *kernel separato* e schedula in maniera esclusiva i processi i cui spazi di indirizzamento si trovano nella memoria locale del nodo. Questo approccio è analogo al partizionamento dei processi per ogni CPU di un sistema SMP, per cui eredita gli inconvenienti di questa organizzazione.

I sistemi operativi per le architetture NUMA generalizzano questo concetto di gestione separata di ogni nodo. Utilizzano la nozione di *regione di applicazione* per garantire buone prestazioni a un'applicazione. Una regione di applicazione si compone di una partizione delle risorse e di una istanza del kernel. La partizione delle risorse contiene una o più CPU, alcune unità di memoria locale e alcune periferiche di I/O. Il kernel di una regione di applicazione gestisce solo i processi di un'applicazione. Il vantaggio di questa organizzazione è che il kernel può ottimizzare le prestazioni dell'applicazione mediante una schedulazione intelligente. Inoltre può garantire elevati hit ratio nella cache L1 schedulando un processo la maggior parte delle volte sulla stessa CPU. Buoni hit ratio si verificano anche nella cache L3 poiché le unità di memoria nella regione di applicazione contengono gli spazi di indirizzamento dei processi di una singola applicazione.

L'uso di un kernel separato per un nodo di un sistema NUMA o per una regione di applicazione ha anche alcuni svantaggi. Gli accessi alle unità di memoria non locali diventano più complessi, poiché spaziano sui domini di più di un kernel. L'organizzazione con kernel separati inoltre soffre dei tipici problemi associati al partizionamento: si può verificare uno scarso utilizzo delle risorse poiché le risorse non utilizzate in una

partizione non possono essere usate dai processi di un'altra partizione. Inoltre l'efficienza risulta scarsa poiché un'elaborazione deve essere interrotta o ritardata se si verifica il malfunzionamento di alcune risorse (compresa la CPU) di una partizione.

## 10.4 Sincronizzazione dei processi

La sincronizzazione dei processi coinvolge l'uso delle sezioni critiche o operazioni di segnalazione indivisibili. Come discusso nel Paragrafo 6.5.2, ognuna di queste è implementata utilizzando una *variabile di lock* che può assumere solo due valori: *open* e *closed*. Un processo non può iniziare l'esecuzione di una sezione critica o di un'operazione indivisibile se la variabile di lock associata alla sezione critica o all'operazione indivisibile ha valore *closed*. Se il valore della variabile di lock è *open*, il processo modifica il valore a *closed*, esegue la sezione critica oppure l'operazione indivisibile di segnalazione e imposta il valore nuovamente a *open*. Un processo che trova il valore della variabile di lock a *closed* deve attendere finché il valore è riportato a *open*. Ci riferiamo a questa organizzazione che coinvolge l'uso di una variabile di lock come *lock di sincronizzazione*, o semplicemente *lock*, e ci riferiamo alle azioni di chiusura e apertura del lock come *set* e *reset* del lock.

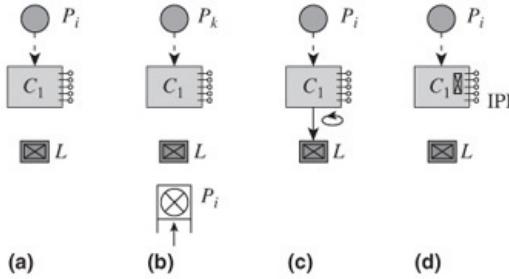
Due qualità dei lock di sincronizzazione sono importanti per le prestazioni di un sistema multiprocessore. La prima qualità è la *scalabilità*, che indica il grado al quale le prestazioni di un'applicazione che utilizza il lock sono indipendenti dal numero dei processi dell'applicazione e dal numero di CPU del sistema. La seconda qualità riguarda la capacità di una CPU di gestire gli interrupt mentre il processo in esecuzione sulla CPU è impegnato a impostare il lock di sincronizzazione. Questa capacità aiuta il kernel a fornire una risposta veloce agli eventi che si verificano nel sistema.

La [Tabella 10.4](#) riassume le caratteristiche dei tre tipi di lock di sincronizzazione, la *coda di lock*, *spin* e *sleep lock*. I processi in attesa di un lock in coda passano nello stato *blocked*; sono attivati in ordine FCFS quando il lock viene rilasciato. Lo spin lock è il lock di sincronizzazione illustrato nelle [Figure 6.9](#) e [6.10](#); porta a un'attesa attiva poiché un processo che sta tentando di impostarlo viene bloccato. È interessante notare che abbiamo scartato lo spin lock a causa della busy wait, ma è utile nei sistemi multiprocessore! Lo sleep lock è un nuovo tipo di lock. Di seguito verranno descritte le caratteristiche di tutti e tre i tipi di lock.

| Lock         | Descrizione  |
|--------------|--|
| Coda di lock | Un processo in attesa di un lock accodato si trova nello stato <i>blocked</i> e il suo id viene inserito in una coda di processi in attesa del lock. Il processo è attivato quando viene eseguito il reset del lock ed è il primo processo in coda.  |
| Spin lock    | Se uno spin lock è già impostato quando il processo tenta di impostarlo, il processo entra in una <i>busy wait</i> per il lock. La CPU su cui il processo è in esecuzione può gestire gli interrupt durante l'attesa attiva.   |
| Sleep lock   | Quando un processo è in attesa di uno sleep lock, la CPU su cui è in esecuzione viene impostata in uno speciale stato <i>sleep</i> in cui non esegue istruzioni né gestisce gli interrupt. La CPU viene attivata quando la CPU che resetta il lock gli invia un interrupt interprocessore. |

**Tabella 10.4** Tipi di lock di sincronizzazione.

La [Figura 10.5](#) mostra l'uso dei tre tipi di lock di sincronizzazione. La [Figura 10.5\(a\)](#) mostra un processo  $P_i$  in esecuzione sulla CPU  $C_1$  e un lock  $L$  utilizzato per controllare una regione di mutua esclusione. Il segno  $\times$  nel contenitore che rappresenta il lock indica che il lock è impostato. Un segno simile all'interno di un cerchio, che rappresenta un processo, indica che il processo è nello stato *blocked*. Descriveremo le caratteristiche di questi lock di sincronizzazione in seguito.



**Figura 10.5** Lock di sincronizzazione nei sistemi operativi multiprocessore. (a) Schema generale di un lock che controlla una regione di mutua esclusione; (b) coda di lock; (c) spin lock; (d) sleep lock.

### Coda di lock

Una coda di lock è un lock convenzionale utilizzato per la sincronizzazione dei processi. Il kernel esegue le seguenti azioni quando il processo  $P_i$  in esecuzione sulla CPU  $C_1$  richiede un lock  $L$ . Il lock  $L$  viene controllato. Se non è già impostato, il kernel impone il lock per conto del processo  $P_i$  e riprende la sua esecuzione. Se il lock è già detenuto da un altro processo,  $P_i$  viene bloccato e la sua richiesta del lock viene inserita in una coda. La [Figura 10.5\(b\)](#) illustra la situazione successivamente al blocco di  $P_i$ . L'id di  $P_i$  viene inserito nella coda del lock  $L$  e la CPU  $C_1$  passa a eseguire un altro processo  $P_k$ . Quando il processo che detiene il lock  $L$  termina l'esecuzione della sezione critica, il processo in testa alla coda di  $L$  viene attivato e gli viene assegnato il lock.

Un processo che non può impostare un lock rilascia la CPU su cui è in esecuzione. Tale processo non utilizzerà la CPU e non accederà alla memoria mentre è in attesa di impostare il lock. La lunghezza media della coda per un lock determina se la soluzione è scalabile. Se i processi non richiedono frequentemente il lock  $L$ , la lunghezza della coda è limitata da una costante  $c$  (ovvero, non è mai più grande di  $c$ ). Dunque, aumentare il numero di CPU o di processi nel sistema non aumenta il ritardo medio nell'acquisizione del lock. La soluzione è scalabile sotto queste condizioni. Se i processi richiedono il lock  $L$  frequentemente, la lunghezza della coda può essere proporzionale al numero di processi. In questo caso la soluzione non è scalabile.

### Spin lock

Uno spin lock differisce dalla coda di lock poiché un processo che non riesce a impostare un lock non rilascia la CPU. Invece, entra in un ciclo in cui esegue tentativi ripetuti per impostare il lock finché non lo ottiene [[Figura 10.5\(c\)](#)]. Da tale comportamento deriva il nome *spin lock*. Rappresentiamo la situazione in cui la CPU  $C_1$  cicla su un lock  $L$  con una freccia da  $C_1$  a  $L$ . La CPU  $C_1$  accede ripetutamente al valore del lock e lo controlla, utilizzando un'istruzione indivisibile come l'istruzione test-and-set (Paragrafo 6.9.4). Questa azione crea traffico sul bus di memoria o sulla rete.

L'uso degli spin lock può causare un degrado delle prestazioni del sistema sotto due aspetti: primo, la CPU è impegnata a eseguire il processo che cicla sullo spin lock per cui gli altri processi non possono utilizzare la CPU; secondo, viene generato traffico per la memoria mentre la CPU cicla sul lock. Il secondo inconveniente può non essere significante se il bus di memoria o la rete hanno poco carico, ma causa un degrado delle prestazioni in altre situazioni. Tuttavia, l'uso degli spin lock può essere giustificato in due situazioni: (1) quando il numero di processi non eccede il numero di CPU nel sistema, poiché non c'è vantaggio nel prelazionare un processo e (2) quando un lock viene usato per controllare una sezione critica e il tempo di CPU necessario per eseguire la sezione critica è minore del tempo totale di CPU necessario per bloccare un processo, schedularne un altro, attivare e rischedularne il processo originario. Nel primo caso il blocco non è necessario. Nel secondo caso è controproducente.

Uno spin lock presenta un vantaggio interessante su una coda di lock. Una CPU che cicla su un lock può gestire gli interrupt e il processo in esecuzione su di essa può gestire i segnali. Questa caratteristica è particolarmente importante in un'applicazione real-time dal momento che i ritardi nel servizio degli interrupt e dei segnali può far degradare i tempi di risposta. Tuttavia, gli spin lock non sono scalabili, a causa del traffico di

memoria e di rete che generano.

In un sistema NUMA, un processo che utilizza gli spin lock può affrontare una situazione chiamata *lock starvation*, in cui gli potrebbe essere negato il lock per lunghi periodi di tempo, anche indefinitamente. Si consideri un processo  $P_i$  che tenta di impostare uno spin lock che si trova nella sua memoria non locale. Siano  $P_j$  e  $P_k$  due processi, residenti nello stesso nodo del lock, che cercano di impostare il lock. Poiché l'accesso alla memoria locale è molto più veloce dell'accesso alla memoria non locale, i processi  $P_j$  e  $P_k$  possono ciclare più velocemente sul lock rispetto al processo  $P_i$ . Dunque, verosimilmente, hanno l'opportunità di impostare il lock prima di  $P_i$ . Se impostano e usano il lock ripetutamente,  $P_i$  può non essere in grado di impostare il lock per un lungo tempo. Uno schema che vedremo nel Paragrafo 10.4.2 evita la starvation del lock.

### ***Sleep lock***

Quando un processo esegue un tentativo non proficuo di impostare uno sleep lock, la CPU su cui è in esecuzione è impostata in uno stato speciale chiamato stato *sleep*. In questo stato non può eseguire istruzioni e non risponde a nessun interrupt, fatta eccezione per gli interrupt interprocessore. Nella [Figura 10.5\(d\)](#) rappresentiamo questa situazione contrassegnando con un  $\times$  tutti gli interrupt, eccetto gli IPI. La CPU in attesa del lock non cicla su di esso, per cui non genera traffico di memoria e di rete.

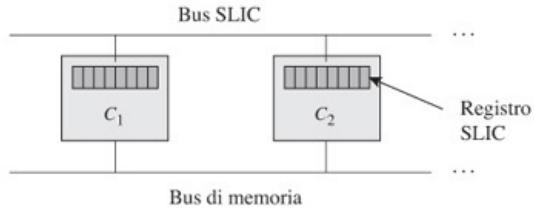
La CPU che rilascia il lock ha la responsabilità di inviare gli interrupt interprocessore a quelle CPU che si trovano nello stato sleep sul lock. Questa caratteristica comporta l'overhead dovuto alla generazione e alla gestione degli interrupt interprocessore, entrambe le quali coinvolgono un cambio di contesto e l'esecuzione di codice kernel. Lo sleep lock non scalerà bene se vi sono molte contese per un lock; tuttavia, ha buone prestazioni se non si verifica questa situazione. L'uso degli sleep lock in un'applicazione real-time può anche interessare i tempi di risposta dell'applicazione. Nonostante ciò gli sleep lock possono essere preferiti agli spin lock in caso di alta densità di traffico di memoria o di rete.

### ***Lock adattivo***

Come discusso in precedenza, alcuni tipi di sincronizzazione sono efficaci solo quando i processi coinvolti nella sincronizzazione sono schedulati per l'esecuzione allo stesso tempo. Il SO Solaris per i sistemi Sun fornisce un lock di sincronizzazione chiamato *lock adattivo*. Un processo in attesa di questo lock cicla su di esso se il detentore del lock è schedulato per l'esecuzione parallela; altrimenti, il processo viene prelazionato e accodato come nella coda di lock. In questo modo, l'implementazione di un lock di sincronizzazione dipende dalle decisioni di scheduling nel sistema.

## **10.4.1 Hardware speciale per la sincronizzazione dei processi**

Alcuni sistemi utilizzano hardware speciale per evitare il problema delle scarse prestazioni causato dalla coda dei lock, dagli spin lock e dagli sleep lock. Il sistema Sequent Balance utilizza un bus speciale chiamato *system link e interface controller* (SLIC) per la sincronizzazione. Il sistema SLIC consiste di uno speciale registro a 64 bit in ogni CPU del sistema. I registri di differenti CPU sono connessi tramite il bus SLIC ([Figura 10.6](#)). Ogni bit rappresenta uno spin lock. In questo modo SLIC può supportare 64 spin lock. Quando una CPU  $C_1$  vuole impostare un lock  $L_k$ , prova a impostare il bit corrispondente, per esempio  $b_k$ , nel proprio registro. Se il bit non è già impostato, un tentativo di impostarlo determina una comunicazione tramite il bus SLIC. Se nessun'altra CPU sta simultaneamente tentando di impostare lo stesso bit, il lock viene attribuito a  $C_1$  e il bit  $b_k$  è impostato nei registri speciali di tutte le CPU.  $C_1$  può ora procedere con la propria esecuzione. Quando rilascia il lock, il bit  $b_k$  nei registri speciali di tutte le CPU viene resettato. Se due o più CPU cercano di impostare lo stesso bit simultaneamente, l'arbitro hardware attribuisce il lock a una CPU. Il tentativo di impostare il lock  $L$  fallisce se il bit  $b_k$  è già impostato per conto di un'altra CPU. In questo caso, la CPU continua a ciclare su questo lock, ovvero sul bit  $b_k$  del proprio registro speciale.



**Figura 10.6** SLIC bus.

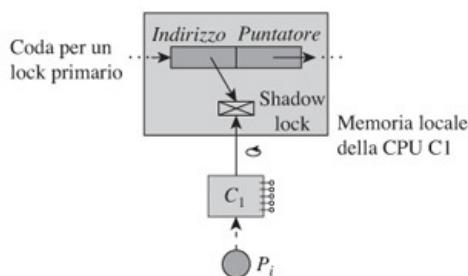
Il vantaggio dell'approccio SLIC è che una CPU cicla su un lock situato nella CPU. Quindi, ciclare non genera traffico di memoria o di rete. Inoltre, l'utilizzo degli spin lock, anziché degli sleep lock, evita l'uso degli interrupt interprocessore per la sincronizzazione. L'uso di uno speciale bus di sincronizzazione allevia la pressione sul bus di memoria. Questo è un vantaggio significativo quando la densità del traffico di memoria è alta.

#### 10.4.2 Uno schema di software scalabile per la sincronizzazione dei processi

Descriviamo uno schema per la sincronizzazione dei processi nelle architetture NUMA e NORMA che ottiene prestazioni scalabili minimizzando il traffico di sincronizzazione verso le unità di memoria non locali nell'architettura NUMA e sulla rete nell'architettura NORMA. Non richiede nessun hardware speciale e fornisce un effetto analogo al chip SLIC. Inoltre, evita il problema della lock starvation tipico degli spin lock.

Lo schema utilizza due tipi di lock. Un *lock primario* è simile a un lock convenzionale utilizzato per la sincronizzazione. Quando un processo non è in grado di impostare un lock primario, incrementa uno *shadow lock* nella memoria locale del nodo dove risiede, associa lo shadow lock al lock primario e cicla sullo shadow lock. In questo modo lo spin non genera traffico di memoria non locale o traffico di rete. Quando un processo desidera resettare un lock primario che ha precedentemente impostato, controlla se qualche shadow lock è associato al lock primario. In questo caso, resetta uno degli shadow lock, il quale riattiva uno dei processi in attesa del lock primario per procedere; altrimenti, resetta il lock primario.

La [Figura 10.7](#) illustra un'implementazione di questo schema, utilizzando la stessa notazione di [Figura 10.5](#). Una coda di shadow lock viene mantenuta per ogni lock primario. Ogni elemento nella coda contiene l'indirizzo di uno shadow lock nella coda. Se un processo non riesce a impostare il lock primario, il processo alloca uno shadow lock nella memoria locale, inserisce il suo indirizzo nella coda del lock primario e inizia a ciclare su di esso. La coda può coprire differenti unità di memoria nel sistema; in questo modo l'inserimento dello shadow lock nella coda genera traffico di memoria non locale o traffico di rete. Anche il reset dello shadow lock genera traffico di memoria non locale o traffico di rete. Tuttavia, lo spin non genera tale traffico. È inutile dire che la manipolazione della coda dovrebbe essere effettuata mediante l'utilizzo di un lock.



**Figura 10.7** Una soluzione software efficiente per la sincronizzazione dei processi.

#### 10.5 Scheduling dei processi

Un processo può essere schedulato su qualunque CPU di un sistema multiprocessore. Tuttavia, le sue prestazioni possono essere migliorate effettuando un scelta intelligente della CPU, ovvero decidendo *dove* schedularlo. Le prestazioni di un gruppo di processi che si sincronizzano e comunicano l'uno con l'altro possono essere migliorate decidendo *come* e *dove* schedularli. Questo paragrafo tratta le problematiche coinvolte nell'intraprendere queste decisioni.

### Scelta della CPU

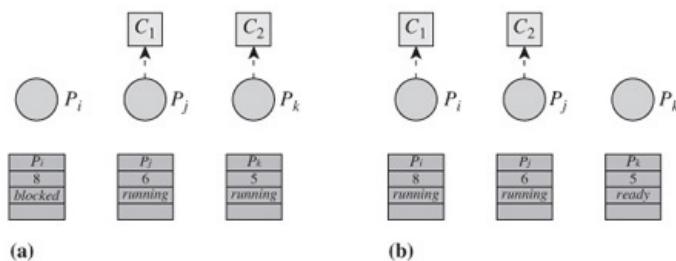
Quando un processo  $P_i$  viene eseguito su una CPU, per esempio la CPU  $C_1$ , alcune parti del suo spazio di indirizzamento sono caricate nella cache L1 del processore. Quando la CPU viene commutata a un altro processo, alcune di queste parti sono sovrascritte da parti dello spazio di indirizzamento del nuovo processo; tuttavia, alcune altre parti dello spazio di indirizzamento di  $P_i$  possono restare nella memoria cache di  $C_1$  per qualche tempo. Queste parti sono chiamate *spazio di indirizzamento residuo* di un processo. Si dice che un processo ha un'*affinità* con una CPU se ha uno spazio di indirizzamento residuo nella sua cache. Il processo avrebbe un hit ratio maggiore su questa CPU piuttosto che su una CPU con la quale non ha affinità.

Lo *scheduling per affinità* schedula un processo su una CPU con la quale ha un'affinità. Questa tecnica fornisce un buon hit ratio, velocizzando di conseguenza l'esecuzione del processo e riducendo il traffico sul bus di memoria. Un altro modo per sfruttare l'affinità è di schedulare i thread di un processo sulla stessa CPU in successione stretta. Tuttavia, lo scheduling per affinità interferisce con il bilanciamento del carico tra le CPU poiché i processi e i thread vengono collegati a determinate CPU. Il Paragrafo 10.6.3 descrive come questo scheduling porta anche ad anomalie nel sistema Windows.

Nel Paragrafo 10.3, abbiamo discusso come il kernel SMP consente a ogni CPU di eseguire il proprio scheduling. Questa organizzazione consente al kernel di non diventare un collo di bottiglia per le prestazioni; tuttavia, questo porta ad anomalie di schedulazione in cui un processo ad alta priorità si trova nello stato *ready* anche se è stato schedulato un processo a più bassa priorità. Correggere questa anomalia richiede la distribuzione dei processi tra le CPU, come indicato nel prossimo esempio.

### Esempio 10.1 - Distribuzione dei processi in un kernel SMP

Un sistema SMP contiene due CPU  $C_1$  e  $C_2$  e tre processi  $P_i$ ,  $P_j$  e  $P_k$  con priorità rispettivamente 8, 6 e 5. La [Figura 10.8\(a\)](#) mostra la situazione in cui il processo  $P_i$  si trova nello stato *blocked* a causa di un'operazione di I/O (vedi il contenuto dei campi del suo PCB) e i processi  $P_j$  e  $P_k$  sono in esecuzione, rispettivamente, sulle CPU  $C_1$  e  $C_2$ . Quando l'operazione di I/O di  $P_i$  viene completata, l'interrupt di I/O viene elaborato dalla CPU  $C_1$ , che cambia lo stato di  $P_i$  a *ready* e passa all'esecuzione del processo  $P_i$ . Dunque, il processo  $P_j$ , che è il processo con la successiva priorità più alta, è nello stato *ready* e  $P_k$ , la cui priorità è la più bassa, è in esecuzione. Per correggere questa situazione, il processo  $P_k$  dovrebbe essere prelazionato e il processo  $P_j$  dovrebbe essere schedulato sulla CPU  $C_2$ . La [Figura 10.8\(b\)](#) mostra la situazione dopo l'esecuzione di queste azioni.



**Figura 10.8** Il processo  $P_j$  è spostato dalla CPU  $C_1$  alla CPU  $C_2$  quando il processo  $P_i$  diventa *ready*.

La distribuzione dei processi può essere implementata utilizzando la tabella di carico assegnato (AWT), illustrata nel Paragrafo 10.3, e gli interrupt interprocessore (IPI).

Tuttavia, la distribuzione dei processi porta a un alto overhead di schedulazione; questo effetto è più evidente in un sistema che contiene un alto numero di CPU. Dunque, alcuni sistemi operativi non correggono le anomalie di schedulazione mediante la distribuzione dei processi.

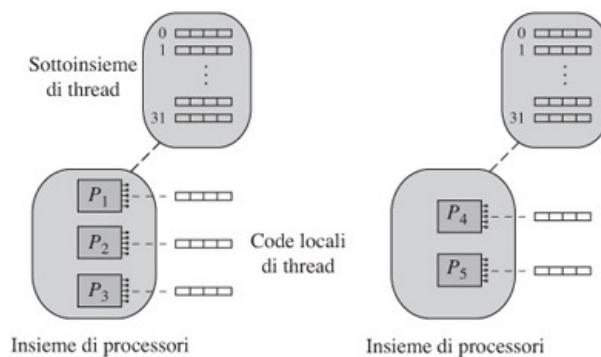
### **Sincronizzazione-scheduling consapevole**

Parti di una elaborazione possono essere eseguiti su differenti CPU per ottenere l'incremento della velocità di elaborazione. Tuttavia, la sincronizzazione e la comunicazione tra i processi di un'applicazione influenzano la natura del parallelismo tra i suoi processi, per cui una politica di scheduling dovrebbe tenerne conto. Come commentato in precedenza nel Paragrafo 10.2, i processi di un'applicazione dovrebbero essere schedulati su diverse CPU allo stesso tempo nel caso in cui utilizzino gli spin lock per la sincronizzazione. Questo è chiamato *co-scheduling* o *gang scheduling*. Un approccio differente è richiesto quando i processi scambiano messaggi utilizzando un protocollo bloccante. Quando  $P_i$  invia un messaggio a  $P_j$ , può proseguire con la propria elaborazione solo dopo che il messaggio è stato consegnato. Questa attesa potrebbe essere abbastanza lunga, per cui è meglio bloccare  $P_i$ . In questi casi vengono effettuate speciali procedure per *non* schedulare questi processi nello stesso intervallo di tempo. Poiché questo approccio è conflittuale con il coscheduling, il kernel deve prendere una decisione difficile. Può basare la sua decisione sul comportamento passato dei processi nell'applicazione o sulle preferenze dell'utente per uno specifico metodo di schedulazione. Il sistema operativo Mach utilizza il secondo approccio.

## **10.6 Casi di studio**

### **10.6.1 Mach**

Il sistema operativo Mach, sviluppato al Carnegie Mellon University, è un SO per sistemi multiprocessore e distribuiti. Il Mach multiprocessore utilizza la struttura del kernel SMP. La [Figura 10.9](#) mostra una visione d'insieme dell'organizzazione dello scheduling in Mach. I processori del sistema multiprocessore sono divisi in *insiemi di processori*. A ogni insieme di processori viene assegnato un sottoinsieme di thread per l'esecuzione. I thread possono avere priorità tra 0 e 31, dove 0 è la priorità più alta. Ogni insieme di processori ha 32 code di processi pronti, per mantenere le informazioni relative ai thread in ogni livello di priorità. Queste code sono condivise da tutti i processori dell'insieme. Inoltre, ogni processore ha una coda locale dei thread. Questi sono i thread che devono essere eseguiti solo su questo processore. Questi thread hanno una priorità maggiore di tutti i thread nelle code dei thread. Questa caratteristica implementa lo scheduling per affinità. Un thread viene prelazionato alla fine di una time slice solo se è presente qualche altro thread pronto nelle code dei thread, altrimenti al thread viene concessa un'altra time slice. La time slice viene modificata in base al numero dei thread pronti: una time slice più breve se sono presenti molti thread pronti e una time slice più lunga se sono presenti pochi thread pronti.



**Figura 10.9** Scheduling in Mach.

Una caratteristica interessante del sistema operativo Mach è la tecnica dei

*suggerimenti per lo scheduling*. Un thread fornisce un suggerimento per influenzare le decisioni di schedulazione del processore. Si presume che un suggerimento sia basato sulla conoscenza da parte del thread di qualche caratteristica dell'esecuzione di un'applicazione. Un thread può fornire un suggerimento per assicurare una migliore schedulazione quando i thread di un'applicazione richiedono sincronizzazione o comunicazione. Un suggerimento negativo riduce la priorità di un thread. Questo tipo di suggerimento può essere fornito da un thread che deve ciclare su un lock che è stato impostato da qualche altro processo. Un suggerimento passivo viene dato da un thread per indicare che vuole rinunciare al processore a favore di un altro thread; il thread può anche indicare l'identità del thread al quale vuole cedere il processore. Nel ricevere questo suggerimento, lo scheduler commuta il processore al thread nominato senza tenere conto della sua priorità. Questa caratteristica può essere utilizzata efficacemente quando un thread cicla su un lock mentre il detentore del lock è prelazionato. Il thread che cicla può rinunciare al processore in favore del thread prelazionato. Questa azione porterà a un rilascio anticipato del lock e può essere utilizzata anche per implementare il priority inheritance protocol (PIP) discusso nel [Capitolo 7](#).

## 10.6.2 Linux

Il supporto per il multiprocessing in Linux fu introdotto nel kernel Linux 2.0. Il locking a grana larga fu adottato per prevenire le race condition sulle strutture dati del kernel. La granularità dei lock fu raffinata nelle versioni successive; tuttavia, il kernel era ancora non prelazionabile. Con il kernel Linux 2.6, il kernel divenne prelezionabile (Paragrafo 4.8.2). Il kernel Linux 2.6 adotta anche lock a grana molto fine.

Il kernel Linux fornisce spin lock per il lock delle strutture dati. Inoltre, fornisce uno speciale *spin lock lettore-scrittore* che consente di avere un qualunque numero di processi lettori, ovvero, i processi che non modificano nessun dato del kernel, per accedere ai dati protetti contemporaneamente; tuttavia, consente a un solo processo scrittore di aggiornare i dati a ogni istante.

Il kernel Linux utilizza un altro lock chiamato *sequence lock* che genera un basso overhead ed è scalabile. Il sequence lock è di fatto un intero che viene usato come contatore di sequenza mediante un'istruzione di incremento atomica, ovvero *indivisibile*. Quando un processo vuole utilizzare una struttura dati del kernel, incrementa semplicemente l'intero nel sequence lock associato con la struttura dati, annota il suo nuovo valore ed esegue l'operazione. Completata l'operazione, controlla se il valore nel sequence lock è cambiato dopo che ha eseguito la sua istruzione di incremento. Se il valore è cambiato, l'operazione viene considerata fallita, per cui annulla l'operazione che ha appena eseguito e cerca di effettuarla nuovamente e così via finché l'operazione riesce.

Linux utilizza strutture dati separate per ogni CPU per ridurre la contesa per i lock sulle strutture dati del kernel. Come menzionato nel Paragrafo 10.3, una CPU accede ad una struttura dati locale solo quando il codice del kernel viene eseguito da quella CPU; tuttavia, anche questa struttura dati ha bisogno di essere bloccata poiché si possono verificare accessi concorrenti quando viene generato un interrupt mentre è in esecuzione il codice kernel per la gestione di una chiamata di sistema e una routine di servizio dell'interrupt è attivata nel kernel. Linux elimina questo lock disabilitando la possibilità di prelazione (detta prelazionabilità) questa CPU causata da interrupt durante l'esecuzione di codice kernel. In pratica, il codice eseguito dalla CPU effettua una chiamata di sistema per disabilitare la prelazionabilità quando sta per accedere a strutture dati locali per ogni CPU ed esegue un'altra chiamata di sistema per abilitare la prelazionabilità quando termina l'accesso alle suddette strutture dati.

Come descritto in precedenza nel Paragrafo 7.6.3, lo scheduling di Linux utilizza le strutture dati per le code dei processi pronti di [Figura 7.12](#). Lo scheduling per multiprocessore incorpora le considerazioni di affinità: un utente può specificare un'affinità *hard* per un processo indicando un insieme di CPU su cui deve essere eseguito, mentre un processo ha un'affinità *soft* con l'ultima CPU su cui è stato eseguito. Poiché lo scheduling è eseguito per ogni CPU, il kernel esegue un *bilanciamento del carico* per garantire che i carichi computazionali diretti a differenti CPU siano paragonabili. Questo task viene effettuato da una CPU che trova le proprie code dei processi pronti vuote; inoltre, viene eseguito periodicamente dal kernel, cioè ogni milliscondo se il sistema è idle e ogni 200 millisecondi altrimenti.

La funzione `load_balance` viene invocata per effettuare il bilanciamento del carico con

l'id di una CPU con poco carico. Il `load_balance` cerca una "CPU occupata" che ha nella sua ready queue almeno il 25 per cento di processi in più rispetto alla ready queue della CPU con poco carico. A questo punto localizza alcuni processi nella sua ready queue che non hanno un'affinità hard con la CPU occupata e li sposta nella ready queue della CPU libera. Procede dapprima a spostare i processi a più alta priorità presenti nella *lista piena* della CPU occupata, poiché questi processi meno verosimilmente hanno uno spazio di indirizzamento residuo nella cache della CPU occupata rispetto a quelli nella lista *attivi*. Se è necessario spostare più processi, sposta quelli a più alta priorità presenti nella *lista attivi* della CPU occupata, cosa che migliorerebbe il loro tempo di risposta.

### 10.6.3 Supporto SMP in Windows

Il kernel Windows fornisce un supporto completo per i sistemi multiprocessore e NUMA e per le CPU dotate di hyperthreading - una CPU dotata di hyperthreading è considerata come un singolo processore fisico con diversi processori logici. Gli spin lock sono utilizzati per implementare la mutua esclusione sulle strutture dati del kernel. Per garantire che i thread non incorrano in lunghe attese per le strutture dati del kernel, Windows non prelaziona mai un thread che detiene uno spin lock se qualche altro thread sta tentando di acquisire lo stesso lock.

Windows Server 2003 e Windows Vista usano diverse free list di aree di memoria come descritto nel Paragrafo 11.5.4, che consentono alle CPU di allocare memoria in parallelo. Questi kernel utilizzano anche strutture dati per la schedulazione locale a ogni processore come descritto nel Paragrafo 10.3. Tuttavia, le CPU possono dover modificare le strutture dati delle altre CPU durante la schedulazione. Per ridurre l'overhead di sincronizzazione di questa operazione, il kernel fornisce una *coda di spinlock* che segue lo schema del Paragrafo 10.4.2: un processore cicla su un lock nella sua memoria locale, evitando di generare traffico di rete in un sistema NUMA e rendendo il lock scalabile.

Gli oggetti processo e thread di Windows hanno diversi attributi relativi allo scheduling. *L'affinità di default con il processore* di un processo e *l'affinità con il processore di un thread*, insieme, definiscono un insieme di affinità per il thread, ovvero un insieme di processori. In un sistema con architettura NUMA, un processo può essere relegato a un singolo nodo nel sistema rendendo il suo insieme di affinità un sottoinsieme dei processori del nodo. Il kernel assegna un *processore ideale* a ogni thread in modo tale che thread differenti di un processo possano essere eseguiti in parallelo, sfruttando i benefici del co-scheduling. L'insieme di affinità e il processore ideale, insieme, definiscono un' *affinità hard* per un thread. Si assume che un processore contenga una parte dello spazio di indirizzamento di un thread per 20 millisecondi dopo che il thread ha terminato la propria esecuzione su di esso. Il thread ha un' *affinità soft* con il processore durante questo intervallo, per cui la sua identità viene conservata nell'attributo *ultimo processore* del thread.

Quando viene eseguito lo scheduling, per esempio per la CPU  $C_1$ , il kernel esamina i thread *ready* in ordine di priorità decrescente e seleziona il primo thread *ready* che soddisfa una delle seguenti condizioni:

- $C_1$  è l'ultimo processore utilizzato dal thread;
- $C_1$  è il processore ideale del thread;
- $C_1$  è nell'insieme di affinità del thread ed è nello stato *ready* da tre tick di clock.

Il primo criterio realizza lo scheduling per affinità soft, mentre gli altri due criteri realizzano lo scheduling per affinità hard. Se il kernel non riesce a trovare un thread che soddisfa uno di questi criteri, schedula semplicemente il primo thread *ready* che trova. Se non esiste nessun thread *ready*, schedula il *thread idle* (Paragrafo 7.6.4).

Quando un thread passa nello stato *ready* a causa di un interrupt, la CPU che gestisce l'interrupt sceglie una CPU per mandare in esecuzione il thread appena passato nello stato *ready* come segue. Dapprima controlla se ci sono CPU idle nel sistema e se una di queste è il processore ideale o l'ultimo processore del thread *ready*. In caso affermativo, schedula il thread passato nello stato *ready* su questa CPU inserendo l'id del thread nella struttura dati per la schedulazione della CPU selezionata. La CPU idle selezionata starebbe eseguendo il *thread idle*, il quale assumerebbe l'identità del thread schedulato nell'iterazione successiva del ciclo idle e commuterebbe al nuovo thread. Se il processore ideale o l'ultimo processore del thread passato nello stato *ready* non è idle, la CPU che gestisce l'interrupt è essa stessa idle e si trova nell'insieme di affinità del thread *ready*,

allora prende il thread per l'esecuzione. Se questo controllo fallisce e alcune CPU nell'insieme di affinità del thread sono idle, viene schedulato il thread sulla CPU idle con numero inferiore; altrimenti viene schedulato il thread sulla CPU idle con numero inferiore che non è inclusa nell'insieme di affinità del thread.

Se non ci sono CPU idle, la CPU che gestisce l'interrupt confronta le priorità del thread diventato *ready* e del thread in esecuzione sul processore ideale del thread diventato *ready*. Se quest'ultimo ha una priorità maggiore, un interrupt interprocessore viene inviato al suo processore ideale con la richiesta di commutare l'esecuzione al thread diventato *ready*. In caso contrario, viene eseguito un controllo analogo sull'ultimo processore utilizzato dal thread diventato *ready*. Se anche questo controllo fallisce, la CPU che gestisce l'interrupt inserisce il thread nella ready queue e, successivamente, potrebbe essere schedulato da una CPU idle. In questo caso, si può verificare una situazione anomala nel sistema poiché la priorità del thread diventato *ready* può essere maggiore della priorità di qualche thread in esecuzione su un'altra CPU. Tuttavia, la correzione di questa anomalia può causare troppa dispersione dei thread tra le CPU, per cui non viene eseguita dalla politica di scheduling.

## Riepilogo

Un SO multiprocessore sfrutta la presenza di più CPU nel computer per fornire un *elevato throughput*, un *aumento della velocità di esecuzione* delle applicazioni e la *graceful degradation* delle funzionalità del SO nel caso in cui si verifichi un malfunzionamento nel sistema. In questo capitolo abbiamo studiato l'architettura dei sistemi multiprocessore e le problematiche dei SO per garantire buone prestazioni.

I sistemi multiprocessore sono classificati in tre tipi in base al modo in cui diverse CPU possono accedere alla memoria. Nell'architettura *uniform memory access* (UMA), la memoria è condivisa tra tutte le CPU. Questa architettura è anche chiamata architettura *symmetrical multiprocessor* (SMP). Nell'architettura *nonuniform memory access* (NUMA), ogni CPU è dotata di una memoria locale cui ha accesso più velocemente rispetto al resto della memoria che è accessibile attraverso una rete di interconnessione.

Un SO multiprocessore dovrebbe sfruttare la presenza di più CPU per schedulare i processi in parallelo e anche per assicurare l'efficienza del proprio funzionamento. Due problematiche sono importanti in questo contesto: la struttura del kernel e i ritardi causati dalla sincronizzazione e dalla schedulazione. Molte CPU dovrebbero essere in grado di eseguire il codice del kernel in parallelo in modo che il kernel possa rispondere agli eventi prontamente e non diventi un collo di bottiglia per le prestazioni. La sincronizzazione e lo scheduling dei processi utente dovrebbero essere eseguiti in modo tale che i processi non incorrano in lunghi ritardi. Il SO deve anche assicurare che i suoi algoritmi siano *scalabili*; ovvero, abbiano buone prestazioni anche quando la dimensione del sistema aumenta in conseguenza dell'aumento del numero di CPU, delle unità di memoria o dei processi utente.

I SO multiprocessore utilizzano speciali tipi di lock chiamati *spin lock* e *sleep lock* per controllare l'overhead della sincronizzazione dei processi. Lo *scheduling per affinità* viene utilizzato per schedulare un processo sulla stessa CPU in modo tale che possa ottenere elevati hit ratio durante l'esecuzione, mentre il *co-scheduling* è utilizzato per schedulare i processi di un'applicazione su differenti CPU allo stesso tempo, in modo che possano comunicare in maniera efficiente tra di loro. I sistemi operativi adottano la *distribuzione dei processi* per assicurare che i processi a più alta priorità siano sempre in esecuzione sulla propria CPU. In questo contesto abbiamo affrontato le caratteristiche dei sistemi operativi Linux, Mach e Windows.

## Domande

- 10.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. Lo scheduling eseguito da una CPU in un sistema multiprocessore simmetrico può risultare nella distribuzione dei processi in esecuzione su molte CPU del sistema.
  - b. L'interrupt interprocessore (IPI) non è usato nella sincronizzazione dei processi in un sistema multiprocessore simmetrico.

- c. Quando un processo cicla su un lock, inficia le prestazioni dei processi in esecuzione su altre CPU.
  - d. Quando viene utilizzato lo scheduling per affinità, un processo può richiedere meno tempo di CPU per completare la propria esecuzione.
- 10.2. Quale sarebbe la conseguenza di non implementare la coerenza della cache in un sistema multiprocessore?
- a. I risultati prodotti da un processo che non interagisce con nessun altro processo potrebbero essere errati.
  - b. I risultati prodotti da un gruppo di processi che interagiscono utilizzando la stessa CPU potrebbero essere errati.
  - c. I risultati prodotti da un gruppo di processi che interagiscono non utilizzando la stessa CPU potrebbero essere errati.
  - d. Nessuna delle precedenti.

## Problemi

- 10.1. Descrivere due situazioni in cui un kernel SMP richiede l'uso dell'interrupt interprocessore (IPI).
- 10.2. Un SO assegna la stessa priorità a tutti i processi (o thread) di un'applicazione, ma utilizza differenti priorità per differenti applicazioni.
- a. In un sistema con singolo processore, questa assegnazione delle priorità fornisce un vantaggio simile a quello fornito dallo scheduling per affinità?
  - b. In un sistema multiprocessore, l'assegnazione delle priorità fornisce un vantaggio simile a quello fornito dal co-scheduling?
- 10.3. La caratteristica del Mach di rilasciare la CPU a un processo può essere utilizzata come vantaggio nell'implementazione dello schema software per la sincronizzazione dei processi discusso nel Paragrafo 10.4.2?
- 10.4. L'inversione della priorità può verificarsi quando vengono utilizzati gli spin lock o gli sleep lock? (Paragrafo 6.5.1 per una definizione di inversione delle priorità.)
- 10.5. Discutere l'idoneità dei vari tipi di lock per la sincronizzazione di attività parallele in un kernel SMP.
- 10.6. I processi di un'applicazione interagiscono tra di loro molto frequentemente. Tra la coda di lock, spin lock e sleep lock quale consideri idoneo per l'implementazione di questa applicazione su un sistema multiprocessore e perché?

## Note bibliografiche

Molti libri sulle architetture dei computer discutono le architetture dei multiprocessori e delle reti di interconnessione, per esempio Hennessy e Patterson (2002), Hamacher et al. (2002) e Stallings (2003).

Mellor-Crummey e Scott (1991), Menasse et al. (1991) e Wisniewski et al. (1997) discutono la sincronizzazione dei processi in un ambiente multiprocessore. La soluzione software efficiente per la sincronizzazione dei processi descritta in [Figura 10.7](#) è adattata da Mellor-Crummey e Scott (1991). Ousterhout (1982), Tucker e Gupta (1989) e Squillante (1990) discutono le problematiche relative allo scheduling nei sistemi operativi multiprocessore.

Eykholt et al. (1992) discutono il multithreading del kernel SunOS per migliorare l'efficacia della sua struttura SMP. Accetta et al. (1986) descrivono il sistema operativo multiprocessore Mach. Love (2005) discute la sincronizzazione e lo scheduling in Linux 2.6, mentre Russinovich e Solomon (2005) descrivono la sincronizzazione e lo scheduling di Windows.

1. Accetta, M., R. Baron, W. Bolosky, D.B. Golub, R. Rashid, A. Tevanian, and M. Young (1986): "Mach: A new kernel foundation for Unix development," *Proceedings of the Summer 1986 USENIX Conference*, June 1986, 93-112.
2. Eykholt, J.R., S.R. Kleiman, S. Barton, S. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. William (1992): "Beyond multiprocessing: multithreading the SunOS kernel," *Proceedings of the Summer 1992 USENIX Conference*, 11-18.

3. Hamacher, C., Z. Vranesic, and S. Zaky (2002): *Computer Organization*, 5th ed., McGraw-Hill, New York.
4. Hennessy, J., and D. Patterson (2002): *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, San Mateo, Calif.
5. Mellor-Crummey, and M.L. Scott (1991): "Algorithms for scalable synchronization on shared memory multiprocessor," *ACM Transactions on Computer Systems*, **9** (1), 21-65.
6. Karlin, A.R., K. Li, M.S. Menasse, and S. Owicki (1991): "Empirical studies of competitive spinning for shared memory multiprocessor," *Proceedings of 13th ACM Symposium on Operating System Principles*, 41-55.
7. Kontothanassis L.I., R.W. Wisniewski, and M.L. Scott (1997): "Scheduler conscious synchronization," *ACM Transactions on Computer Systems*, **15** (1), 3-40.
8. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
9. Ousterhout, J.K. (1982): "Scheduling techniques for concurrent systems," *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 22-30.
10. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
11. Squillante, M. (1990): "Issues in shared-memory multiprocessor scheduling: A performance evaluation," Ph.D. dissertation, Dept. of Computer Science & Engineering, University of Washington.
12. Stallings, W. (2003): *Computer Organization and Architecture*, 6th ed., Prentice Hall, Upper Saddle River, N.J.
13. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
14. Tucker, A., and A. Gupta (1989): "Process control and scheduling issues for multiprogrammed shared memory multiprocessors," *Proceedings of 12th ACM Symposium on Operating System Principles*, 159-166.

---

## PARTE 3

# Gestione della memoria

---

La memoria di un computer è condivisa da un grande numero di processi, per cui la gestione della memoria è tradizionalmente una parte molto importante di un sistema operativo. Le memorie diventano sempre meno costose e sempre più capienti; tuttavia, a tutt'oggi permangono le aspettative rispetto alla memoria come risorsa di un SO poiché sia la dimensione dei processi che il numero di processi che un sistema operativo deve gestire aumentano costantemente. Le problematiche principali nella gestione della memoria sono l'uso efficiente della memoria, la protezione della memoria allocata a un processo contro gli accessi illegali da parte di altri processi, le prestazioni dei singoli processi e le prestazioni del sistema.

L'uso efficiente della memoria è importante poiché determina il numero di processi che possono essere tenuti in memoria a ogni istante. Questo numero, d'altronde, influenza le prestazioni del sistema poiché la presenza di pochi processi potrebbe portare a uno spreco della CPU. Sia l'efficienza della memoria che le prestazioni del sistema decadono quando alcune aree di memoria restano inutilizzate perché troppo piccole per contenere un processo. Questa situazione viene chiamata *frammentazione della memoria*.

La tecnica dell'*allocazione non contigua della memoria* consente di ottenere un uso efficiente della memoria contrastando la frammentazione della memoria. Quando un SO non trova un'area di memoria abbastanza grande da contenere un processo, alloca diverse aree di memoria non contigue al processo. Nell'hardware di un computer esistono speciali funzionalità per supportare l'esecuzione di tale processo. I sistemi operativi sfruttano l'allocazione non contigua della memoria per utilizzare solo alcune parti di un processo, cosa che crea l'illusione di una memoria del computer maggiore di quella realmente installata. Questa illusione viene chiamata *memoria virtuale*.

---

### Linee guida per la Parte 3



---

Schema che mostra l'ordine in cui i capitoli di questa parte dovrebbero essere affrontati in un corso.

### Capitolo 11 - Gestione della memoria

Questo capitolo è dedicato ai principi fondamentali della gestione della memoria. Comincia discutendo come la protezione della memoria è implementata nell'hardware utilizzando speciali registri nella CPU. Affronta le problematiche relative all'uso efficiente della memoria e le tecniche per l'allocazione e la deallocazione veloce della memoria. Successivamente vengono descritti gli approcci basati sull'allocazione non contigua della memoria chiamati *paginazione* e *segmentazione*. Il capitolo discute anche le speciali tecniche adottate dal kernel per gestire le proprie richieste di memoria in maniera efficiente.

## Capitolo 12 - Memoria virtuale

Questo capitolo affronta in dettaglio l'implementazione della memoria virtuale mediante *paginazione*. Discute di come il kernel mantiene sul disco il codice e i dati di un processo e carica le parti in memoria quando necessarie e di come le prestazioni di un processo sono determinate dal tasso al quale le parti di un processo devono essere caricate dal disco. Mostra come questo tasso dipenda dalla quantità di memoria allocata a un processo e l'*algoritmo di sostituzione delle pagine* utilizzato per decidere quali pagine di un processo debbano essere rimosse dalla memoria in modo tale che nuove pagine possano essere caricate. Successivamente, sono discussi gli algoritmi di sostituzione delle pagine che usano la regola empirica della *località dei riferimenti*. Inoltre, viene descritta l'implementazione della memoria virtuale mediante *segmentazione*.

---

# CAPITOLO 11

## Gestione della memoria

---

### Obiettivi di apprendimento

- Gerarchia di memoria
- Modalità di allocazione della memoria: statica e dinamica
- Esecuzione dei programmi
- Gestione dell'allocazione della memoria ai processi
- Allocazione contigua e non contigua della memoria
- Paginazione, segmentazione e segmentazione con paginazione
- Gestione della memoria del kernel

Come visto nel [Capitolo 2](#), la gerarchia della memoria comprende la cache, la memory management unit (MMU), la random access memory (RAM), in questo capitolo chiamata semplicemente *memoria*, e un disco. Affronteremo la gestione della memoria di un SO in due parti: questo capitolo mostrerà le tecniche per l'uso efficiente della memoria, mentre nel prossimo capitolo verrà affrontata la gestione della *memoria virtuale*, ovvero quella parte della gerarchia di memoria costituita dalla memoria e dal disco.

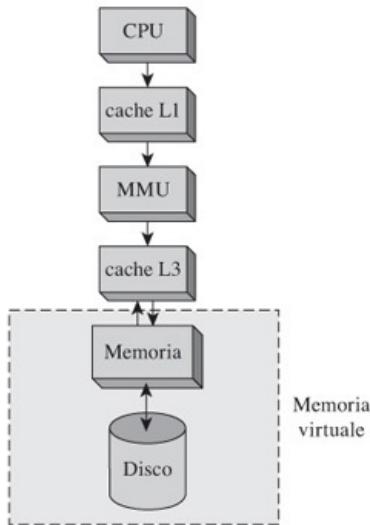
Il *memory binding* è l'associazione degli indirizzi di memoria con le istruzioni e i dati di un programma. Per fornire convenienza e flessibilità, il memory binding viene eseguito diverse volte in un programma – il compilatore e il linker lo eseguono *staticamente*, ovvero prima che il programma cominci l'esecuzione, mentre il SO lo esegue *dinamicamente*, ovvero durante l'esecuzione del programma. Il kernel alloca memoria a un processo utilizzando un modello che fornisce sia binding statici che dinamici.

La velocità di allocazione della memoria e l'uso efficiente della memoria sono due aspetti fondamentali nella progettazione di un allocatore di memoria. Per assicurare l'uso efficiente, il kernel riutilizza la memoria rilasciata da un processo assegnandola ad altri processi che la richiedono. La *frammentazione della memoria* è un problema che si verifica nel riutilizzo della memoria e che porta a un uso non efficiente. Mostriremo le tecniche per ridurre la quantità di memoria frammentata in un SO, in particolare *l'allocazione non contigua della memoria* utilizzando la *paginazione* o la *segmentazione*.

Il kernel crea e distrugge le strutture dati utilizzate per memorizzare i dati di controllo – principalmente, i vari blocchi di controllo come i PCB – a un tasso molto elevato. Le dimensioni di queste strutture dati sono note *a priori*, per cui il kernel utilizza un insieme di tecniche che sfruttano queste informazioni per ottenere una veloce allocazione/deallocazione e l'uso efficiente della memoria.

### 11.1 Gestione della gerarchia di memoria

Come discusso precedentemente nel [Capitolo 2](#), una gerarchia di memoria comprende la memoria cache, per esempio L1 e L3, la memory management unit (MMU), la memoria e un disco. Lo scopo è di creare l'illusione di una memoria veloce e grande a un costo basso. La parte superiore della [Figura 11.1](#) illustra la gerarchia di memoria. La CPU fa riferimento alla memoria più veloce, la *cache*, quando deve accedere a un'istruzione o a un dato. Se l'istruzione o il dato richiesto non è disponibile in cache, viene prelevato dal livello successivo della gerarchia di memoria, che potrebbe essere una cache più lenta o la *memoria* (RAM). Se l'istruzione o il dato richiesto non è disponibile nemmeno al livello successivo della gerarchia di memoria, viene prelevato da un livello inferiore e così via. Le prestazioni di un processo dipendono dagli *hit ratio* ai vari livelli della gerarchia di memoria, dove l'hit ratio indica la frazione di istruzioni o dati effettivamente presenti. L'Equazione 2.1 del [Capitolo 2](#) indica come il tempo effettivo di accesso alla memoria dipenda dall'hit ratio.



| Livelli | Gestione  | Prestazioni  |
|---------|---|--|
| Cache   | Allocazione e uso gestiti in hardware   | Garantire hit ratio elevati  |
| Memoria | Allocazione gestita dal kernel ed utilizzo della memoria allocata gestita dalla libreria run-time | (1) Mantenere più processi in memoria, (2) Garantire hit ratio elevati                     |
| Disco   | Allocazione ed uso gestiti dal kernel   | Caricamento e memorizzazione veloci di parti dello spazio di indirizzamento di un processo |

**Figura 11.1** Gestione della gerarchia di memoria.

Le cache sono gestite interamente in hardware. Il kernel utilizza tecniche speciali per garantire hit ratio elevati a un processo. Per esempio, il kernel commuta tra i thread dello stesso processo per sfruttare la presenza di parti dello spazio di indirizzamento nella cache e utilizza lo scheduling per affinità in un sistema multiprocessore (Paragrafo 10.5), per schedulare un processo sempre sulla stessa CPU e ottenere hit ratio elevati.

La memoria viene gestita congiuntamente dal kernel e dalla *libreria run-time* del linguaggio di programmazione con cui è scritto il codice del processo. Il kernel alloca la memoria ai processi utente. L'aspetto principale delle prestazioni di questa funzione riguarda l'inserimento di più processi utente in memoria, così che sia le prestazioni del sistema che il servizio per l'utente vengano migliorati. Il kernel soddisfa questo requisito mediante il riutilizzo efficiente della memoria quando un processo termina l'esecuzione. Durante l'esecuzione, un processo crea strutture dati *all'interno* della memoria allocatagli dal kernel. Questa funzione di fatto viene effettuata dalla libreria run-time, adottando tecniche per riutilizzare in modo efficiente la memoria quando un processo crea e distrugge le strutture dati durante la propria esecuzione. Per questo motivo molti aspetti e tecniche utilizzate dal kernel e dalla libreria run-time sono simili.

Per mantenere in memoria un elevato numero di processi, il kernel può decidere di mantenere in memoria solo una parte dello spazio di indirizzamento di ogni processo. A tal fine si utilizza la parte della gerarchia della memoria chiamata *memoria virtuale* che si compone della memoria e del disco (contenitore tratteggiato in Figura 11.1). Le parti dello spazio di indirizzamento di un processo non presenti in memoria vengono caricate dal disco quando necessario. In questa organizzazione, l'hit ratio di un processo in memoria determina le sue prestazioni. Dunque il kernel utilizza un insieme di tecniche per garantire ai processi un elevato hit ratio. Il disco nella memoria virtuale viene gestito interamente dal kernel; il kernel memorizza parti differenti dello spazio di indirizzamento di ogni processo sul disco in maniera tale che vi si possa accedere in maniera efficiente. Questo contribuisce alle buone prestazioni dei processi utilizzando la memoria virtuale.

Discuteremo la gestione della gerarchia di memoria da parte di un sistema operativo in

due parti. Questo capitolo si focalizza sulla gestione della memoria e sulle tecniche adottate per un uso efficiente della memoria e per l'allocazione e la deallocazione veloce della memoria. Successivamente discuteremo di come la presenza della memory management unit (MMU) semplifichi entrambe queste funzioni. Il [Capitolo 12](#) affronta la gestione della memoria virtuale, con particolare riguardo alle tecniche adottate dal kernel per assicurare hit ratio elevati in memoria e limitare la memoria assegnata a ogni processo.

## 11.2 Allocazione statica e dinamica della memoria

L'allocazione della memoria è un aspetto di un'azione più generale del funzionamento del software conosciuta come *binding*. Altre due aspetti del binding sono il linking e il caricamento.

Un'entità in un programma, per esempio una funzione o una variabile, ha un insieme di attributi e ogni attributo ha un valore. Il binding consiste nello specificare il valore di un attributo. Per esempio, una variabile in un programma ha attributi come il nome, il tipo, la dimensione, l'ambiente e l'indirizzo di memoria. Un binding del nome specifica il nome della variabile e un binding del tipo specifica il suo tipo. Il memory binding serve a specificare l'indirizzo di memoria di una variabile; consiste nell'allocazione della memoria per la variabile. L'allocazione delle memorie a un processo consiste nell'attribuire gli indirizzi di memoria alle istruzioni e ai dati.

Il binding per un attributo di un'entità come una funzione o una variabile può essere eseguito in qualsiasi momento prima dell'utilizzo dell'attributo. Metodi differenti di binding eseguono il binding in momenti differenti. Il momento esatto nel quale viene eseguito il binding può determinare l'efficienza e la flessibilità con cui l'entità può essere utilizzata. In generale, possiamo distinguere tra early binding e late binding. Il late binding è utile nei casi in cui il SO o la libreria run-time possono ottenere più informazioni riguardanti un'entità in un momento successivo e con queste eseguire un binding migliore. Per esempio, il SO può utilizzare in maniera efficiente le risorse come la memoria. I binding early e late sono rappresentati dai due metodi fondamentali di binding, rispettivamente, *statico* e *dinamico*.

**Definizione 11.1 Binding statico** Un binding eseguito prima dell'esecuzione di un programma (o dell'attivazione di un software).

**Definizione 11.2 Binding dinamico** Un binding eseguito durante l'esecuzione di un programma (o dell'attivazione di un software).

*L'allocazione statica della memoria* può essere eseguita da un compilatore, da un linker o da un loader quando un programma è pronto per l'esecuzione. *L'allocazione dinamica delle memorie* viene eseguita in maniera "pigra" durante l'esecuzione di un programma; la memoria viene allocata a una funzione o a una variabile subito prima di venire usata per la prima volta.

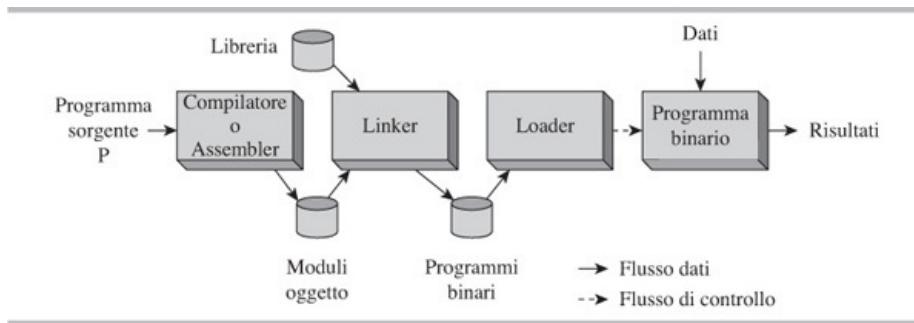
L'allocazione statica della memoria a un processo è possibile solo se la dimensione delle sue strutture dati è nota prima che inizi l'esecuzione. Se la dimensione non è nota, deve essere stimata; una stima errata può portare a uno spreco di memoria e alla mancanza di flessibilità. Per esempio, si consideri un array la cui dimensione non è nota al momento della compilazione. Se la dimensione dell'array è sovrastimata la memoria viene sprecata, mentre il processo potrebbe non funzionare correttamente se la dimensione fosse sottostimata. L'allocazione dinamica della memoria può evitare entrambi questi problemi allocando un'area di memoria la cui dimensione coincide con la dimensione attuale dell'array, che sarebbe nota al momento dell'allocazione. Inoltre può consentire di modificare la dimensione dell'array durante l'esecuzione del processo. Tuttavia, l'allocazione dinamica genera overhead dovuto alle azioni di allocazione della memoria eseguite durante l'esecuzione del processo.

I sistemi operativi scelgono l'allocazione statica e dinamica della memoria a seconda delle circostanze per ottenere la migliore combinazione di efficienza di esecuzione ed efficienza nell'utilizzo della memoria. Quando sono disponibili sufficienti informazioni riguardanti i requisiti di memoria, il kernel o la libreria run-time effettuano staticamente

le decisioni relative all'allocazione della memoria, cosa che fornisce efficienza dell'esecuzione. Quando sono disponibili poche informazioni a priori, le decisioni relative all'allocazione della memoria sono effettuate dinamicamente, cosa che genera un maggiore overhead ma assicura un utilizzo efficiente della memoria. In altre situazioni, l'informazione disponibile viene utilizzata per prendere decisioni riguardanti l'allocazione della memoria staticamente, così che l'overhead dell'allocazione dinamica della memoria possa essere ridotto. Discuteremo un esempio di questo approccio nel Paragrafo 11.11, dove il kernel sfrutta la conoscenza delle proprie strutture dati per ottenere un'allocazione efficiente della memoria.

### 11.3 Esecuzione dei programmi

Un programma  $P$  scritto in un linguaggio  $L$  deve essere trasformato prima di essere eseguito. Molte di queste trasformazioni effettuano il memory binding; ognuna effettua il collegamento delle istruzioni e i dati del programma a un nuovo insieme di indirizzi. La [Figura 11.2](#) mostra uno schema delle tre trasformazioni eseguite sul programma  $P$  prima di poter essere caricato in memoria per l'esecuzione.



**Figura 11.2** Schema della trasformazione e dell'esecuzione di un programma.

- *Compilazione o assemblaggio*: un compilatore o un assembler è chiamato genericamente *traduttore*. Traduce il programma  $P$  in un programma equivalente nella forma di *modulo oggetto*. Questo programma contiene le istruzioni in linguaggio macchina del computer. Nell'invocare il compilatore, l'utente specifica *l'origine* del programma, ovvero l'indirizzo della sua prima istruzione o del primo byte; altrimenti, il compilatore assume un indirizzo di default, tipicamente 0. Il compilatore di conseguenza assegna gli indirizzi alle altre istruzioni e dati del programma e utilizza questi indirizzi come indirizzi degli operandi nelle sue istruzioni. *L'indirizzo di avvio dell'esecuzione* o semplicemente *start address* di un programma è l'indirizzo dell'istruzione con la quale comincia la sua esecuzione. Può coincidere con l'origine del programma o può essere differente. Gli indirizzi assegnati dal compilatore sono chiamati *indirizzi tradotti*. In questo modo, il compilatore collega le istruzioni e i dati del programma  $P$  agli indirizzi tradotti. Un modulo oggetto indica l'origine tradotta del programma, l'indirizzo d'avvio tradotto e la dimensione.
- *Linking*: il programma  $P$  richiama altri programmi durante la sua esecuzione, per esempio le funzioni delle librerie matematiche. Queste funzioni dovrebbero essere incluse nel programma e i loro indirizzi d'avvio dovrebbero essere utilizzati nelle istruzioni chiamata a funzione presenti in  $P$ . Questa procedura viene chiamata *linking*. Si ottiene selezionando i moduli oggetto relativi alle funzioni richiamate da una o più librerie e unendoli al programma  $P$ .
- *Rilocazione*: alcuni moduli oggetto uniti al programma  $P$  possono avere indirizzi tradotti che sono in conflitto nel tempo. Questo conflitto viene risolto cambiando il binding di memoria dei moduli oggetto; quest'azione viene chiamata *rilocazione* dei moduli oggetti e richiede la modifica degli indirizzi degli operandi usati nelle loro istruzioni.

Le funzioni di rilocazione e linking sono effettuate da un programma chiamato *linker*. Gli indirizzi assegnati dal linker sono chiamati *indirizzi linkati*. L'utente può specificare l'origine linkata del programma; altrimenti, il linker assume che l'origine linkata sia la stessa dell'origine tradotta. Sulla base dell'origine linkata e della rilocazione necessaria per evitare conflitti negli indirizzi, il linker collega le istruzioni e i dati del programma a un insieme di indirizzi linkati. Il programma risultante, chiamato *programma binario*,

viene memorizzato in una libreria. La directory della libreria memorizza il suo nome, l'origine linkata, la dimensione e l'indirizzo di avvio linkato.

Per l'esecuzione, il programma binario deve essere caricato in memoria. Questa funzione viene eseguita dal *loader*. Se l'indirizzo d'avvio dell'area di memoria dove caricare un programma, chiamata *origine di caricamento*, differisce dall'origine linkata del programma, il loader deve modificare nuovamente il memory binding. Un loader che possiede questa funzionalità viene definito *loader assoluto*. Va notato che i compilatori, i linker e i loader non fanno parte del SO. In questo paragrafo discuteremo differenti tipi di programmi e le loro proprietà relative ai binding di memoria, all'esecuzione del linker e ai requisiti durante l'esecuzione. Useremo programmi scritti in un semplice linguaggio assembly per illustrare le azioni di rilocazione e linking eseguite dal linker.

### ***Un semplice linguaggio assembly***

Un'istruzione di linguaggio assembly ha il seguente formato:

[Label]    <Opcode>    <operando>, <operando>

Il primo operando è sempre un registro general purpose (GPR), per esempio AREG, BREG, CREG o DREG. Il secondo operando è un GPR oppure un nome simbolico che corrisponde a un byte di memoria. Gli opcode (o codici operativi) ADD e MULT vengono utilizzati per determinare operazioni aritmetiche. L'istruzione MOVER sposta un valore dall'operando in memoria all'operando in un registro, mentre l'istruzione MOVEM esegue l'operazione inversa. Tutte le operazioni aritmetiche sono eseguite in un registro e impostano un *codice di condizione*. Il codice di condizione può essere controllato con l'istruzione branch-on-condition (BC). L'istruzione assembly corrispondente ha il formato:

BC    <codice di condizione>, <indirizzo dell'istruzione>

dove <codice di condizione> è una stringa di caratteri che descrive la condizione, per esempio, GT per > e EQ for =. L'istruzione BC trasferisce il controllo all'istruzione memorizzata all'indirizzo <indirizzo dell'istruzione> se il valore attuale del codice di condizione coincide con <codice di condizione>. Per semplicità, assumiamo che tutti gli indirizzi e le costanti siano espresse in decimale e tutte le istruzioni occupino 4 byte. Il segno non è parte dell'istruzione. L'opcode e gli operandi di un'istruzione occupano, rispettivamente, 2, 1 e 3 cifre e i GPR AREG, BREG, CREG, e DREG sono rappresentati, rispettivamente, da 1, 2, 3 e 4 in una istruzione.

#### **11.3.1 Rilocazione**

La [Figura 11.3](#) mostra il programma P, un programma assembly e il suo codice generato. Le istruzioni ENTRY e EXTRN si riferiscono al linking; sono discusse successivamente nel Paragrafo 11.3.2. Un'istruzione DS semplicemente riserva il numero di byte passato come operando. L'istruzione START 500 indica che l'origine tradotta del programma dovrebbe essere 500. L'indirizzo tradotto di LOOP è quindi 504. L'indirizzo di A è 540. Le istruzioni in byte con indirizzi 532 e 500 utilizzano questi indirizzi per riferirsi, rispettivamente a LOOP e A. Questi indirizzi ovviamente dipendono dall'origine del programma. Le istruzioni che utilizzano questi indirizzi sono chiamate *istruzioni address-sensitive*. Un programma che contiene istruzioni address-sensitive può essere eseguito correttamente solo se viene caricato nell'area di memoria il cui indirizzo d'avvio coincide con l'origine del programma. Se deve essere eseguito in qualche altra area di memoria, gli indirizzi delle istruzioni address-sensitive devono essere modificati. Questa azione è chiamata *rilocazione* e richiede la conoscenza delle origini tradotte e linkate e le informazioni relative alle istruzioni address-sensitive. Il prossimo esempio illustra la rilocazione di P.

| <i>Istruzione assembly</i> |                   | <i>Codice generato</i> |               |
|----------------------------|-------------------|------------------------|---------------|
|                            |                   | <i>Indirizzo</i>       | <i>Codice</i> |
|                            | START 500         |                        |               |
|                            | ENTRY TOTAL       |                        |               |
|                            | EXTRN MAX, ALPHA  |                        |               |
|                            | READ A            | 500)                   | 09 0 540      |
| LOOP                       |                   | 504)                   |               |
|                            | :                 |                        |               |
|                            | MOVER AREG, ALPHA | 516)                   | 04 1 000      |
|                            | BC ANY, MAX       | 520)                   | 06 6 000      |
|                            | :                 |                        |               |
|                            | BC LT, LOOP       | 532)                   | 06 1 504      |
|                            | STOP              | 536)                   | 00 0 000      |
| A                          | DS 1              | 540)                   |               |
| TOTAL                      | DS 3              | 541)                   |               |
|                            | END               |                        |               |

**Figura 11.3** Programma assembly P e il suo codice generato.

### Esempio 11.1 Rilocazione di un programma

L'origine tradotta del programma P in [Figura 11.3](#) è 500. L'indirizzo tradotto del simbolo A è 540. L'istruzione corrispondente a *READ A* è un'istruzione address-sensitive. Se l'origine linkata di P è 900, l'indirizzo linkato di A è 940. Può essere ottenuto aggiungendo la differenza tra le origini tradotte e linkate, ovvero, 900-500, al suo indirizzo tradotto. In questo modo, la rilocazione può essere effettuata aggiungendo 400 all'indirizzo utilizzato in ogni istruzione address-sensitive. Dunque, l'indirizzo nell'istruzione *READ* verrebbe modificato a 940. In modo analogo, l'istruzione nel byte di memoria tradotta 532 utilizza l'indirizzo 504, che è l'indirizzo di *LOOP*. Questo indirizzo viene posto a 904. (Va notato che anche gli indirizzi degli operandi nelle istruzioni con indirizzi 516 e 520 devono essere "corretti". Tuttavia, questo è un esempio di linking che sarà discusso nel prossimo paragrafo.)

### Rilocazione statica e dinamica dei programmi

Quando si deve eseguire un programma, il kernel gli assegna un'area di memoria grande abbastanza da contenerlo e richiama il loader con il nome del programma e l'origine di caricamento come parametri. Il loader carica il programma nella memoria allocatagli, lo riloca utilizzando lo schema illustrato nell'Esempio 11.1 se l'origine linkata è differente dall'origine di caricamento e gli passa il controllo per l'esecuzione. In questo caso si parla di rilocazione statica poiché viene eseguita prima che abbia inizio l'esecuzione del programma. A un certo punto, il kernel può voler cambiare l'area di memoria allocata al programma in modo da poter caricare in memoria altri programmi. Questa volta, la rilocazione deve essere effettuata durante l'esecuzione del programma, dunque si parla di rilocazione dinamica.

La rilocazione dinamica può essere effettuata sospendendo l'esecuzione del programma, eseguendo la procedura di rilocazione descritta in precedenza e riprendendo l'esecuzione. Tuttavia, durante l'esecuzione del programma è necessario avere informazioni relative all'origine tradotta e alle istruzioni address-sensitive. Inoltre si è soggetti ai costi di memoria ed elaborazione descritti in precedenza. Alcune architetture forniscono un *registro di rilocazione* per semplificare la rilocazione dinamica. Il *registro di rilocazione* è uno speciale registro della CPU il cui contenuto viene sommato a ogni indirizzo di memoria utilizzato durante l'esecuzione di un programma. Il risultato è un altro indirizzo di memoria, che viene utilizzato per fare riferimento alla memoria. In questo modo,

$$\begin{aligned} \text{indirizzo di memoria effettivo} &= \text{indirizzo di memoria utilizzato} \\ &\quad \text{nell'istruzione corrente} \\ &\quad + \text{contenuto del registro di rilocazione} \end{aligned}$$

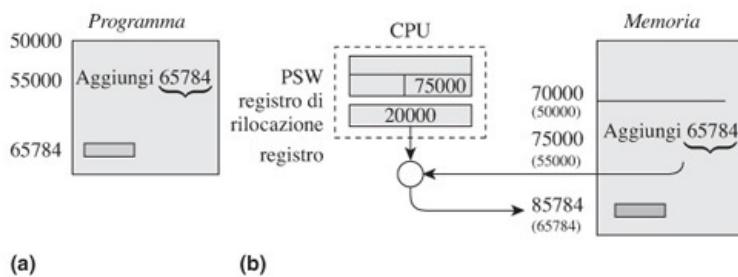
L'esempio seguente illustra l'uso del registro di rilocazione per implementare la rilocazione dinamica.

### Esempio 11.2 Rilocazione dinamica mediante il registro di rilocazione

Un programma ha origine linkata a 50 000 ed è stato caricato nell'area di memoria con l'indirizzo di avvio pari a 50 000. Durante l'esecuzione, deve essere spostato nell'area di memoria che ha l'indirizzo di avvio pari a 70 000, per cui deve essere rilocato. Questa rilocazione è ottenuta semplicemente caricando un valore appropriato nel registro di rilocazione, calcolato come segue:

$$\begin{aligned} &\text{valore da caricare nel registro di rilocazione} \\ &= \text{indirizzo di avvio dell'area di memoria allocata} - \text{origine linkata del programma} \\ &= 70\,000 - 50\,000 = 20\,000 \end{aligned}$$

Si consideri l'esecuzione dell'istruzione ADD nel programma mostrato in [Figura 11.4\(a\)](#). Questa istruzione ha indirizzo linkato nel programma 55 000 e utilizza un operando il cui indirizzo linkato è 65 784. Come risultato della rilocazione, il programma si trova nell'area di memoria che inizia all'indirizzo 70 000. La [Figura 11.4\(b\)](#) mostra gli indirizzi di caricamento delle istruzioni e dei dati; i corrispondenti indirizzi linkati sono mostrati tra parentesi per renderne semplice il riferimento. L'istruzione ADD si trova alla locazione con indirizzo 75 000. L'indirizzo del suo operando è 65 784 e il registro di rilocazione contiene 20 000, per cui durante l'esecuzione dell'istruzione, l'indirizzo effettivo del suo operando è  $65\,784 + 20\,000 = 85\,784$ . Dunque l'accesso effettivo alla memoria viene eseguito all'indirizzo 85 784.



**Figura 11.4** Rilocazione di un programma utilizzando un registro di rilocazione: (a) programma; (b) sua esecuzione.

### 11.3.2 Linking

Un'istruzione **ENTRY** in un programma assembly indica i simboli che sono definiti nel programma e ai quali è possibile riferire in altri programmi assembly. Questi simboli sono chiamati *entry point*. Un'istruzione **EXTRN** in un programma assembly indica i simboli utilizzati nel programma assembly ma definiti in altri programmi assembly. Questi simboli sono chiamati simboli esterni e il loro utilizzo in un programma assembly è detto *riferimento esterno*. L'assembler mette le informazioni relative alle istruzioni **ENTRY** e **EXTRN** in un modulo oggetto per essere utilizzato dal linker.

Il *linking* è il processo con cui si collega un riferimento esterno al corretto indirizzo linkato. Inizialmente il linker esamina tutti i moduli oggetto linkati insieme per collezionare i nomi di tutti gli entry point e i loro indirizzi linkati e memorizza queste informazioni in una tabella. Successivamente considera ogni riferimento esterno, ottiene dalla tabella l'indirizzo linkato del simbolo esterno al quale fa riferimento e inserisce questo indirizzo nell'istruzione che contiene il riferimento esterno. Il prossimo esempio illustra i passi del linking.

### Esempio 11.3 Linking

L'istruzione **ENTRY TOTAL** nel programma P di [Figura 11.3](#) indica che **TOTAL** è un entry point del programma. Si noti che **LOOP** e **A** non sono entry point sebbene siano definiti nel programma. L'istruzione **EXTRN MAX, ALPHA** indica che il programma contiene riferimenti esterni a **MAX** e **ALPHA**. L'assembler non conosce gli indirizzi di **MAX** e **ALPHA** durante l'elaborazione del programma P, per cui inserisce degli zero nei campi relativi agli indirizzi degli operandi dell'istruzione che contiene i riferimenti a questi simboli ([Figura 11.3](#)).

Si consideri il programma Q mostrato di seguito:

|       |            |       | Codice Generato |            |
|-------|------------|-------|-----------------|------------|
|       | Istruzione |       | Indirizzo       | Codice     |
|       | START      | 200   |                 |            |
|       | ENTRY      | ALPHA |                 |            |
|       | ---        |       |                 |            |
| ALPHA | DC         | 25    | 232)            | + 00 0 025 |
|       | END        |       |                 |            |

L'indirizzo DC dichiara una costante 25. Il simbolo ALPHA è un entry point in Q e il suo indirizzo tradotto è 232. Sia 900 l'origine linkata del programma P di [Figura 11.3](#). La dimensione di P è 44 byte, per cui il linker assegna l'indirizzo 944 all'origine linkata di Q. Di conseguenza, l'indirizzo linkato di ALPHA è  $232 - 200 + 944 = 976$ . Il linker risolve il riferimento esterno ad ALPHA nel programma P inserendo l'indirizzo 974 nel campo relativo all'indirizzo dell'operando dell'istruzione che utilizza ALPHA, ovvero nell'istruzione di P con indirizzo tradotto 516. Questa istruzione ha indirizzo linkato 916.

### ***Linking/loading statico e dinamico***

La differenza tra i termini linking e loading è diventata sempre meno netta nei moderni sistemi operativi. Tuttavia, usiamo i termini nel seguente modo: un *linker* collega insieme i moduli per formare un programma eseguibile. Un *loader* carica un programma o una parte di programma in memoria per l'esecuzione.

Nel *linking statico*, il linker collega tutti i moduli di un programma prima che cominci la sua esecuzione; produce un programma binario che non contiene nessun riferimento esterno non assegnato. Se più programmi utilizzano lo stesso modulo di una libreria, ogni programma riceverà una propria copia del modulo; diverse copie del modulo potranno essere presenti in memoria allo stesso tempo se i programmi che usano il modulo vengono eseguiti simultaneamente.

Il *linking dinamico* viene eseguito durante l'esecuzione di un programma binario. Il linker viene invocato quando, durante l'esecuzione, si incontra un riferimento esterno non assegnato. Il linker risolve il riferimento esterno e riprende l'esecuzione del programma. Questa organizzazione ha diversi vantaggi riguardanti l'uso, la condivisione e l'aggiornamento dei moduli delle librerie. I moduli non invocati durante l'esecuzione non vengono linkati. Se un modulo riferito da un programma è già stato linkato a un altro programma in esecuzione, la stessa copia del modulo può essere linkata a questo programma, risparmiando memoria. Il linking dinamico inoltre fornisce un interessante vantaggio quando si aggiorna una libreria di moduli: un programma che invoca il modulo della libreria comincia automaticamente a utilizzare la nuova versione del modulo! Le librerie linkate dinamicamente (dynamically linked libraries - DLL) utilizzano alcune di queste caratteristiche.

Per facilitare il linking dinamico, ogni programma viene prima elaborato dal linker statico. Il linker statico collega ogni riferimento esterno presente nel programma a un modulo fittizio la cui unica funzione è di richiamare il linker dinamico e passargli il nome del simbolo esterno. In questo modo, il linker dinamico viene attivato quando, durante l'esecuzione del programma, si fa riferimento ad un simbolo esterno. Il linker mantiene una tabella degli entry point e il loro indirizzo di caricamento. Se il simbolo esterno è presente nella tabella, utilizza l'indirizzo di caricamento del simbolo per risolvere il riferimento esterno. Altrimenti, cerca nella libreria dei moduli oggetto un modulo che contenga il simbolo richiesto come entry point. Questo modulo oggetto viene linkato al programma binario attraverso lo schema illustrato nell'[Esempio 11.3](#) e l'informazione relativa agli entry point è aggiunta alla tabella del linker.

### **[11.3.3 Tipologie di programmi utilizzate nei sistemi operativi](#)**

Due caratteristiche di un programma influenzano la sua esecuzione da parte del SO:

- il programma può essere eseguito in ogni area di memoria o deve essere eseguito in un'area di memoria specifica?
- il codice del programma può essere condiviso da più utenti in maniera concorrente?

Se l'origine di caricamento del programma non coincide con l'indirizzo di avvio dell'area di memoria, il programma deve essere rilocato prima di poter essere eseguito. Questa

operazione è costosa. Risulta pertanto vantaggioso che un programma possa essere eseguito in ogni area di memoria. La condivisione di un programma è importante se il programma può essere usato da più utenti allo stesso tempo. Se un programma non è condivisibile, ogni utente deve avere una copia del programma, per cui diverse copie del programma dovranno risiedere in memoria allo stesso tempo.

La [Tabella 11.1](#) riassume le tipologie di programmi utilizzate nei sistemi operativi. Un modulo oggetto è una tipologia di programma che può essere rilocato da un linker, mentre un programma binario non può essere rilocato da un linker.

| Tipologia di programma          | Caratteristiche  |
|---------------------------------|--|
| Modulo oggetto                  | Contiene istruzioni e dati di un programma e informazioni richieste per la sua rilocazione e linking.  |
| Programma binario               | Tipologia di programma pronto per essere eseguito.   |
| Programmi linkati dinamicamente | Il linkaggio è effettuato in maniera "lenta", ovvero un modulo oggetto che definisce un simbolo viene linkato al programma solo quando il simbolo viene riferito durante l'esecuzione del programma. |
| Programma auto-rilocante        | Il programma può rilocare se stesso per essere eseguito in qualsiasi area di memoria.  |
| Programma rientrante            | Il programma può essere eseguito su diversi insiemi di dati in maniera concorrente.  |

**Tabella 11.1** Tipologie di programmi utilizzate nei sistemi operativi.

La tipologia di programma linkato dinamicamente consente di risparmiare memoria linkando solo quei moduli oggetto ai quali si fa riferimento durante l'esecuzione. Nelle sezioni precedenti abbiamo discusso queste tre tipologie di programma. Un *programma auto-rilocante* può essere eseguito in qualsiasi parte della memoria. Questa tipologia di programma non è importante quando un computer utilizza il registro di rilocazione o la memoria virtuale. La tipologia di *programma rientrante* evita la necessità di avere in memoria più copie di un programma. Queste due tipologie di programma verranno discusse nei paragrafi successivi.

### **Programmi auto-rilocanti**

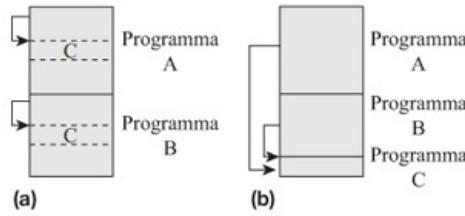
Si ricordi dal Paragrafo 11.3.1 che la rilocazione di un programma coinvolge modifiche delle sue istruzioni address-sensitive tali che il programma possa essere eseguito correttamente in una specifica area di memoria. La rilocazione di un programma da parte del linker richiede che sia disponibile il suo modulo oggetto; inoltre genera un notevole overhead. La tipologia di programma auto-rilocante fu sviluppata per eliminare questi problemi; esegue da solo la rilocazione per adattarsi all'area di memoria assegnatagli.

Un programma auto-rilocante conosce la sua origine tradotta e gli indirizzi tradotti delle sue istruzioni address-sensitive. Inoltre dispone di una *logica di rilocazione*, ovvero codice che esegue la rilocazione. L'indirizzo d'avvio della logica di rilocazione viene specificata come indirizzo d'avvio del programma, per cui la logica di rilocazione ottiene il controllo quando il programma viene caricato per l'esecuzione. L'esecuzione inizia richiamando una funzione fittizia. L'indirizzo di ritorno costruito da questa chiamata di funzione è l'indirizzo della prossima istruzione. Utilizzando questo indirizzo, si ottiene l'indirizzo dell'area di memoria dove viene caricato per l'esecuzione, ovvero la sua origine di caricamento. A questo punto ha a disposizione tutte le informazioni necessarie per implementare lo schema di rilocazione del Paragrafo 11.3.1. Dopo aver eseguito la propria rilocazione, passa il controllo alla prima istruzione per cominciare l'esecuzione.

### **Programmi rientranti**

I programmi possono essere condivisi sia in maniera statica che in maniera dinamica. Si considerino due programmi A e B che utilizzano un programma C. Indichiamo A e B come programmi *che condividono* e C come programma *condiviso*. La condivisione statica di C è eseguita utilizzando il linking statico. Dunque il codice e i dati di C sono inclusi sia in A che in B; l'identità di C viene persa nei programmi binari prodotti dal linker. Se i programmi A e B sono eseguiti simultaneamente, esisteranno due copie di C in memoria

[Figura 11.5(a)]. In questo modo, la condivisione statica di un programma è semplice da implementare, ma può sprecare memoria.



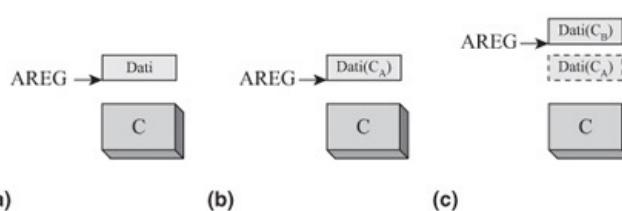
**Figura 11.5** Condivisione di un programma C da parte dei programmi A e B: (a) condivisione statica; (b) condivisione dinamica.

Quando si utilizza la condivisione dinamica, una singola copia del codice di un programma condiviso viene caricata in memoria e utilizzata da tutti i programmi in esecuzione che condividono quel programma. La condivisione dinamica viene implementata utilizzando il linking dinamico. Il kernel tiene traccia dei programmi condivisi in memoria.

Quando un programma vuole usare uno dei programmi condivisi, il kernel linka dinamicamente il programma alla copia in memoria del programma condiviso. La Figura 11.5(b) illustra la condivisione dinamica. Quando il programma A ha bisogno di usare il programma C in modo condiviso, il kernel rileva che C non esiste in memoria. Dunque carica una copia di C in memoria e dinamicamente la linka ad A. In Figura 11.5(b), questo linkaggio è rappresentato da una freccia da A a C. Quando un programma B ha bisogno di usare il programma C, il kernel verifica che una copia di C già esiste in memoria, per cui linka questa copia a B. Questa organizzazione evita la necessità di avere copie multiple di un programma in memoria, ma bisogna garantire che le esecuzioni concorrenti di un programma non interferiscano l'una con l'altra.

Un *programma rientrante* è un programma che può essere eseguito in maniera concorrente da molti utenti senza reciproca interferenza. Quando è invocato, il programma rientrante alloca una nuova copia delle sue strutture dati e carica gli indirizzi di memoria di questa copia in un registro general purpose (GPR). Il suo codice accede alle strutture dati attraverso il GPR. In questo modo, se un programma rientrante viene invocato concorrentemente da molti programmi, le invocazioni concorrenti utilizzano copie differenti delle strutture dati.

La Figura 11.6 illustra l'esecuzione del programma C implementato come programma rientrante. Il programma C è implementato in modo da assumere che AREG punti all'inizio della sua area dati [Figura 11.6(a)]. L'accesso agli elementi in quest'area viene eseguito utilizzando differenti offset a partire dall'indirizzo contenuto in AREG. Quando il programma A chiama C, C alloca un'area dati da utilizzare durante questa invocazione. Viene rappresentato come Dati(C<sub>A</sub>) in Figura 11.6(b). Quando A viene prelazionato, il contenuto di AREG viene memorizzato nel PCB di A; sarà ricaricato in AREG quando A sarà schedulato di nuovo. Quando C è richiamato da B, in modo analogo viene allocata un'area dati Dati(C<sub>B</sub>) e AREG è impostato per puntare a quest'area [Figura 11.6(c)]. In questo modo le esecuzioni dei programmi A e B non interferiscono l'una con l'altra.



**Figura 11.6** (a) Struttura di un programma rientrante; (b)-(c) invocazioni concorrenti del programma.

## 11.4 Allocazione della memoria a un processo

### 11.4.1 Stack e heap

Il compilatore di un linguaggio di programmazione genera il codice di un programma, alloca i suoi dati statici e crea un modulo oggetto del programma (Paragrafo 11.3). Il linker linka il programma con le funzioni di libreria e il supporto run-time del linguaggio di programmazione, prepara un eseguibile del programma e lo memorizza in un file. L'informazione relativa alla dimensione del programma è memorizzata nell'elemento della directory relativo all'eseguibile.

Il supporto run-time alloca due tipi di dati durante l'esecuzione del programma. Il primo tipo di dati include le variabili il cui ambiente è associato alle funzioni, le procedure o i blocchi di un programma e i parametri delle chiamate di funzione o procedura. Questi dati sono allocati quando si opera all'interno di una funzione, procedura o blocco e sono deallocati al termine della relativa esecuzione. A causa del modo di allocazione/deallocazione last-in first-out, i dati sono allocati sullo stack. Il secondo tipo di dati viene creato dinamicamente da un programma utilizzando le funzioni del linguaggio come l'istruzione *new* del Pascal, C++ o Java, o le funzioni *malloc*, *calloc* del C. Ci riferiamo a tali dati come *dati dinamici controllati dal programma* (in breve, dati PCD, *program-controlled dynamic*). I dati PCD sono allocati utilizzando una struttura dati chiamata *heap*.

#### Stack

In uno *stack*, le allocazioni e le deallocazioni sono eseguite secondo la modalità last-in first-out (LIFO) in corrispondenza, rispettivamente, alle operazioni *push* e *pop*. Si assume che ogni elemento dello stack abbia dimensione standard, per esempio, 1 byte. Ogni volta si può accedere solo all'ultimo elemento dello stack. Un'area di memoria contigua viene riservata allo stack. Un puntatore chiamato *stack base* (SB) punta al primo elemento dello stack, mentre un puntatore chiamato *top of stack* (TSO) punta all'ultimo elemento allocato nello stack. Useremo la convenzione che uno stack cresce verso gli indirizzi bassi di memoria; nelle figure cresce verso l'alto.

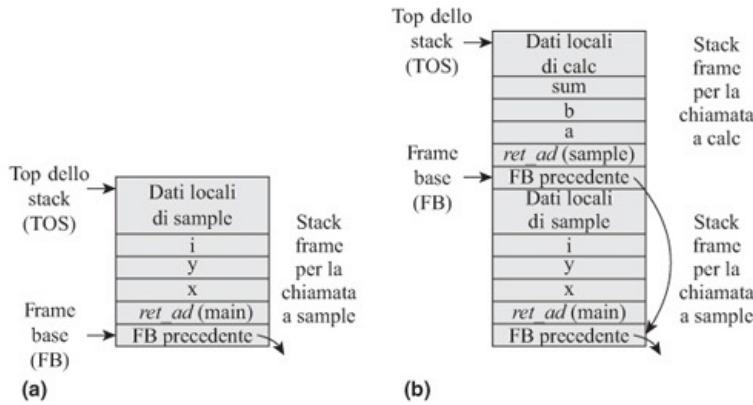
Durante l'esecuzione di un programma, lo stack è utilizzato per coadiuvare le chiamate di funzione. L'insieme di elementi dello stack che fanno riferimento a una chiamata di funzione è chiamato *stack frame*; nella terminologia dei compilatori è anche chiamato *record di attivazione* (activation record). Quando viene richiamata una funzione, il relativo stack frame è inserito nello stack. Per iniziare, lo stack frame contiene gli indirizzi o i valori dei parametri della funzione e l'indirizzo di ritorno, ovvero l'indirizzo dell'istruzione cui restituire il controllo una volta terminata l'esecuzione della funzione. Durante l'esecuzione della funzione, il supporto run-time del linguaggio di programmazione in cui è implementato il programma crea i dati locali della funzione all'interno dello stack frame. Al termine dell'esecuzione della funzione, l'intero stack frame viene estratto dallo stack e l'indirizzo di ritorno contenuto al suo interno è utilizzato per passare il controllo al programma chiamante.

Per facilitare l'uso degli stack frame vengono utilizzati due elementi: il primo elemento in uno stack frame è un puntatore al precedente stack frame. Questo elemento facilita l'estrazione di uno stack frame. Inoltre si usa un puntatore chiamato *frame base* (FB) per puntare all'inizio dello stack frame presente al top dello stack. Questo puntatore aiuta ad accedere ai vari elementi nello stack frame. L'Esempio 11.4 illustra l'utilizzo dello stack per implementare le chiamate di funzione.

#### Esempio 11.4 Uso di uno stack

La Figura 11.7 mostra lo stack durante l'esecuzione di un programma costituito di chiamate di funzione annidate. La Figura 11.7(a) mostra lo stack dopo che *main*, la prima funzione del programma, ha eseguito la chiamata della funzione *sample* (*x*, *y*, *i*). Uno stack frame era stato inserito sullo stack al momento della chiamata. Il primo elemento dello stack frame contiene il valore precedente del frame base, ovvero un puntatore al precedente stack frame nello stack. Il secondo elemento è *ret\_ad(main)*, ovvero l'indirizzo di ritorno della funzione *main*. I successivi tre elementi riguardano i parametri *x*, *y* e *i*, mentre gli elementi seguenti riguardano i dati locali della funzione *sample*. Il frame base (FB) punta al primo elemento in questo stack frame. Il puntatore TSO punta all'ultimo dato locale nello stack frame. Il codice della funzione *sample* accede all'indirizzo di ritorno, alle informazioni relative ai parametri e ai dati locali utilizzando gli scostamenti dal frame base (FB): assumendo che ogni elemento dello

stack sia di 4 byte, l'indirizzo di ritorno si trova a distanza 4 dall'indirizzo nel frame base, il primo parametro è a distanza 8, ecc.



**Figura 11.7** Lo stack dopo che (a) *main* chiama la funzione *sample*; (b) *sample* chiama la funzione *calc*.

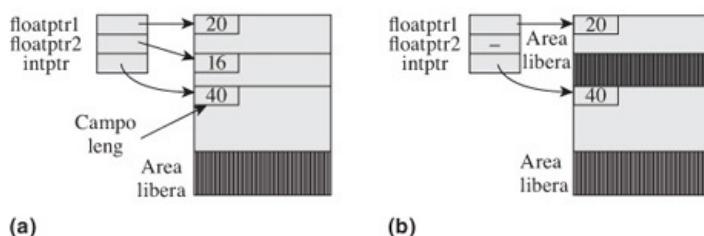
La Figura 11.7(b) mostra lo stack dopo che la funzione *sample* ha eseguito una chiamata della funzione *calc(a, b, sum)*. Un nuovo stack frame è stato inserito sullo stack, il valore dell'FB è stato salvato nel primo elemento di questo stack frame, l'FB è stato impostato per puntare all'inizio del nuovo stack frame e il puntatore al top dello stack ora punta all'ultimo elemento all'interno del nuovo stack frame. Al termine della funzione, il puntatore TSO viene impostato per puntare all'elemento dello stack che precede l'elemento puntato da FB ed FB viene caricato con l'indirizzo contenuto nell'elemento dello stack cui puntava. Queste azioni eseguono il pop dello stack frame di *calc* e impostano FB per puntare all'inizio dello stack frame di *sample*. Lo stack risultante è identico allo stack prima che la funzione *sample* chiamasse *calc*.

### Heap

Un *heap* consente di allocare e deallocare la memoria in ordine casuale. Una richiesta di allocazione da parte di un processo ritorna un puntatore all'area di memoria allocata nell'heap e il processo accede all'area di memoria allocata attraverso questo puntatore. Una richiesta di deallocazione deve fornire un puntatore all'area di memoria da deallocare. L'Esempio 11.5 illustra l'uso dell'heap per gestire i dati PCD di un processo. Come illustrato, si creano buchi nell'allocazione della memoria man mano che si creano e si rilasciano le strutture dati. L'allocatore dell'heap deve riutilizzare queste aree di memoria libere per soddisfare le future richieste di memoria.

### Esempio 11.5 Uso di un heap

La Figura 11.8 mostra lo stato di un heap dopo l'esecuzione del seguente programma C:



**Figura 11.8** (a) Un heap; (b) Un “buco” dovuto alla deallocazione della memoria.

```

float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc(5, sizeof(float));
floatptr2 = (float *) calloc(4, sizeof(float));
intptr = (int *) calloc(10, sizeof(int));
free(floatptr2);

```

La routine `calloc` viene utilizzata per effettuare una richiesta di memoria. La prima chiamata richiede memoria sufficiente a memorizzare 5 numeri floating point. L'allocatore dell'heap alloca un'area di memoria e ritorna un puntatore a essa. Il puntatore viene memorizzato in `floatptr1`. Si assume che i primi byte di ogni area di memoria allocata contengano un campo `length`. Questo campo viene usato durante la deallocazione al momento della chiamata a `free` con un puntatore a un'area di memoria allocata. La [Figura 11.8\(a\)](#) mostra l'heap dopo che tutte le chiamate `calloc` sono state elaborate. La [Figura 11.8\(b\)](#) mostra l'heap dopo la chiamata `free`. `free` ha liberato l'area di memoria puntata da `floatptr2`. Quest'azione ha creato un "buco" nell'allocazione.

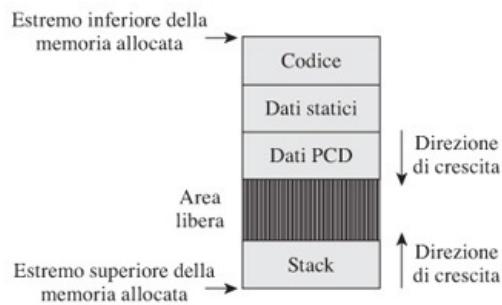
### 11.4.2 Il Modello di allocazione della memoria

Quando un utente esegue un comando per eseguire un programma il kernel crea un nuovo processo. A questo punto, deve decidere quanta memoria allocare alle seguenti componenti:

- codice e dati statici del programma;
- stack;
- dati dinamici controllati da programma (dati PCD).

La dimensione del programma può essere ottenuta dall'elemento relativo al programma all'interno della directory. Le dimensioni dello stack e dei dati PCD variano durante l'esecuzione di un programma, per cui il kernel non conosce quanta memoria allocare a queste componenti. Può ipotizzare le dimensioni massime che possono raggiungere lo stack e l'heap e allocarli di conseguenza. Tuttavia, questa è un'allocazione statica, che pecca di flessibilità. Come discusso nel Paragrafo 11.2, la memoria allocata può essere sprecata oppure un processo può richiedere una maggiore quantità di memoria durante la sua esecuzione.

Per evitare di dover affrontare questi problemi singolarmente per queste due componenti, i sistemi operativi utilizzano il modello di allocazione della memoria mostrato in [Figura 11.9](#). Alle componenti codice e dati statici del programma vengono allocate aree di memoria che corrispondono esattamente alla loro dimensione. I dati PCD e lo stack condividono una sola grande area di memoria, ma crescono in direzioni opposte quando la memoria è allocata a nuovi dati. I dati PCD sono allocati partendo dalla estremità bassa di quest'area mentre lo stack è allocato partendo dalla estremità alta dell'area. La memoria tra queste due componenti non è utilizzata. Può essere usata per creare nuovi dati in entrambe le componenti. In questo modello non esistono restrizioni sulla dimensione delle componenti stack e dati PCD.



**Figura 11.9** Modello di allocazione della memoria per un processo.

Un programma crea o distrugge i dati PCD chiamando le routine appropriate della libreria run-time del linguaggio di programmazione con il qual è implementato. Le

routine di libreria eseguono l'allocazione/deallocazione nell'area dei dati PCD allocata al processo. Dunque il kernel non è coinvolto in questo tipo di gestione della memoria e tali azioni appaiono trasparenti al kernel.

### 11.4.3 Protezione della memoria

Come discusso nel Paragrafo 2.2.3, la protezione della memoria è implementata mediante due registri di controllo della CPU chiamati *registro base* e *registro ampiezza*. Questi registri contengono, rispettivamente, l'indirizzo di avvio dell'area di memoria allocata a un processo e la sua dimensione. L'hardware per la protezione della memoria genera un interrupt di *violazione di protezione della memoria* se un indirizzo di memoria utilizzato nell'istruzione corrente di un processo risiede al di fuori dei limiti degli indirizzi definiti dal contenuto dei registri base e ampiezza (Figura 2.5). Nell'elaborare tale interrupt, il kernel termina il processo. I registri base e ampiezza costituiscono il campo relativo alle informazioni di protezione della memoria (MPI) della program status word (PSW). Il kernel carica i valori appropriati in questi registri quando schedula un processo per l'esecuzione. Un processo utente, eseguito con la CPU in modalità utente, non può modificare il contenuto di questi registri poiché le istruzioni per caricare e salvare questi registri sono istruzioni privilegiate.

Quando si utilizza un *registro di rilocazione* (Paragrafo 11.3.1), i controlli relativi alla protezione della memoria diventano più semplici se ogni programma ha l'origine linkata a 0. In Figura 2.5, il confronto con l'indirizzo contenuto nel registro base può essere omesso poiché l'indirizzo usato in un'istruzione non può essere  $< 0$ . L'hardware di protezione della memoria controlla semplicemente se un indirizzo è minore del contenuto del registro ampiezza. Il registro di rilocazione e il registro ampiezza costituiscono il campo MPI del PSW.

## 11.5 Gestione dell'heap

### 11.5.1 Riuso della memoria

La velocità dell'allocazione della memoria e l'uso efficiente sono i due problemi principali nella progettazione di un allocatore di memoria. L'allocazione basata su stack affronta entrambi questi problemi in maniera efficiente dal momento che l'allocazione e la deallocazione sono molto veloci - l'allocatore modifica solo i puntatori SB, FB e TSO per gestire la memoria libera e allocata (Paragrafo 11.4.1) - e la memoria rilasciata viene riutilizzata automaticamente per le nuove allocazioni. Tuttavia, l'allocazione basata su stack non può essere usata per dati allocati e rilasciati in maniera non ordinata. Per questo motivo gli allocatori dell'heap sono utilizzati dal supporto run-time del linguaggio di programmazione per gestire i dati PCD e dal kernel per gestire le proprie richieste di memoria.

In un heap, il riuso della memoria non è automatico; l'allocatore dell'heap deve tentare di riutilizzare un'area di memoria libera mentre esegue nuove allocazioni. Tuttavia, la dimensione di una richiesta di memoria raramente coincide con la dimensione di un'area di memoria precedentemente utilizzata, per cui alcune aree di memoria non sono considerate quando si effettua una nuova allocazione. Quest'area di memoria sarà sprecata se è troppo piccola per soddisfare una richiesta di memoria, per cui l'allocatore deve selezionare con cura l'area di memoria da allocare alla richiesta rallentando le sue operazioni. A causa dell'effetto combinato della non usabilità delle aree di memoria piccole e della memoria usata dall'allocatore per le sue strutture dati, un allocatore di heap può non essere in grado di assicurare un'efficienza elevata nell'utilizzo della memoria.

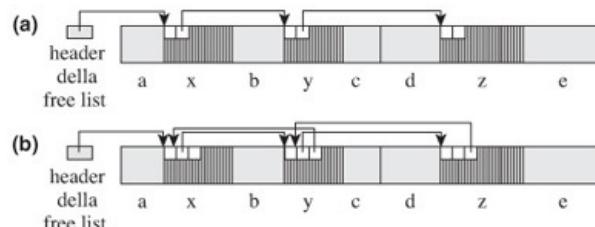
Il kernel utilizza le tre funzioni descritte Tabella 11.2 per garantire il riuso efficiente della memoria. Il kernel mantiene una *free list* per gestire le informazioni relative alle aree di memoria libere nel sistema. Una richiesta di memoria viene soddisfatta utilizzando l'area di memoria libera considerata più adatta per la richiesta e la memoria non utilizzata di quest'area di memoria viene inserita nella free list. La politica di allocazione prevede la creazione di aree di memoria troppo piccole da non poter essere utilizzate. Il kernel prova a unire le aree di memoria libere in aree di maggiori dimensioni in modo da soddisfare richieste di memoria maggiori.

| Funzione   | Descrizione  |
|--|--|
| Mantenere una free list                          | La <i>free list</i> contiene informazioni relative a ogni area di memoria libera. Quando un processo libera parti di memoria, le informazioni relative alla memoria liberata vengono inserite nella free list. Quando un processo termina, ogni area di memoria a esso allocata e le informazioni relative vengono inserite nella free list. |
| Selezionare un'area di memoria per l'allocazione | Quando viene effettuata una nuova richiesta di memoria, il kernel seleziona l'area di memoria più adatta per soddisfare la richiesta.  |
| Unire le aree di memoria libere                  | Due o più aree di memoria libere contigue possono essere unite per formare una sola grande area libera. Le aree da unire sono rimosse dalla free list e viene inserita al loro posto l'area più grande creata.   |

**Tabella 11.2** Funzioni del kernel per il riutilizzo della memoria.

### Mantenere una free list

Il kernel ha bisogno di mantenere due elementi di informazione di controllo per ogni area di memoria nella free list: la dimensione dell'area di memoria e i puntatori utilizzati per creare la lista. Per evitare di incorrere nell'overhead di memoria causato da queste informazioni di controllo, il kernel le memorizza nei primi byte di un'area di memoria libera. La [Figura 11.10\(a\)](#) mostra una *free list con singolo puntatore* in un heap che contiene cinque aree in uso (a–e) e tre aree libere (x–z). Ogni area di memoria nella free list contiene la sua dimensione e un puntatore alla prossima area di memoria nella lista. Questa organizzazione è semplice, ma richiede molto lavoro quando un'area di memoria deve essere inserita o cancellata dalla lista. Per esempio, la cancellazione di un'area di memoria dalla lista richiede una modifica nel puntatore memorizzato nell'area di memoria che la precede nella lista. L'inserimento di un'area di memoria in una posizione specifica della lista richiede un'operazione simile. Per questo motivo, le operazioni di inserimento e cancellazione in una lista con singolo puntatore sono eseguite scorrendo la lista a partire dalla testa. La complessità è dell'ordine di  $m$ , dove  $m$  è il numero di aree di memoria nella free list.



**Figura 11.10** Gestione dello spazio libero: (a) free list con singolo puntatore; (b) free list con doppio puntatore.

Al fine di facilitare operazioni di inserimento e cancellazioni veloci, può essere utilizzata una *free list con doppio puntatore*. Ogni elemento nella lista contiene due puntatori: uno punta alla prossima area di memoria nella lista, mentre l'altro punta all'area di memoria precedente [[Figura 11.10\(b\)](#)]. Se un'area di memoria con un indirizzo specifico deve essere cancellata dalla lista, il kernel può semplicemente prendere i puntatori alle aree di memoria precedente e successiva nella lista e modificare i puntatori in queste aree per eseguire la cancellazione. Operazioni analoghe sono sufficienti ad aggiungere nuove aree di memoria in una posizione specifica della lista. In questo caso la complessità necessaria per l'inserimento e la cancellazione delle aree di memoria è costante, indipendentemente dal numero di aree di memoria nella free list.

### Tecniche di allocazione mediante free list

Possono essere utilizzate tre tecniche per eseguire l'allocazione della memoria utilizzando una free list:

- tecnica first-fit;
- tecnica best-fit;
- tecnica next-fit.

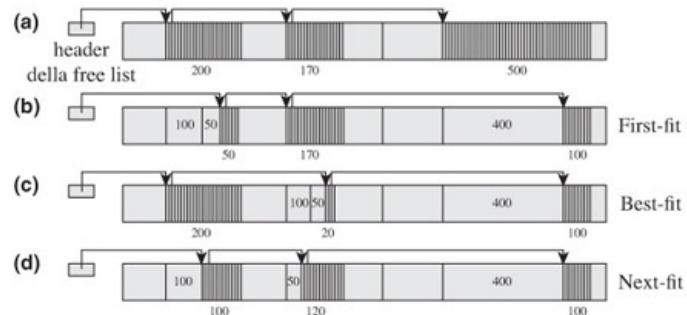
Per servire una richiesta per  $n$  byte di memoria, la tecnica *first-fit* utilizza la prima area di memoria libera con dimensione  $\geq n$  byte. Divide quest'area di memoria in due parti:  $n$  byte sono allocati per la richiesta e la parte restante dell'area di memoria, se presente, viene reinserita nella free list. Questa tecnica può dividere l'area di memoria all'inizio della free list ripetutamente, per cui le aree di memoria diventano più piccole col tempo. Di conseguenza, l'allocatore potrebbe nel tempo non trovare grandi aree di memoria libere per soddisfare richieste di grandi aree di memoria. Inoltre, diverse aree di memoria libera possono diventare troppo piccole per essere utilizzate.

La tecnica *best-fit* utilizza l'area di memoria più piccola con dimensione  $\geq n$ . In questo modo, evita la frammentazione di grandi aree di memoria, tuttavia tende a generare una piccola area di memoria a ogni divisione. Dunque nel lungo periodo anche questa tecnica può soffrire del problema delle numerose aree di memoria libera di piccole dimensioni. La tecnica *best-fit* inoltre genera un overhead di allocazione maggiore poiché oppure deve scorrere tutta la free list per ogni allocazione oppure deve mantenere la free list in ordine crescente di dimensione.

La tecnica *next-fit* ricorda quale elemento della free list è stato usato per l'ultima allocazione. Per eseguire una nuova allocazione, scorre la free list a partire dall'elemento successivo ed esegue l'allocazione utilizzando la prima area di memoria con dimensione  $\geq n$  byte. In questo modo, si evita la divisione ripetuta della stessa area come avviene nella tecnica *first-fit* e inoltre si evita l'overhead di allocazione delle tecnica *best-fit*.

### Esempio 11.6 Allocazioni first, best e next-fit

La free list di [Figura 11.11\(a\)](#) contiene tre aree di memoria libera di dimensioni, rispettivamente, 200, 170 e 500 byte. I processi eseguono richieste di allocazione per 100, 50 e 400 byte. La tecnica *first-fit* allocherà 100 e 50 byte dalla prima area della free list, lasciando in questo modo un'area di memoria libera di 50 byte e alloca 400 byte dalla terza area di memoria. La tecnica *best-fit* allocherà 100 e 50 byte dalla seconda area di memoria, lasciando un'area di memoria libera di 20 byte. La tecnica *next-fit* alloca 100, 50 e 400 byte dalle tre aree di memoria.



**Figura 11.11** (a) Free list; (b)-(d) allocazione mediante first-fit, best-fit e next-fit.

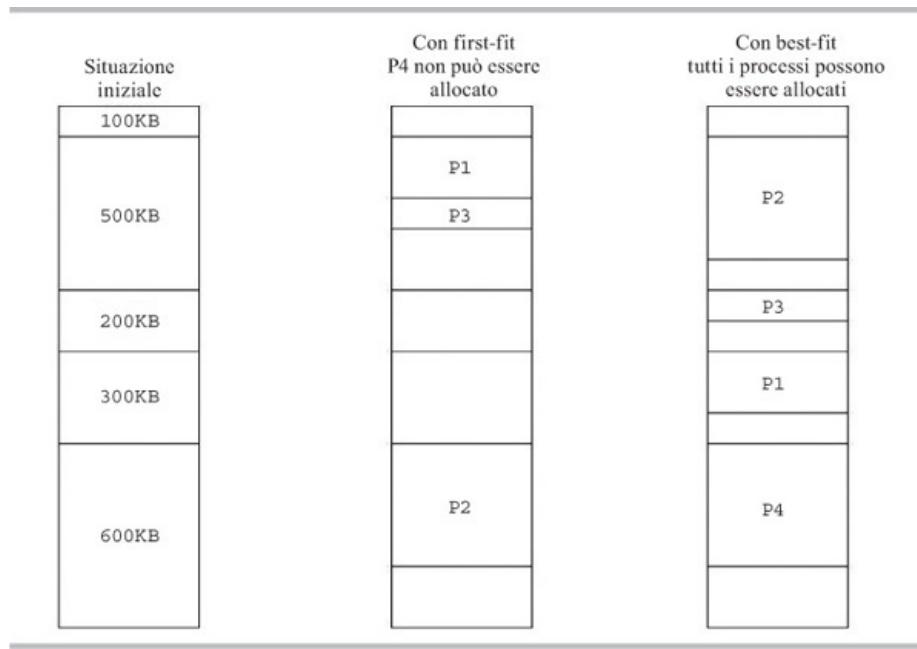
Knuth (1973) presenta dati sperimentali sul riutilizzo della memoria e conclude che sia il *first-fit* che il *next-fit* hanno prestazioni migliori del *best-fit*. Tuttavia, il *next-fit* tende a dividere *tutte* le aree libere se il sistema resta in esecuzione abbastanza a lungo, mentre il *first-fit* può non dividere le ultime aree libere. Questa proprietà del *first-fit* facilita l'allocazione di grandi aree di memoria.

### Esempio di allocazione della memoria

Si supponga che siano le seguenti partizioni di memoria: 100 KB, 500 KB, 200 KB, 300 KB e 600 KB. L'obiettivo è quello di descrivere come vengono disposti dagli algoritmi di *first-fit* e *best-fit* i processi di 2121 KB, 417 KB, 112 KB e 426 KB (in ordine).

La disposizione dei processi è riportata in [Figura 11.12](#), dove appare evidente che il quarto processo non può essere allocato in virtù della frammentazione prodotta dalla

tecnica *first-fit*. Nel seguito sarà chiarito il concetto di frammentazione della memoria e le sue implicazioni.



**Figura 11.12** Disposizione dei processi in memoria.

### Frammentazione della memoria

**Definizione 11.3 Frammentazione della memoria** Presenza di aree di memoria inutilizzabili nella memoria di un computer.

La [Tabella 11.3](#) descrive due forme di frammentazione della memoria. La *frammentazione esterna* si verifica quando un'area di memoria rimane inutilizzata poiché è troppo piccola per essere allocata. La *frammentazione interna* si verifica quando alcune aree di memoria allocate a un processo restano inutilizzate, cosa che accade quando a un processo viene allocata più memoria del necessario. In [Figura 11.11\(c\)](#), l'allocazione best-fit crea un'area di memoria libera di 20 byte, che risulta troppo piccola per essere allocata. Questo è un esempio di frammentazione esterna. Si ha frammentazione interna se un allocatore deve allocare, per esempio 100 byte di memoria quando un processo ne richiede 50 byte; questo problema si verifica se un allocatore gestisce esclusivamente blocchi di memoria di poche dimensioni prefissate per limitare il suo overhead.

| Tipologia di frammentazione | Descrizione   |
|-----------------------------|---|
| Frammentazione esterna      | Alcune aree di memoria sono troppo piccole per essere allocate.   |
| Frammentazione interna      | Viene allocata più memoria rispetto a quella richiesta, dunque una parte della memoria allocata resta inutilizzata. |

**Tabella 11.3** Tipologie di frammentazione della memoria.

La frammentazione della memoria porta a un utilizzo non efficiente della memoria. In questo paragrafo e nella parte restante del capitolo, mostremo diverse tecniche per evitare o minimizzare la frammentazione della memoria.

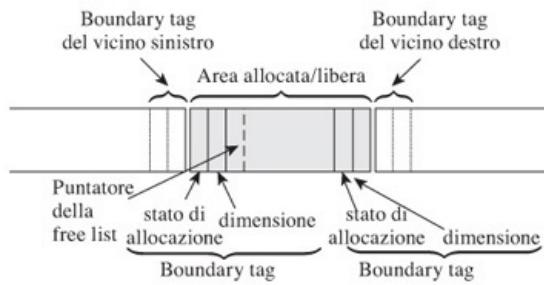
### Unione delle aree di memoria libera

La frammentazione esterna può essere evitata unendo le aree di memoria libera per formare aree di memoria più grandi. L'unione può essere tentata ogni volta che una

nuova area di memoria viene aggiunta alla free list. Un semplice metodo sarebbe quello di scorrere la free list per verificare se un'area contigua è già nella free list. In caso affermativo, può essere rimossa dalla free list e unita con la nuova area per creare un'area di memoria libera più grande. Quest'azione può essere ripetuta finché non è più possibile eseguire altre unioni e l'area di memoria libera risultante può essere aggiunta alla free list. Tuttavia, questo metodo è costoso poiché coinvolge la ricerca nella free list ogni volta che viene liberata una nuova area di memoria. Verranno descritte due tecniche generali che eseguono l'unione in maniera più efficiente; nel Paragrafo 11.5.2 descriveremo una speciale tecnica di unione utilizzata nell'allocatore buddy system.

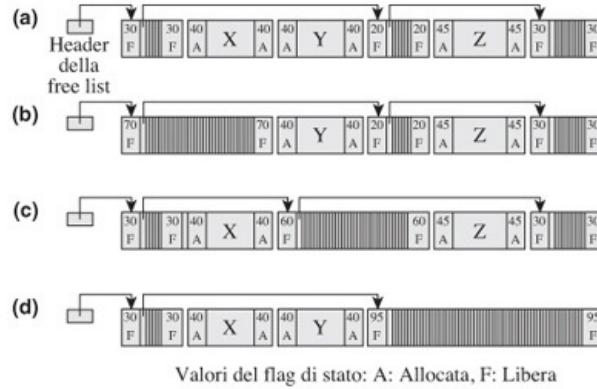
### **Boundary tag**

Un *tag* è un descrittore di stato di un'area di memoria. Consiste di una coppia ordinata che rappresenta lo stato di allocazione dell'area e la sua dimensione. I boundary tag sono tag identici memorizzati all'inizio e alla fine dell'area di memoria, ovvero nei primi e negli ultimi byte dell'area. Se un'area di memoria è libera, il puntatore della free list può essere inserito dopo il boundary tag di inizio. La [Figura 11.13](#) mostra questa organizzazione.



**Figura 11.13** I boundary tag e il puntatore della free list.

Quando un'area di memoria diventa libera, il kernel controlla i boundary tag delle aree vicine. Questi tag sono facili da trovare poiché precedono e seguono immediatamente la nuova area liberata. Se qualcuno dei vicini è libero, viene unito con l'area appena liberata. La [Figura 11.14](#) mostra le azioni da eseguire quando le aree di memoria X, Y e Z sono liberate, mentre un sistema che utilizza i boundary tag si trova nella situazione raffigurata in [Figura 11.14\(a\)](#). In [Figura 11.14\(b\)](#), l'area di memoria X viene liberata. Solo il suo vicino di sinistra è libero per cui X è unito a quest'ultimo. I boundary tag sono ora impostati per l'area creata. Il vicino di sinistra già era presente nella free list, per cui è sufficiente modificare il suo campo dimensione. Solo il vicino di destra di Y è libero. Dunque quando Y è liberato, viene unito al suo vicino di destra e i boundary tag sono impostati per l'area unita. In questo caso la free list deve essere modificata per rimuovere l'elemento relativo al vicino di destra e aggiungere un elemento per l'area creata [[Figura 11.14\(c\)](#)]. Entrambi i vicini dell'area di memoria Z sono liberi. Dunque quando Z viene rilasciato, viene unito a entrambi per formare una singola area libera. Il campo dimensione dell'elemento relativo al vicino di sinistra nella free list è modificato come conseguenza dell'unione. Poiché il vicino di destra è già presente nella free list, questa viene modificata per rimuovere l'elemento relativo [[Figura 11.14\(d\)](#)]. Quando l'unione si verifica con il vicino di destra, la gestione della free list richiede una complessità di ordine di  $m$ , dove  $m$  è il numero di elementi della free list.



**Figura 11.14** Unione mediante boundary tag: (a) free list; (b)-(d) rilascio, rispettivamente, delle aree X, Y e Z.

Come indicato in precedenza nel Paragrafo 11.5.1, mantenere la free list come lista a doppio puntatore consente di eseguire questa operazione in modo efficiente.

Quando si usa questo metodo di unione, viene applicata una relazione chiamata *regola del 50 per cento*. Quando un'area di memoria viene rilasciata, il numero totale di aree libere nel sistema aumenta di 1, diminuisce di 1 o non cambia in base al fatto che l'area di memoria rilasciata ha una, due o nessun'area di memoria libera vicina. Queste aree di memoria sono mostrate, rispettivamente, come aree di tipo B, A e C, di seguito:



Quando si esegue un'allocazione, il numero di aree di memoria libere si riduce di 1 se la dimensione richiesta coincide con la dimensione di qualche area libera; altrimenti, non viene modificato dal momento che la restante area libera sarebbe inserita nuovamente nella free list. Assumendo una memoria grande abbastanza da poter ignorare la situazione alle estremità della memoria e assumendo che ogni area di memoria abbia la stessa probabilità di essere rilasciata, abbiamo:

$$\text{numero di aree allocate, } n = \#A + \#B + \#C$$

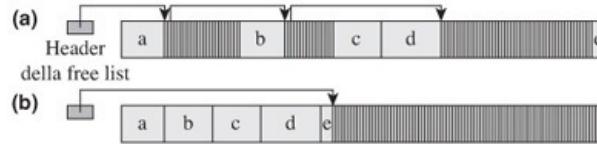
$$\text{numero di aree libere, } m = \frac{1}{2} (2 \times \#A + \#B)$$

dove  $\#A$  è il numero di aree libere di tipo A, e lo stesso vale per  $\#B$  e per  $\#C$ . Nello stato stabile  $\#A = \#C$ . Dunque  $m = n/2$ , ovvero il numero di aree libere è la metà del numero di aree allocate. Questa è la relazione chiamata regola del 50 per cento.

La regola del 50 per cento aiuta a stimare la dimensione della free list e, dunque, le risorse necessarie in un metodo di allocazione come il metodo best-fit che richiede l'analisi dell'intera free list. Inoltre fornisce un metodo per stimare l'area libera in memoria a ogni istante. Se  $s_f$  è la dimensione media di queste aree di memoria, la memoria libera totale è  $s_f \times n/2$ .

### Compattazione della memoria

In questo approccio i memory binding sono modificati in maniera tale che tutte le aree di memoria possano essere unite per formare un'unica area di memoria libera. Come suggerisce il nome, può essere ottenuta "impacchettando" tutte le aree di memoria allocata verso un'estremità della memoria. La [Figura 11.15](#) illustra la compattazione per l'unione delle aree libere.



**Figura 11.15** Compattazione della memoria.

La compattazione non è semplice come potrebbe apparire, poiché coinvolge spostamenti di codice e dati in memoria. Se l'area *b* in [Figura 11.15](#) contiene un processo, questo deve essere rilocato per essere eseguito correttamente nella nuova area di memoria. La rilocazione coinvolge la modifica di tutti gli indirizzi utilizzati da un processo, inclusi gli indirizzi dei dati allocati sull'heap e gli indirizzi contenuti nei registri general purpose. È fattibile solo se il computer utilizza un registro di rilocazione (Paragrafo 11.3.1); la rilocazione può essere ottenuta modificando semplicemente l'indirizzo contenuto nel registro di rilocazione.

### 11.5.2 Buddy system e allocatori potenza del 2

Il buddy system e gli allocatori potenza del 2 eseguono l'allocazione di memoria in blocchi di poche dimensioni predefinite. Questa caratteristica porta alla frammentazione interna poiché parte della memoria in ogni blocco allocato può essere sprecata. Tuttavia, consente all'allocatore di mantenere free list separate per blocchi di dimensioni differenti. Questa organizzazione evita ricerche costose nella free list e porta ad allocazioni e deallocazioni veloci.

#### Allocatore buddy

Un buddy system divide e ricombina i blocchi di memoria in modo predeterminato durante l'allocazione e la deallocazione. I blocchi creati dividendo un blocco sono chiamati *blocchi buddy*. I blocchi buddy liberi vengono uniti per ricreare il blocco da cui erano stati creati. Questa operazione si chiama *fusion*. Secondo questo sistema, i blocchi liberi contigui che non sono buddy non sono fusi. Il buddy system *binario*, che descriviamo qui, divide un blocco in due buddy di dimensione uguale. In questo modo ogni blocco *b* ha un singolo blocco buddy che precede *b* in memoria o lo segue. Le dimensioni del blocco di memoria sono  $2^n$  per differenti valori di  $n \geq t$ , dove  $t$  è un valore soglia. Questa restrizione assicura che i blocchi di memoria non diventino di dimensione troppo piccola.

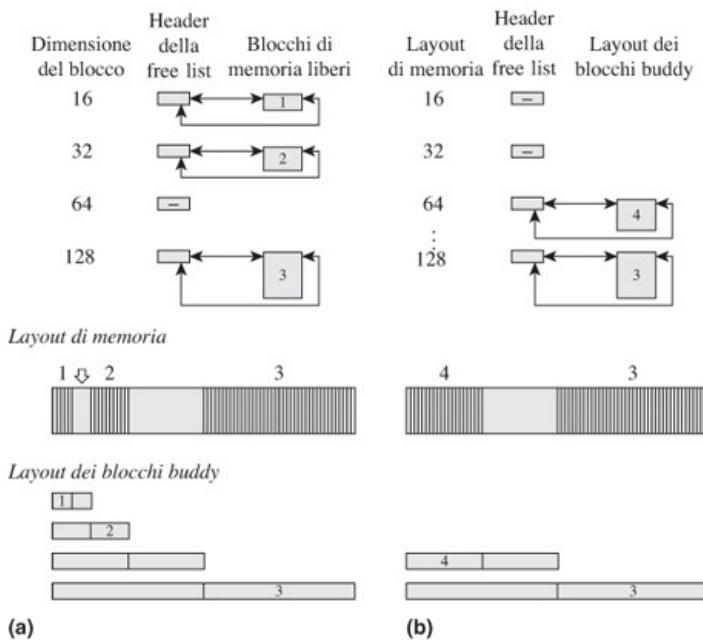
L'allocatore buddy system associa un tag di 1 bit ad ogni blocco per indicare se il blocco è *allocato* o *libero*. Il tag di un blocco può trovarsi nel blocco stesso o può essere memorizzato separatamente. L'allocatore mantiene molte liste di blocchi liberi; ogni free list viene memorizzata come lista a doppio puntatore e contiene blocchi liberi di uguale dimensione, ovvero blocchi di dimensione  $2^k$  per qualche  $k \geq t$ . Il funzionamento dell'allocatore comincia con un singolo blocco di memoria libera di dimensione  $2^z$ , per qualche  $z > t$ . Viene inserito nella free list dei blocchi di dimensione  $2^z$ . Quando un processo richiede un blocco di memoria di dimensione  $m$ ; vengono eseguite le azioni elencate di seguito. Il sistema cerca la più piccola potenza del 2 che sia maggiore o uguale a  $m$ . Sia questa  $2^i$ . Se la lista per blocchi con dimensione  $2^i$  è non vuota, allora il processo il primo blocco della lista e cambia il tag del blocco da *libero* ad *allocato*. Se la lista è vuota, controlla la lista per blocchi di dimensione  $2^{i+1}$ . Estrae un blocco dalla lista e lo divide in due metà di dimensione  $2^i$ . Questi blocchi diventano buddy. Inserisce uno di questi blocchi nella free list per blocchi di dimensione  $2^i$  e utilizza l'altro blocco per soddisfare la richiesta. Se un blocco di dimensione  $2^{i+1}$  non è disponibile, cerca nella lista per blocchi di dimensione  $2^{i+2}$ , ne divide uno per ottenere blocchi di dimensione  $2^{i+1}$ , divide ulteriormente uno di questi blocchi per ottenere blocchi di dimensione  $2^i$  e ne alloca uno, e così via. In questo modo, può essere necessario effettuare molte divisioni prima che una richiesta possa essere soddisfatta.

Quando un processo libera un blocco di memoria di dimensione  $2^i$ , il buddy system cambia il tag del blocco a *libero* e controlla il tag del suo blocco buddy per vedere se anche il blocco buddy è libero. In caso affermativo, unisce questi due blocchi in un

singolo blocco di dimensione  $2^{i+1}$ . A questo punto ripete il controllo della fusione in maniera transitiva; ovvero verifica se il buddy di questo nuovo blocco di dimensione  $2^{i+1}$  è libero, e così via. Inserisce un blocco in una free list solo quando trova che il suo blocco buddy non è libero.

### Esempio 11.7 Funzionamento di un buddy system

La Figura 11.16 illustra il funzionamento di un buddy system binario. Le parti (a) e (b) della figura mostrano lo stato del sistema prima e dopo che il blocco etichettato con il simbolo  $\downarrow$  venga rilasciato da un processo. In ogni parte mostriamo due viste del sistema. La metà superiore mostra la free list mentre la metà inferiore mostra il layout della memoria e i blocchi buddy. Per facilità di compressione, i blocchi corrispondenti nelle due metà hanno gli stessi numeri. Il blocco da rilasciare ha una dimensione di 16 byte. Il suo buddy è il blocco libero con numero 1 nella Figura 11.16(a), per cui l'allocatore buddy system unisce questi due blocchi per creare un nuovo blocco di 32 byte. Il buddy di questo nuovo blocco è il blocco 2, anch'esso libero. Per cui il blocco 2 viene rimosso dalla free list dei blocchi di 32 byte e unito al nuovo blocco per creare un blocco libero di dimensione 64 byte. Questo blocco libero ha il numero 4 in Figura 11.16(b). Viene ora inserito nella free list appropriata.



**Figura 11.16** Funzionamento del buddy system quando si rilascia un blocco.

Il controllo del tag di un buddy può essere eseguito in maniera efficiente poiché le dimensioni dei blocchi sono potenze del 2. Supponiamo la dimensione del blocco rilasciato 16 byte. Poiché 16 è  $2^4$ , il suo indirizzo è del tipo ...  $y0000$ , dove i quattro 0 seguono  $y$  e  $y$  è 0 o 1. Il suo blocco buddy ha l'indirizzo ...  $z0000$  dove  $z = 1 - y$ . Questo indirizzo può essere ottenuto semplicemente eseguendo un'operazione di or esclusivo con un numero ...  $10000$ , ovvero, con  $2^4$ . Per esempio, se l'indirizzo di un blocco è  $101010000$ , l'indirizzo del suo buddy è  $101000000$ . In generale, l'indirizzo del buddy di un blocco di dimensione  $2^n$  bytes può essere trovato eseguendo l'or esclusivo con  $2^n$ . Questo vantaggio si ottiene anche se i tag sono memorizzati separatamente in una bitmap (Problema 11.8).

### Allocatore potenza del 2

Come nel buddy system binario, le dimensioni dei blocchi di memoria sono potenze del 2 e vengono mantenute free list separate per blocchi di diverse dimensioni. Tuttavia, le similitudini con il buddy system terminano qui. Ogni blocco contiene un elemento header che, a sua volta, contiene l'indirizzo della free list alla quale dovrebbe essere aggiunto nel caso venga deallocated. Quando viene eseguita una richiesta per  $m$  byte, l'allocatore

dapprima controlla la free list contenente i blocchi la cui dimensione è  $2^i$  per il più piccolo valore di  $i$  tale che  $2^i \geq m$ . Se questa free list è vuota, controlla la lista contenente i blocchi con dimensione potenza del 2 successiva e così via. Un intero blocco è allocato a una richiesta, ovvero non viene eseguita nessuna divisione dei blocchi. Inoltre, non viene eseguita la fusione dei blocchi adiacenti per creare blocchi più grandi; quando un blocco viene rilasciato, viene semplicemente restituito alla sua free list.

Il funzionamento del sistema comincia creando blocchi della dimensione desiderata e inserendoli nelle free list appropriate. I nuovi blocchi possono essere creati dinamicamente sia quando l'allocatore non ha a disposizione più blocchi di una data dimensione sia quando una richiesta non può essere soddisfatta.

### 11.5.3 Confronto tra allocatori di memoria

Gli allocatori di memoria possono essere confrontati sulla base della velocità di allocazione e l'uso efficiente della memoria. Gli allocatori buddy e potenza del 2 sono più veloci degli allocatori first-fit, best-fit e next-fit poiché evitano le ricerche nelle free list. L'allocatore potenza del 2 è più veloce dell'allocatore buddy poiché non esegue la divisione e l'unione.

Per confrontare l'efficienza di utilizzo della memoria nei differenti allocatori di memoria, definiamo il fattore di utilizzo della memoria come segue:

$$\text{fattore di utilizzo della memoria} = \frac{\text{memoria in uso}}{\text{memoria totale impegnata}}$$

dove *memoria in uso* è la quantità di memoria utilizzata dai processi che richiedono memoria e *memoria totale impegnata* include la memoria allocata ai processi, la memoria libera presente nell'allocatore di memoria e la memoria occupata dalle strutture dati dell'allocatore. La memoria in uso può essere inferiore rispetto alla memoria allocata ai processi a causa della frammentazione interna e inferiore rispetto alla memoria totale impegnata a causa della frammentazione esterna. Il maggiore fattore di utilizzo della memoria rappresenta il caso migliore di un allocatore e il valore più basso in cui l'allocatore non riesce a soddisfare una richiesta rappresenta il suo caso peggiore.

Gli allocatori che utilizzano le tecniche first-fit, best-fit e next-fit non generano frammentazione interna. Tuttavia, la frammentazione esterna limita le loro prestazioni nel caso peggiore poiché i blocchi liberi possono essere troppo piccoli per soddisfare una richiesta (Problema 11.4). Gli allocatori buddy e potenza del due allocano blocchi le cui dimensioni sono potenze del 2, per cui esiste frammentazione interna a meno che le richieste di memoria non coincidano con le dimensioni dei blocchi. Questi allocatori utilizzano memoria aggiuntiva per memorizzare gli header delle free list e i tag o gli header per i blocchi. In un allocatore potenza del 2, l'elemento header in un blocco non può essere utilizzato da un processo. In questo modo la parte utile di un blocco è leggermente inferiore di una potenza del 2. Se una richiesta di memoria è per un'area che sia esattamente una potenza del 2, questo metodo utilizza fino a due volte quella quantità di memoria. Un allocatore potenza del 2 fallisce nel soddisfare una richiesta se non esiste un blocco libero sufficientemente grande. Poiché non unisce blocchi liberi in blocchi di dimensione maggiore, questa situazione può verificarsi anche quando la memoria libera totale disponibile in blocchi di dimensione inferiore eccede la dimensione della richiesta. In un buddy system questa situazione può verificarsi solo se blocchi liberi contigui non sono buddy. In pratica questo è raro. Infatti, Knuth (1973) riporta che in studi di simulazione le prestazioni nel caso migliore di un allocatore buddy sono del 95 per cento.

### 11.5.4 Gestione dell'heap in Windows

Il sistema operativo Windows utilizza un approccio per la gestione dell'heap che punta a fornire un basso overhead di allocazione e una bassa frammentazione. Di default, utilizza una free list e una politica di allocazione best-fit. Tuttavia, questa organizzazione non è adeguata per due tipi di situazioni: se un processo fa un utilizzo eccessivo dell'heap, potrebbe allocare e rilasciare ripetutamente aree di memoria di poche dimensioni specifiche, per cui risulta inutile l'overhead causato dalla politica best-fit e dall'unione di aree libere. In un ambiente multiprocessore, la free list può diventare un collo di bottiglia per le prestazioni (Paragrafo 10.3). Per cui in queste situazioni Windows utilizza

un'organizzazione chiamata *heap a bassa frammentazione* (low-fragmentation heap - LFH).

Il low-fragmentation heap mantiene molte free list, ognuna contenente le aree di memoria di una dimensione specifica. Le dimensioni delle aree di memoria sono multipli di 8 byte fino a 256 byte, multipli di 16 byte fino a 512 byte, multipli di 32 byte fino a 1 KB, dove 1 KB = 1024 byte, ecc. e multipli di 1 KB fino a 16 KB. Quando un processo richiede un'area di memoria minore di 16 KB di dimensione, un'area di memoria viene estratta dalla free list appropriata e allocata al processo. Non vengono effettuate né la divisione né l'unione di tali aree di memoria. Questa organizzazione è analoga a quella usata negli allocator potenza del 2, sebbene i blocchi *non* siano potenze del 2, per cui ne eredita sia i vantaggi sia gli svantaggi: l'allocazione della memoria è veloce, ma si crea frammentazione interna nell'area allocata. Per soddisfare le richieste di memoria che eccedono i 16 KB di dimensione, il gestore dell'heap mantiene una singola free list e alloca un'area di memoria la cui dimensione coincide esattamente con la richiesta.

Se il gestore dell'heap non riesce a trovare un'area di memoria sufficientemente grande in una free list per richieste <16 KB, passa la richiesta al nucleo del gestore dell'heap. Inoltre mantiene statistiche delle richieste e del modo in cui sono state soddisfatte quali per esempio il tasso al quale le aree di memoria di una specifica dimensione sono state richieste e un conteggio del numero di volte in cui non ha trovato aree di memoria della dimensione richiesta, e le utilizza per una messa a punto fine delle proprie prestazioni creando aree di memoria libera di dimensione appropriata in anticipo rispetto alle richieste effettive.

## 11.6 Allocazione contigua della memoria

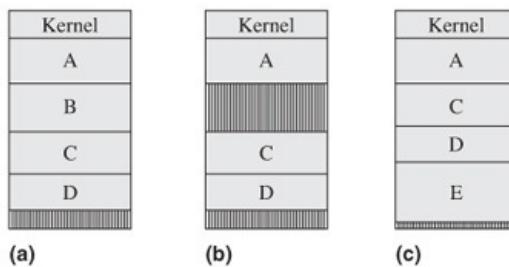
L'allocazione contigua della memoria è il classico modello di allocazione in cui a ogni processo viene allocata un'unica area di memoria contigua. In questo modo il kernel alloca un'area di memoria sufficientemente grande per contenere il codice, i dati, lo stack e i dati PCD di un processo come mostrato in [Figura 11.9](#). L'allocazione di memoria contigua affronta il problema della frammentazione della memoria. La *rilocazione* di un programma nell'allocazione di memoria contigua e la protezione della memoria sono state discusse precedentemente nei Paragrafi 11.3.1 e 11.4.3.

### Gestione della frammentazione della memoria

Abbiamo discusso le cause della frammentazione interna ed esterna precedentemente nel Paragrafo 11.5.1.3. La frammentazione interna non trova rimedio nell'allocazione contigua della memoria poiché il kernel non ha modo di stimare accuratamente le richieste di memoria di un processo. Per risolvere il problema della frammentazione esterna, possono essere adottate le tecniche di compattazione della memoria e riutilizzo della memoria discusse precedentemente nel Paragrafo 11.5. L'Esempio 11.8 illustra l'uso della compattazione della memoria.

### Esempio 11.8 Allocazione contigua della memoria

I processi A, B, C e D sono in memoria, come rappresentato in [Figura 11.17\(a\)](#). Ci sono due aree di memoria dopo che B è terminato; tuttavia nessuna delle due è abbastanza grande da contenere un altro processo [[Figura 11.17\(b\)](#)]. Il kernel esegue la compattazione per creare un'unica area di memoria libera e avvia il processo E in quest'area [[Figura 11.17\(c\)](#)]. Questo procedimento consiste nel muovere i processi C e B in memoria durante la loro esecuzione.



**Figura 11.17** Compattazione della memoria.

La compattazione della memoria coinvolge la *rilocazione dinamica*, che non può essere seguita senza un registro di rilocazione (Paragrafo 11.3.1). Nei computer che non hanno un registro di rilocazione, il kernel deve ricorrere al riuso delle aree di memoria libere. Tuttavia, questo approccio provoca ritardi nell'avvio dei processi quando non sono disponibili grandi aree di memoria libera o, equivalentemente, ovvero l'avvio del processo E sarebbe ritardato nell'Esempio 11.8 sebbene la memoria libera globale disponibile nel sistema sia sufficiente a contenere E.

### **Swapping**

Il meccanismo base dello swapping e la logica su cui si basa sono stati descritti nel Paragrafo 3.6.1. Il kernel riversa un processo che non si trova nello stato *running* ricopiando il suo codice e lo spazio dati in un'*area di swap* sul disco. Il processo swappato viene riportato in memoria prima che gli venga assegnato un altro burst di tempo di CPU.

Una problematica legata allo swapping riguarda la ricollocazione di un processo swappato sul disco nella stessa area di memoria che occupava prima di essere swappato. Se questo è il caso, la sua ricollocazione in memoria dipende dallo swap su disco di qualche altro processo al quale nel frattempo può essere stata allocata quell'area di memoria. Sarebbe utile poter riportare il processo in una qualsiasi altra area di memoria; tuttavia ciò implicherebbe l'utilizzo della rilocazione dinamica del processo in una nuova area di memoria. Come detto precedentemente, solo i computer dotati di registro di rilocazione possono utilizzare questa tecnica.

## **11.7 Allocazione non contigua della memoria**

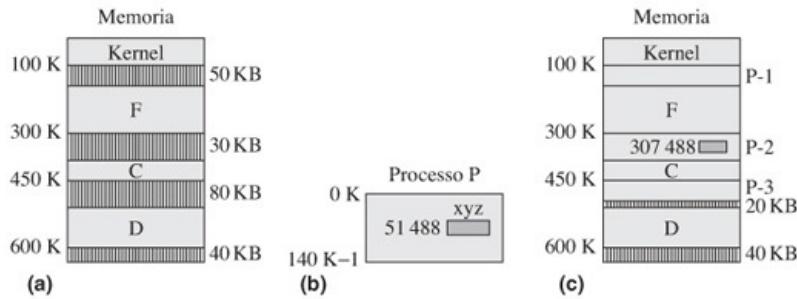
Le moderne architetture di computer utilizzano il modello di *allocazione non contigua della memoria*, in cui un processo funziona correttamente anche quando le parti del suo spazio di indirizzamento sono distribuite in più aree di memoria. Questo modello di allocazione della memoria consente al kernel di riutilizzare le aree di memoria libere più piccole rispetto alla dimensione di un processo, per cui si può ridurre la frammentazione esterna. Come vedremo successivamente in questo paragrafo, l'allocazione di memoria non contigua mediante paginazione può eliminare completamente la frammentazione esterna.

L'Esempio 11.9 illustra l'allocazione non contigua della memoria. Usiamo il termine *componente* per quella parte dello spazio di indirizzamento del processo caricata in un'unica area di memoria.

### **Esempio 11.9 Allocazione non contigua della memoria**

In [Figura 11.18\(a\)](#), sono presenti quattro aree di memoria libera a partire dagli indirizzi 100K, 300K, 450K e 600K, dove K = 1024, con dimensioni, rispettivamente, di 50 KB, 30 KB, 80 KB e 40 KB. Deve essere avviato il processo P, che ha dimensione pari a 140 KB [vedi [Figura 11.18\(b\)](#)]. Se il processo P si compone di tre componenti chiamate P-1, P-2 e P-3, con dimensioni, rispettivamente, di 50 KB, 30 KB e 60 KB, queste componenti possono essere caricate in tre delle aree di memoria libere come segue [[Figura 11.18\(c\)](#)]:

| Componente del processo | Dimensione | Indirizzo di avvio in memoria |
|-------------------------|------------|-------------------------------|
| P-1                     | 50 KB      | 100K                          |
| P-2                     | 30 KB      | 300K                          |
| P-3                     | 60 KB      | 450K                          |



**Figura 11.18** Allocazione non contigua della memoria al processo P.

### 11.7.1 Indirizzo logico, indirizzo fisico e traduzione di indirizzo

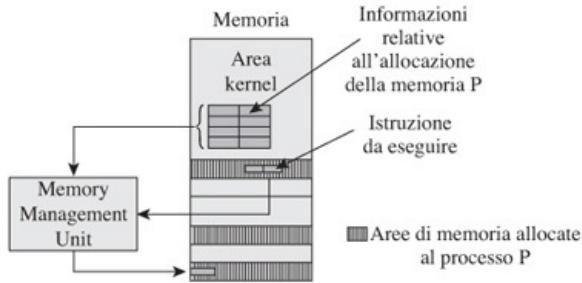
Nel Paragrafo 1.1, abbiamo visto che la visione astratta di un sistema è chiamata *vista logica* e l'organizzazione e la relazione tra le sue componenti è chiamata *organizzazione logica*. Inoltre la reale visione del sistema è chiamata *vista fisica* e l'organizzazione mostrata in essa è chiamata *organizzazione fisica*. Di conseguenza, le viste del processo P mostrato nelle Figure 11.18(b) e 11.18(c) costituiscono, rispettivamente, le viste logiche e fisiche del processo P dell'Esempio 11.9.

Un *indirizzo logico* è l'indirizzo di un'istruzione o di un byte di dati come utilizzato in un processo; può essere ottenuto utilizzando i registri indice, base o segmento. Gli indirizzi logici in un processo costituiscono lo *spazio di indirizzamento logico* del processo. Un *indirizzo fisico* è l'indirizzo in memoria in cui sono presenti un'istruzione o byte di dati. L'insieme degli indirizzi fisici nel sistema costituisce lo *spazio di indirizzamento fisico* del sistema.

#### Esempio 11.10 Spazi di indirizzamento logico e fisico

Nell'Esempio 11.9, lo spazio di indirizzamento logico di P si estende da 0 a 140K-1, mentre lo spazio di indirizzamento fisico si estende da 0 a 640K-1. L'area dati xyz nel programma del processo P ha l'indirizzo 51 488 [Figura 11.18(b)]. Questo è l'indirizzo logico di xyz. La componente P-1 del processo in Figura 11.18 ha una dimensione di 50 KB, ovvero 51200 byte, per cui xyz si trova nella componente P-2 ed ha il byte numero 288. Poiché P-2 è caricato nell'area di memoria con indirizzo di avvio 300 KB, ovvero 307200 byte, l'indirizzo fisico di xyz è 307488 [Figura 11.18(c)].

Lo schema di Figura 11.19 mostra come la CPU ottiene l'indirizzo fisico corrispondente all'indirizzo logico. Il kernel memorizza le informazioni relative alle aree di memoria allocate al processo P in una tabella e le rende disponibili alla *memory management unit* (MMU). Nell'Esempio 11.9, queste informazioni sono le dimensioni e gli indirizzi di avvio in memoria di P-1, P-2 e P-3. La CPU invia gli indirizzi logici di ogni dato o istruzione utilizzati nel processo alla MMU e la MMU utilizza le informazioni relative all'allocazione della memoria, memorizzate nella tabella, per calcolare i corrispondenti indirizzi fisici. Questo indirizzo è chiamato *indirizzo di memoria effettivo* del dato o dell'istruzione. La procedura per calcolare l'indirizzo di memoria effettivo a partire da un indirizzo logico è chiamata *traduzione dell'indirizzo*.



**Figura 11.19** Uno schema della traduzione dell'indirizzo nell'allocazione non contigua della memoria.

Un indirizzo logico usato in un'istruzione si compone di due parti: l'id della componente del processo contenente l'indirizzo e l'id del byte all'interno della componente. Rappresentiamo ogni indirizzo logico mediante una coppia nella forma:

$$(comp_i, byte_i)$$

La memory management unit calcola il suo indirizzo di memoria effettivo mediante la formula:

$$\begin{aligned} & \text{indirizzo di memoria effettivo di } (comp_i, byte_i) \\ &= \text{indirizzo di avvio dell'area di memoria allocata a } comp_i \\ & \quad + \text{numero del byte di } byte_i \text{ in } comp_i \end{aligned} \quad (11.1)$$

Negli Esempi 11.9 e 11.10, le istruzioni di P si riferiscono all'area dati xyz attraverso l'indirizzo logico (P-2, 288). La MMU calcola il suo indirizzo di memoria effettivo come  $307\ 200 + 288 = 307\ 488$ .

## 11.7.2 Approcci all'allocazione non contigua della memoria

Esistono due approcci fondamentali per implementare l'allocazione non contigua della memoria:

- paginazione;
- segmentazione.

Nella *paginazione*, ogni processo viene diviso in parti di dimensione fissata chiamate *pagine*. La dimensione della pagina viene definita dall'architettura del sistema e la divisione delle pagine è implicita in essa. La memoria può memorizzare un numero intero di pagine. Viene partizionata in aree di memoria che hanno la stessa dimensione di una pagina e ognuna di queste aree di memoria, detti frame, è considerata separatamente per l'allocazione a una pagina. In questo modo, ogni area di memoria libera è esattamente della stessa dimensione della pagina, per cui non si crea frammentazione esterna nel sistema. La frammentazione interna può crearsi poiché all'ultima pagina di un processo viene allocata un'area di memoria della dimensione di una pagina anche se è più piccola della dimensione di una pagina.

| Funzione                     | Allocazione contigua  | Allocazione non contigua   |
|------------------------------|---|--|
| Allocazione della memoria    | Il kernel alloca una singola area di memoria a un processo.   | Il kernel alloca diverse aree di memoria a un processo – ogni area contiene una componente del processo.   |
| Traduzione dell'indirizzo    | La traduzione dell'indirizzo non è necessaria.  | La traduzione dell'indirizzo è eseguita dalla MMU durante l'esecuzione del programma.  |
| Frammentazione della memoria | Si verifica frammentazione esterna quando viene usata l'allocazione first-fit, best-fit o next-fit. Si verifica frammentazione interna se l'allocazione della memoria viene eseguita in blocchi di poche dimensioni standard. | Nella paginazione non si verifica la frammentazione esterna ma può verificarsi la frammentazione interna. Nella segmentazione si verifica la frammentazione esterna, ma non si verifica la frammentazione interna. |
| Swapping                     | Se il computer non ha un registro di rilocazione, un processo swapato deve essere rimesso nell'area originariamente allocata a esso.  | Le componenti di un processo swapato possono essere caricate in una qualunque parte della memoria.   |

**Tabella 11.4** Confronto tra l'allocazione di memoria contigua e non contigua.

Nella *segmentazione* un programmatore identifica le componenti di un processo chiamate *segmenti*. Un segmento è un'entità logica di un programma, per esempio, un insieme di funzioni, di strutture dati o oggetti. La segmentazione facilita la condivisione del codice, dei dati e dei moduli dei programmi tra i processi. Tuttavia, i segmenti hanno dimensioni differenti, per cui il kernel deve usare tecniche di riutilizzo della memoria come l'allocazione first-fit o best-fit. Di conseguenza, si può verificare frammentazione esterna. Un approccio ibrido chiamato *segmentazione con paginazione* combina le caratteristiche della segmentazione e della paginazione. Facilita la condivisione di codice, dati e moduli di programma tra i processi senza incorrere nella frammentazione esterna; tuttavia la frammentazione interna si verifica come nella paginazione.

Affronteremo le caratteristiche di questi tre approcci nei paragrafi successivi.

La **Tabella 11.4** riassume i vantaggi dell'allocazione non contigua della memoria rispetto all'allocazione contigua della memoria. Lo swapping è più efficiente nell'allocazione non contigua della memoria poiché la traduzione degli indirizzi consente al kernel di caricare le componenti di un processo swapato in una qualsiasi parte della memoria.

### 11.7.3 Protezione della memoria

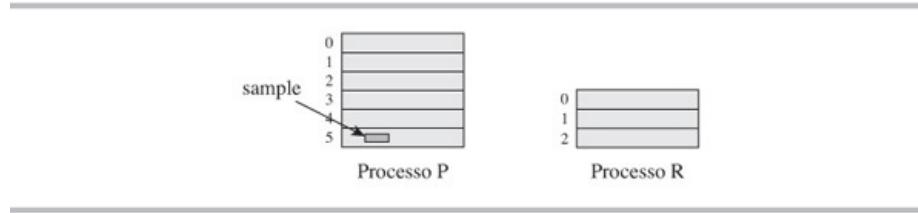
Ogni area di memoria allocata a un programma deve essere protetta contro le interferenze da parte di altri programmi. La MMU implementa questa funzione mediante il controllo dei limiti. Nel tradurre un indirizzo logico ( $comp_i, byte_i$ ), la MMU controlla se  $comp_i$  è un indirizzo del programma e se  $byte_i$  è presente in  $comp_i$ . Un interrupt di violazione della protezione viene generato se uno di questi controlli fallisce. Il controllo dei limiti può essere semplificato nella paginazione; non è necessario verificare se  $byte_i$  è in  $comp_i$  poiché, come vedremo nel prossimo paragrafo, un indirizzo logico non ha un numero di bit sufficiente a rappresentare un valore di  $byte_i$  che eccede la dimensione della pagina.

## 11.8 Paginazione

Nella vista logica, lo spazio di indirizzamento di un processo consiste in un'organizzazione lineare delle pagine. Ogni pagina è composta da  $s$  byte, dove  $s$  è una potenza del 2. Il valore di  $s$  è specificato nell'architettura del computer. I processi usano indirizzi logici numerici. La MMU scomponete un indirizzo logico nella coppia  $(p_i, b_i)$ , dove  $p_i$  è il numero di pagina e  $b_i$  è il numero del byte all'interno della pagina  $p_i$ . Le pagine in un programma e i byte in una pagina sono numerati a partire da 0; per cui, in un indirizzo logico  $(p_i, b_i)$ ,  $p_i \geq 0$  e  $0 \leq b_i < s$ . Nella vista fisica, le pagine di un processo si trovano in aree di memoria non adiacenti.

Si considerino due processi P ed R in un sistema che utilizza una pagina di dimensione 1 KB. I byte in una pagina sono numerati da 0 a 1023. Il processo P ha indirizzo di avvio

0 e dimensione 5500 byte. Dunque ha 6 pagine numerate da 0 a 5. L'ultima pagina contiene solo 380 byte. Se un dato sample ha indirizzo 5248, ovvero  $5 \times 1024 + 128$ , la MMU vede il suo indirizzo come la coppia (5, 128). Il processo P ha dimensione 2500 byte. Dunque ha 3 pagine, numerate da 0 a 2. La [Figura 11.20](#) mostra la vista logica dei processi P ed R.



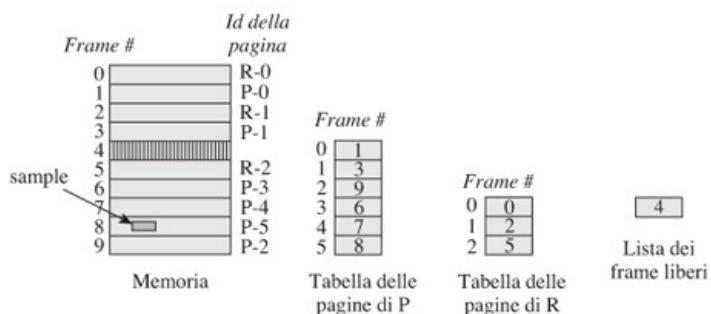
**Figura 11.20** Vista logica dei processi nella paginazione.

L'hardware partiziona la memoria in aree chiamate *frame*; i frame in memoria sono numerati a partire da 0. Ogni frame ha la stessa dimensione della pagina. A ogni istante, alcuni frame sono allocati alle pagine dei processi, mentre altri sono liberi. Il kernel mantiene un lista chiamata *lista dei frame liberi* per tenere traccia dei numeri dei frame liberi. Nel caricare un processo per l'esecuzione, il kernel consulta la lista dei frame liberi e alloca un frame libero a ogni pagina del processo.

Per facilitare la traduzione dell'indirizzo, il kernel costruisce una *tabella delle pagine* (PT) per ogni processo. La tabella delle pagine ha un elemento per ogni pagina del processo, che indica il frame allocato alla pagina. Durante la traduzione di un indirizzo logico  $(p_i, b_i)$ , la MMU utilizza il numero di pagina  $p_i$  per indicizzare la tabella delle pagine del processo, ottiene il numero del frame della pagina allocata a  $p_i$  e calcola l'indirizzo di memoria effettivo secondo l'[Equazione 11.1](#).

La [Figura 11.21](#) mostra la vista fisica dell'esecuzione dei processi P e R. Ogni frame ha dimensione 1 KB. Il computer ha una memoria di 10 KB, per cui i frame sono numerati da 0 a 9. Sei frame sono occupati dal processo P e tre frame sono occupati dal processo R. Le pagine contenute nei frame sono mostrate come P-0, ..., P-5 e R-0, ..., R-2. Il frame 4 è libero. Dunque la lista dei frame liberi contiene un solo elemento. La tabella delle pagine di P indica il frame allocato a ogni pagina di P. Come detto precedentemente, la variabile *sample* del processo P ha indirizzo logico (5, 128). Quando il processo P usa questo indirizzo logico durante la sua esecuzione, questo sarà tradotto nell'effettivo indirizzo di memoria utilizzando l'[Equazione 11.1](#) come segue:

$$\begin{aligned} & \text{indirizzo di memoria effettivo di (5, 128)} \\ &= \text{indirizzo di avvio del frame } \#8 + 128 \\ &= 8 \times 1024 + 128 \\ &= 8320 \end{aligned}$$



**Figura 11.21** Organizzazione fisica della paginazione.

Usiamo la seguente notazione per descrivere come viene eseguita la traduzione dell'indirizzo:

$s$  dimensione di una pagina;

lunghezza di un indirizzo logico (ovvero, numero di bit in esso contenuti);

$l_l$

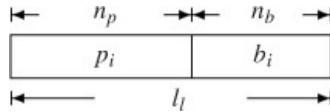
$l_p$  lunghezza dell'indirizzo fisico;

$n_b$  numero di bit utilizzati per rappresentare il byte in un indirizzo logico;

$n_p$  numero di bit utilizzati per rappresentare il numero di pagina in un indirizzo logico;

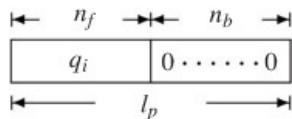
$n_f$  numero di bit utilizzati per rappresentare il numero di frame in un indirizzo fisico.

La dimensione di una pagina,  $s$ , è una potenza del 2.  $n_b$  viene scelto in modo tale che  $s = 2^{n_b}$ . Dunque gli  $n_b$  bit meno significativi in un indirizzo logico danno  $b_i$ , il byte all'interno della pagina. I restanti bit in un indirizzo logico a partire da  $p_i$ , rappresentano il numero di pagina. La MMU ottiene il valore di  $p_i$  e  $b_i$  semplicemente raggruppando i bit di un indirizzo logico come segue:

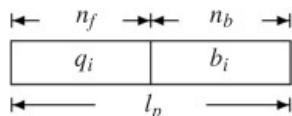


dove  $n_p = l_l - n_b$ . L'uso di una potenza del 2 come dimensione della pagina semplifica la costruzione dell'indirizzo di memoria effettivo. Sia la pagina  $p_i$  allocata nel frame  $q_i$ .

Poiché le pagine e i frame hanno le stesse dimensioni, sono necessari  $n_b$  bit per indirizzare i byte in un frame. L'indirizzo fisico del byte 0 di un frame  $q_i$  è dunque:



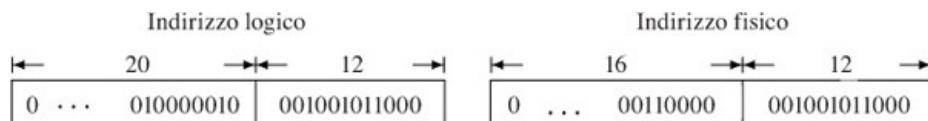
dove  $n_f$  è il numero di bit utilizzato per rappresentare il numero del frame. Dunque  $n_f = l_p - n_b$ . L'indirizzo fisico del byte  $b_i$  nel frame  $q_i$  è ora dato da:



La MMU ottiene questo indirizzo semplicemente concatenando  $q_i$  e  $b_i$  per ottenere un numero di bit  $l_p$ . Il prossimo esempio illustra la traduzione dell'indirizzo in un sistema che utilizza la paginazione.

### Esempio 11.11 Traduzione dell'indirizzo nella paginazione

Un ipotetico computer utilizza indirizzi logici a 32-bit e una dimensione di pagina di 4 KB. Si supponga che 12 bit siano sufficienti per indirizzare i byte in una pagina. In questo modo, i venti bit più significativi in un indirizzo logico rappresentano  $p_i$  e i 12 bit meno significativi rappresentano  $b_i$ . Per una memoria di 256 MB,  $l_p = 28$ . In questo modo, i 16 bit più significativi in un indirizzo fisico rappresentano  $q_i$ . Se la pagina 130 si trova nel frame 48,  $p_i = 130$  e  $q_i = 48$ . Se  $b_i = 600$ , gli indirizzi fisico e logico appaiono come segue:

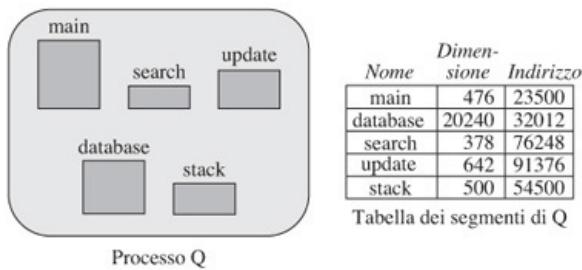


Durante la traduzione dell'indirizzo, la MMU ottiene  $p_i$  e  $b_i$  semplicemente raggruppando i bit dell'indirizzo logico come mostrato sopra. Si accede al 130-esimo elemento della tabella delle pagine per ottenere  $q_i$ , ovvero 48. Questo numero viene concatenato a  $b_i$  per formare l'indirizzo fisico.

## 11.9 Segmentazione

Un *segmento* è un'entità logica in un programma, per esempio una funzione, una struttura dati o un oggetto. Dunque ha senso gestirlo come una unità: caricarlo in memoria per l'esecuzione o condividerlo con altri programmi. Nella vista logica, un processo si compone di un insieme di segmenti. Nella vista fisica, i segmenti di un processo si trovano in aree di memoria non adiacenti.

Un processo Q si compone di cinque entità logiche con nomi simbolici `main`, `database`, `search`, `update` e `stack`. Nel codificare il programma, il programmatore dichiara questi cinque segmenti in Q. Tale informazione viene utilizzata dal compilatore o dall'assembler per generare gli indirizzi logici durante la traduzione del programma. Ogni indirizzo logico utilizzato in Q ha la forma  $(s_i, b_i)$  dove  $s_i$  e  $b_i$  sono rispettivamente gli id di un segmento e un byte all'interno di un segmento. Per esempio, l'istruzione corrispondente alla chiamata `get_sample`, dove `get_sample` è una procedura nel segmento `update`, può usare l'indirizzo dell'operando  $(update, get\_sample)$ . In alternativa, può usare una rappresentazione numerica in cui  $s_i$  e  $b_i$  sono, rispettivamente, il numero del segmento e il numero del byte all'interno del segmento. Per semplicità, in questo capitolo assumiamo questa rappresentazione.



**Figura 11.22** Un processo Q nella segmentazione.

La [Figura 11.22](#) mostra come il kernel gestisce il processo Q. La parte sinistra della [Figura 11.22](#) mostra la visione logica del processo Q. Per facilitare la traduzione dell'indirizzo, il kernel costruisce una tabella dei segmenti per Q. Ogni elemento in questa tabella mostra la dimensione di un segmento e l'indirizzo dell'area di memoria a esso allocata. Il campo dimensione viene utilizzato per eseguire il controllo dei limiti per la protezione della memoria. La MMU utilizza la tabella dei segmenti per eseguire la traduzione degli indirizzi ([Figura 11.19](#)). I segmenti non hanno dimensioni fissate, per cui la traduzione degli indirizzi non può essere eseguita mediante la concatenazione dei bit come nella paginazione. Il calcolo dell'indirizzo di memoria effettivo per l'indirizzo logico  $(s_i, b_i)$  consiste nel sommare  $b_i$  all'indirizzo di avvio del segmento  $s_i$  secondo l'Equazione 11.1. In [Figura 11.22](#), se `get_sample` ha numero di byte 232 nel segmento `update`, la traduzione dell'indirizzo di  $(update, get\_sample)$  fornirà l'indirizzo  $91\ 376 + 232 = 91\ 608$ .

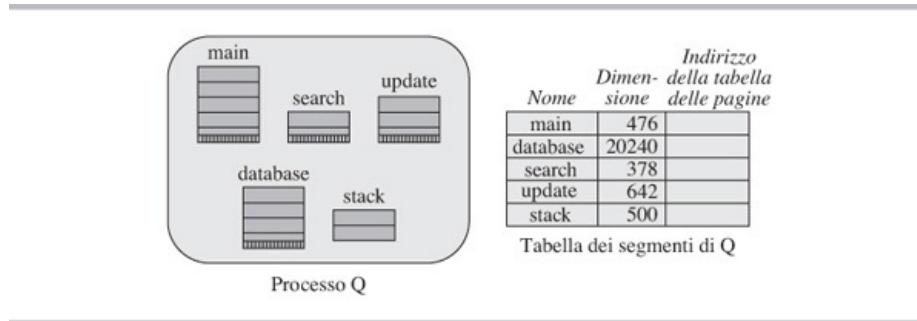
L'allocazione della memoria per ogni segmento viene eseguita come nel modello di allocazione della memoria contigua. Il kernel mantiene una lista delle aree di memoria libera. Nel caricare un processo, cerca in questa lista per eseguire l'allocazione first-fit o best-fit per ogni segmento del processo. Quando un processo termina, le aree di memoria allocate ai suoi segmenti vengono aggiunte alla free list. La frammentazione esterna può verificarsi poiché i segmenti hanno dimensioni differenti.

## 11.10 Segmentazione con paginazione

Secondo tale approccio, ogni segmento di un programma viene paginato separatamente. Di conseguenza, viene allocato a ogni segmento un numero intero di pagine. Questo approccio semplifica l'allocazione della memoria e la rende più veloce e inoltre evita la frammentazione esterna. Per ogni segmento viene creata una tabella delle pagine e l'indirizzo della tabella delle pagine viene memorizzato nell'elemento relativo al segmento nella tabella dei segmenti. La traduzione di un indirizzo logico  $(s_i, b_i)$  viene eseguita in due fasi. Nella prima fase, si cerca l'elemento di  $s_i$  nella tabella dei segmenti

e viene estratto l'indirizzo della sua tabella delle pagine. Il numero del byte  $b_i$  è ora diviso in una coppia  $(ps_i, bp_i)$ , dove  $ps_i$  è il numero di pagina nel segmento  $s_i$ , e  $bp_i$  è il numero del byte nella pagina  $p_i$ . Il calcolo dell'indirizzo effettivo è completato come nella paginazione, ovvero si ottiene il numero del frame di  $ps_i$  e  $bp_i$  viene concatenato a esso per ottenere l'indirizzo effettivo.

La Figura 11.23 mostra il processo Q di Figura 11.22 in un sistema che utilizza la segmentazione con paginazione. Ogni segmento viene paginato separatamente, per cui si verifica frammentazione interna nell'ultima pagina di ogni segmento. Ogni elemento della tabella dei segmenti contiene ora l'indirizzo della tabella delle pagine relativa al segmento. Si utilizza il campo dimensione in un elemento del segmento per facilitare la verifica dei limiti per la protezione della memoria.



**Figura 11.23** Un processo Q nella segmentazione con paginazione.

## 11.11 Allocazione della memoria del kernel

Il kernel crea e distrugge strutture dati a un tasso elevato durante il suo funzionamento. Questi sono per la maggior parte blocchi di controllo (*control block*, CB) che controllano l'allocazione e l'utilizzo delle risorse del sistema. Alcuni control block famosi sono il process control block (PCB) creato per ogni processo e l'event control block (ECB) creato per anticipare l'occorrenza di un evento. Nei Capitoli 13 e 14, illustreremo altri due control block utilizzati di frequente: l'I/O control block (IOCB) creato per un'operazione di I/O e il file control block (FCB) creato per ogni file aperto. Le dimensioni dei control block sono rese note nella fase di progettazione di un SO. Questa informazione aiuta a rendere l'allocazione della memoria del kernel facile ed efficiente; la memoria liberata quando un control block viene distrutto può essere riutilizzata quando si crea un control block simile. Per realizzare questo beneficio, è possibile mantenere una free list separata per ogni tipo di control block.

I kernel dei moderni sistemi operativi usano l'allocazione non contigua della memoria con paginazione per soddisfare le proprie richieste di memoria con particolare attenzione all'utilizzo efficiente di ogni pagina. Tre dei più importanti allocator sono:

- allocatore McKusick-Karels;
- allocatore lazy buddy;
- allocatore slab.

Gli allocator McKusick-Karels e lazy buddy allocano aree di memoria con dimensione potenza del 2 *nella* pagina. Poiché l'indirizzo di avvio di ogni pagina in memoria è una potenza maggiore del 2, l'indirizzo di avvio di ogni area di memoria allocata di dimensione  $2^n$  è un multiplo di  $2^n$ . Questa caratteristica, chiamata allineamento al confine su una potenza del 2, comporta il seguente problema di prestazioni della cache: si accede più frequentemente ad alcune parti rispetto ad altre. A causa dell'allineamento al confine su una potenza del 2, le parti di un oggetto cui si accede più di frequente possono essere mappate nella stessa area della cache mediante la tecnica auto-associativa di ricerca nella cache. Dunque in alcune parti della cache si verificano molte contese che portano il codice del kernel ad avere scarse prestazioni. L'allocatore slab utilizza una tecnica interessante per evitare questo problema di prestazioni.

Seguono le descrizioni di questi tre allocator. Per coerenza, usiamo la stessa terminologia utilizzata nei paragrafi precedenti; essa differisce dalla terminologia utilizzata della letteratura esistente. La bibliografia alla fine del capitolo indica quali

moderni sistemi operativi usano questi allocatori.

### **Allocatore McKusick-Karels**

Si tratta di un allocator potenza del 2 modificato e viene utilizzato in Unix 4.4. BSD. L'allocator a ogni istante ha a disposizione un numero intero di pagine e richiede al sistema di paginazione ulteriori pagine quando non ha pagine da allocare. Il principio di funzionamento di base dell'allocator è di dividere ogni pagina in blocchi di eguale dimensione e registrare due elementi di informazione - la dimensione del blocco e il puntatore alla free list - nell'ambito dell'indirizzo logico della pagina. In questo modo, l'indirizzo della pagina in cui si trova il blocco sarà sufficiente a trovare la dimensione del blocco e la free list cui il blocco dovrebbe essere aggiunto quando è liberato. Dunque, non è necessario avere un header contenente questa informazione in ogni blocco allocato come nei normali allocatori potenza del 2. Con l'eliminazione dell'header, può essere usata per l'allocazione l'intera memoria in un blocco. Di conseguenza, l'allocator McKusick-Karels è superiore all'allocator potenza del 2 quando viene eseguita una richiesta di memoria per un'area la cui dimensione è una esatta potenza del 2. Per soddisfare la richiesta può essere allocato un blocco di identiche dimensioni, mentre il normale allocator potenza del 2 avrebbe dovuto allocare un blocco con dimensione pari alla successiva potenza del 2.

L'allocator cerca una pagina libera tra quelle in suo possesso quando non trova un blocco della dimensione che sta cercando. Successivamente divide questa pagina in blocchi della dimensione desiderata. Alloca uno di questi blocchi per soddisfare la richiesta corrente e inserisce i blocchi rimanenti nella free list appropriata. Se l'allocator non detiene pagine libere, richiede al sistema di paginazione una nuova pagina da allocare. Per assicurare che non venga utilizzato un numero di pagine maggiore di quelle necessarie, l'allocator segna ogni pagina in suo possesso come libera quando tutti i blocchi al suo interno diventano liberi. Tuttavia, gli manca una funzionalità per restituire le pagine libere al sistema di paginazione. In questo modo, il numero totale di pagine allocate all'allocator è ogni volta il più grande numero di pagine che abbia mai posseduto, riducendo notevolmente il fattore di utilizzo della memoria.

### **Allocatore lazy buddy**

Il buddy system nella sua forma base può eseguire una o più divisioni per ogni allocazione e una o più azioni di fusione a ogni rilascio. Alcune di queste azioni sono inutili poiché un blocco ricomposto può dover essere diviso nuovamente. Il principio base di progettazione dell'allocator lazy buddy è di ritardare le azioni di fusione se verosimilmente deve essere creata una struttura dati che richiede la stessa quantità di memoria rilasciata. In determinate condizioni, questo principio evita l'overhead sia della fusione che della divisione.

L'allocator lazy buddy utilizzato in Unix 5.4 funziona come segue: i blocchi con la stessa dimensione sono considerati parte di una *classe* di blocchi. Le decisioni riguardanti la fusione per una classe sono fatte sulla base dei tassi ai quali le strutture dati di una classe vengono create e distrutte. Di conseguenza, l'allocator caratterizza il comportamento del SO rispetto a una classe di blocchi in tre stati chiamati *lazy*, *reclaiming* e *accelerated*. Per semplicità ci riferiamo a questi come *stati* di una classe di blocchi.

Nello stato *lazy*, le allocazioni e i rilasci dei blocchi di una classe si verificano a tassi corrispondenti. Di conseguenza, esiste un ciclo stabile e potenzialmente inutile di divisioni e fusioni. Come rimedio, le fusioni e le divisioni eccessive possono essere entrambe evitate ritardando la fusione. Nello stato *reclaiming*, i rilasci si verificano a un tasso più elevato rispetto alle allocazioni, per cui è una buona idea fondere a ogni rilascio. Nello stato *accelerated*, i rilasci si verificano molto più velocemente delle allocazioni, per cui è auspicabile fondere a un tasso ancora più elevato; l'allocator dovrebbe tentare di fondere un blocco rilasciato e, inoltre, dovrebbe tentare di fondere altri blocchi che erano stati precedentemente rilasciati ma non fusi.

L'allocator lazy buddy mantiene la free list come una lista a doppio puntatore. In questo modo è possibile accedere sia alla testa che alla coda della lista in maniera egualmente facile. Si adotta una bitmap per indicare lo stato di allocazione dei blocchi. Nello stato *lazy*, un blocco rilasciato viene semplicemente aggiunto alla testa della free list. Non vengono fatti tentativi di fonderlo con il suo buddy. Inoltre non viene marcato come libero nella bitmap. In questo modo il blocco non sarà fuso anche se il suo buddy verrà rilasciato in futuro. Tale blocco viene detto *localmente libero*. Stando in testa alla

lista, questo blocco sarà allocato prima di ogni altro blocco nella lista. La sua allocazione è efficiente e veloce poiché la bitmap non necessita di essere aggiornata; il blocco è ancora marcato come allocato.

Negli stati reclaiming e accelerated un blocco è sia aggiunto alla free list che marcato come libero nella bitmap. Tale blocco è detto *globalmente libero*. I blocchi globalmente liberi sono aggiunti alla fine della lista. Nello stato reclaiming l'allocatore cerca di fondere un nuovo blocco globalmente libero in modo transitivo al suo buddy. Un blocco può essere eventualmente aggiunto a qualche free list, o a una alla quale il blocco rilasciato apparteneva inizialmente, oppure a una contenente blocchi di dimensione maggiore. È da notare che il blocco aggiunto a una free list potrebbe essere un blocco localmente libero o un blocco globalmente libero in base allo stato di quella classe di blocchi. Nello stato accelerated l'allocatore cerca di fondere il blocco rilasciato, come nello stato reclaiming e inoltre cerca di fondere un altro blocco localmente libero - il blocco trovato in testa alla free list - al suo buddy.

Lo stato di una classe di blocchi è caratterizzato come segue. Siano A, L e G il numeri di blocchi di una classe, rispettivamente, allocati, localmente liberi e globalmente liberi. Il numero totale di blocchi di una classe è dato da  $N = A + L + G$ . Un parametro chiamato *slack* viene calcolato come segue:

$$\text{slack} = N - 2 \times L - G$$

Una classe si definisce nello stato lazy, reclaiming o accelerated se il valore di *slack* risulta, rispettivamente,  $\geq 2$ , 1 o 0. (L'allocatore garantisce che slack non sia mai  $< 0$ .) L'overhead di fusione è differente in questi tre stati. Non c'è overhead nello stato lazy. Dunque il rilascio e l'allocazione dei blocchi risulta veloce. Nello stato reclaiming l'overhead è confrontabile con quello del buddy system, mentre nello stato accelerated l'overhead è maggiore rispetto al buddy system. È stato mostrato che i ritardi medi con l'allocatore lazy buddy sono dal 10 a 32 per cento più bassi rispetto ai ritardi medi nel caso di un allocatore buddy.

L'implementazione dell'allocatore lazy buddy in Unix 5.4 usa due tipi di blocchi. I blocchi piccoli variano in dimensione tra 8 e 256 byte. I blocchi grandi variano in dimensione tra 512 e 16 KB. L'allocatore ottiene memoria dal sistema di paginazione in aree di 4 KB. In ogni area, crea un pool di blocchi e una bitmap per tenere traccia dello stato di allocazione dei blocchi. Quando tutti i blocchi nel pool sono liberi, restituisce l'area al sistema di paginazione. Tale comportamento risolve il problema dei blocchi non restituiti che si è riscontrato nell'allocatore McKusick-Karels.

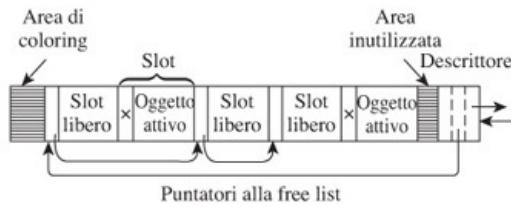
### **Allocatore slab**

L'allocatore slab era inizialmente usato nel sistema operativo Solaris 2.4; è stato successivamente utilizzato in Linux dalla versione 2.2. Uno *slab* consiste di molti *slot*, dove ogni slot può mantenere un oggetto attivo quale una struttura dati del kernel, o può essere vuoto. L'allocatore ottiene aree di memoria di dimensione standard dal sistema di paginazione e organizza uno *slab* in ogni area di memoria. Ottiene un'area di memoria aggiuntiva dal sistema di paginazione e costruisce uno slab al suo interno quando non ha più memoria da allocare e restituisce un'area di memoria al sistema di paginazione quando tutti gli slot dello slab sono inutilizzati.

Tutti gli oggetti del kernel della stessa classe formano un pool. Per piccoli oggetti, un pool consiste di molti slab e ogni slab contiene molti slot. (Gli oggetti grandi non sono discussi qui.) Gli slab di un pool sono inseriti in una lista a doppio puntatore per facilitare l'aggiunta e la rimozione degli slab. Uno slab può essere pieno, parzialmente vuoto o vuoto, in base al numero di oggetti attivi in esso contenuti. Per facilitare le ricerche di uno slab vuoto, la lista a doppio puntatore contenente gli slab di un pool è ordinata secondo lo stato degli slab - tutti gli slab pieni si trovano in testa alla lista, gli slab parzialmente vuoti si trovano al centro e gli slab vuoti si trovano alla fine della lista. Ogni slab contiene una free list da cui possono essere allocati gli slot liberi. Questa organizzazione rende l'allocazione molto efficiente.

La [Figura 11.24](#) mostra il formato di uno stab. Quando l'allocatore ottiene un'area di memoria dal sistema di paginazione, formatta l'area di memoria in uno slab creando un numero finito di slot, una free list contenente tutti gli slot e un campo descrittore alla fine dello slab che contiene sia il conteggio degli oggetti attivi al suo interno sia l'header della free list. Ogni slot nello slab viene inizializzato; in particolare, si inizializzano i vari campi con le informazioni specifiche di ogni oggetto, quali stringhe di lunghezza fissa di

valori costanti. Una volta allocato, lo slot può essere usato direttamente come un oggetto. Al momento della deallocazione, l'oggetto viene riportato allo stato dell'allocazione e lo slot viene inserito nella free list. Poiché alcuni campi degli oggetti non cambiano mai, o cambiano in maniera tale che i loro valori al momento della deallocazione sono uguali a quelli al momento dell'allocazione, questo approccio elimina l'overhead ripetitivo dell'inizializzazione dell'oggetto presente in molti altri allocator. Tuttavia, l'uso di oggetti inizializzati ha influenza nel fattore di utilizzo della memoria. Se uno slot libero fosse semplicemente memoria libera, una parte di questa memoria potrebbe essere usata come puntatore alla free list; ma uno slot è un oggetto inizializzato, per cui il campo puntatore deve essere posto al di fuori dell'area dell'oggetto anche quando lo slot è libero (Figura 11.24).



**Figura 11.24** Formato di uno slab.

L'allocatore slab fornisce un miglior comportamento della cache evitando il problema di prestazioni della cache mostrato dagli allocator potenza del 2 e le loro varianti descritte all'inizio di questo paragrafo. Ogni slab contiene al suo inizio un'area riservata chiamata *area di coloring* (Figura 11.24). L'allocatore usa aree di coloring di differente dimensione negli slab di un pool. Di conseguenza, gli oggetti nei differenti slab di un pool hanno differenti allineamenti rispetto al più prossimo multiplo potenza del 2, per cui vengono mappati in aree differenti di una cache set-associativa. A tal proposito si ricorda che in una cache set-associativa, detta anche semi-associativa, ogni blocco della memoria inferiore può essere mappato in un insieme di blocchi della memoria superiore. Questa caratteristica evita una contesa eccessiva per alcune aree della cache, migliorando le prestazioni della cache.

L'allocatore slab fornisce inoltre un migliore fattore di utilizzo della memoria poiché alloca solo la quantità di memoria richiesta per ogni oggetto. In questo modo, diversamente dagli allocator McKusick-Karels e lazy buddy, non si verifica frammentazione interna per gli oggetti; si verifica solo frammentazione esterna nella forma di un'area inutilizzata in ogni slab. Bonwick (1994) ha riportato che la frammentazione rappresenta solo il 14 per cento nell'allocatore slab contro il 45 e 46 per cento, rispettivamente, degli allocator McKusick-Karels e lazy buddy. I tempi di allocazione media sono anche migliori rispetto ad altri allocator.

## 11.12 Utilizzo efficiente della RAM idle

Una workstation o un laptop sono solitamente dotati di una grande memoria necessaria per eseguire specifiche applicazioni. Tuttavia, la memoria rimane idle quando le applicazioni non sono attive. I progettisti dei sistemi operativi hanno riflettuto a lungo su come la memoria idle possa essere sfruttata a beneficio dell'utente. Una soluzione tipica è di eseguire programmi di utilità come software antivirus durante i periodi idle di un computer in modo che la loro esecuzione non occupi memoria e consumi tempo di CPU quando il computer viene utilizzato per scopi produttivi. Tuttavia, anche queste operazioni di utilità possono avere un impatto negativo sulle prestazioni poiché potrebbero spostare applicazioni importanti dalla memoria, e devono essere ricaricate in memoria prima di poter essere utilizzate.

Il sistema operativo Windows Vista ha una funzionalità chiamata *SuperFetch* che mantiene informazioni prioritarie nelle applicazioni e i documenti usati frequentemente e le utilizza per precaricare le applicazioni e i documenti ad alta priorità nelle parti idle della memoria. Inoltre assicura che solo le applicazioni idle a bassa priorità vengano rimosse dalla memoria per eseguire antivirus o altri programmi di utilità. Vista ha anche un'altra funzionalità chiamata *Readyboost* che utilizza una memoria flash in un drive USB per velocizzare le prestazioni del sistema copiando le applicazioni sul drive USB, da cui possono essere caricate in memoria più velocemente rispetto al disco. Quando usato

insieme col SuperFetch, il Readyboost rende il drive USB una cache tra la memoria e il disco, cosa che migliora le prestazioni del sistema mediante il caricamento veloce delle applicazioni.

## Riepilogo

In questo capitolo, abbiamo discusso le tecniche per la gestione efficiente della memoria, che coinvolge l'esecuzione dell'allocazione e deallocazione veloce della memoria ai processi e assicura l'uso efficiente della memoria in modo che molti processi possano essere presenti in memoria simultaneamente.

Quando un programma è codificato o compilato, non è possibile sapere quale area di memoria gli sarà allocata per l'esecuzione. Tuttavia, le istruzioni utilizzate devono utilizzare degli indirizzi di memoria per gli operandi. Tale problema viene risolto come segue: un compilatore assume che una specifica area di memoria sia disponibile per un programma e genera una tipologia di programma chiamata *modulo oggetto*. Il *linker*, che è un programma di sistema, utilizza la procedura chiamata *rilocazione*, che modifica gli indirizzi degli operandi nelle istruzioni di un programma in modo tale che il programma possa essere eseguito correttamente nell'area di memoria allocata. Il linker inoltre collega il programma con le funzioni di libreria per creare un programma pronto per l'esecuzione. I programmi *auto-rilocanti* possono eseguire la propria rilocazione. L'hardware del computer supporta la rilocazione dinamica del programma mediante un registro speciale nella CPU chiamato *registro di rilocazione*. Questo registro consente al kernel di modificare l'area di memoria allocata a un programma durante l'esecuzione.

L'allocazione della memoria può essere eseguita in due modi. *L'allocazione statica della memoria* viene eseguita prima dell'esecuzione di un programma; tuttavia richiede la conoscenza dell'esatta quantità di memoria richiesta, rischiando di sovrallocare e sprecare memoria. *L'allocazione dinamica della memoria* viene eseguita durante l'esecuzione di un programma, generando un overhead dovuto alla gestione della memoria durante l'esecuzione, ma fa un uso efficiente della memoria allocando solo la quantità di memoria richiesta. Il kernel usa un modello di allocazione della memoria per un processo che contiene una componente allocata staticamente per il codice e i dati del programma e componenti allocate dinamicamente per lo *stack* e l'*heap* nel quale un programma può allocare dinamicamente memoria mediante le istruzioni come *new* o *alloc*.

Quando un processo completa la propria esecuzione o rilascia la memoria allocata, il kernel riutilizza la memoria per soddisfare le richieste di altri processi. Utilizzando l'allocazione statica della memoria, parte della memoria allocata a un processo può restare inutilizzata, nel qual caso si parla di *frammentazione interna*. Utilizzando l'allocazione dinamica della memoria, a meno che le nuove richieste non coincidano con le dimensioni della memoria rilasciata, parte della memoria non viene allocata quando viene soddisfatta una nuova richiesta. Questa parte della memoria resta inutilizzata se è troppo piccola per soddisfare una richiesta, nel qual caso si parla di *frammentazione esterna*.

Due approcci possono essere usati per affrontare il problema della frammentazione: nel primo approccio, il kernel minimizza la frammentazione riutilizzando la memoria. Varie tecniche, chiamate allocazione *first-fit*, allocazione *best-fit*, ecc., vengono utilizzate per minimizzare la frammentazione esterna, mentre le tecniche, chiamate allocazione *buddy system* e allocazione *potenza del 2*, vengono utilizzate per eliminare la frammentazione esterna. Nell'altro approccio si usa l'*allocazione non contigua della memoria*, in cui un processo può essere eseguito anche quando gli vengono allocate molte aree di memoria di piccole dimensioni che, sommate, consentono tutta la richiesta di memoria. In questo modo la frammentazione esterna viene eliminata. La *paginazione* e la *segmentazione* sono due esempi di tale approccio. L'allocazione non contigua della memoria richiede l'uso di una *memory management unit* in hardware.

Il kernel crea e distrugge i control block, come il PCB, a un tasso molto elevato. Poiché le dimensioni dei control block sono note al kernel, è possibile minimizzare l'overhead per la gestione della memoria e il problema della frammentazione avendo a disposizione molti blocchi di memoria della dimensione richiesta e allocando uno di essi al momento della creazione di un nuovo control block. L'*allocatore lazy buddy* e l'*allocatore slab* sono alcune delle tecniche usate dal kernel.

## Domande

- 11.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
- Quando viene usato uno stack, il riuso di un'area di memoria rilasciata è automatico.
  - I dati PCD possono essere allocati sullo stack.
  - Il registro di rilocazione aiuta il kernel a eseguire la compattazione dei programmi per evitare la frammentazione esterna.
  - L'allocazione della memoria eseguita utilizzando un allocatore buddy system non soffre di frammentazione interna.
  - Quando un'area di memoria viene rilasciata in un sistema che adotta un allocatore buddy system, il numero di aree di memoria libere viene incrementato di 1, decrementato di 1 o non viene modificato.
  - La frammentazione esterna può verificarsi quando è utilizzato un allocatore buddy system o un allocatore potenza del 2.
  - Quando si adottano il linking dinamico e il caricamento, una routine non utilizzata in un'esecuzione del programma non è caricata in memoria.
  - In un sistema di paginazione, non è possibile swappare in memoria un processo in un insieme di aree di memoria non contigue differenti dall'insieme di aree di memoria non contigue da cui era stato swappato sul disco.
  - In un sistema di paginazione, un programmatore deve dividere le pagine in codice e dati di un programma.
  - Non ci sarebbe bisogno dei linker se tutti i programmi fossero implementati come programmi auto-rilocanti.
- 11.2. Seleziona la risposta corretta a ognuna delle seguenti domande.
- Un allocatore *worst-fit* divide sempre l'area di memoria libera più grande durante l'allocazione. Una free list contiene tre aree di memoria di dimensione 6 KB, 15 KB e 12 KB. Le successive quattro richieste di memoria sono per 10 KB, 2 KB, 5 KB e 14 KB di memoria. L'unica strategia di posizionamento in grado di allocare tutti e quattro i processi è
    - first-fit,
    - best-fit,
    - worst-fit,
    - next-fit.
  - Tre processi che richiedono 150 KB, 100 KB e 300 KB di memoria sono in esecuzione in un SO che adotta un sistema di paginazione con una dimensione di pagina di 2 KB. La massima frammentazione interna della memoria dovuta all'allocazione della memoria ai tre processi è
    - circa 2 KB
    - circa 6 KB
    - 275 KB
    - nessuna delle (i)-(iii)
  - Un programma rientrante è un programma che
    - chiama se stesso ricorsivamente;
    - può avere diverse copie in memoria che possono essere usate da differenti utenti;
    - può avere una singola copia in memoria che viene eseguita da molti utenti in maniera concorrente.

## Problemi

- 11.1. Un ipotetico linguaggio di programmazione consente di assegnare uno dei seguenti tre attributi a una variabile in un programma.
- Statico*: alle variabili con questo attributo viene allocata memoria al momento della compilazione.
  - Automatico*: quando l'esecuzione di un programma è avviata o una funzione/subroutine è invocata, le variabili con l'attributo *automatico* dichiarate nel programma, funzione o subroutine sono allocate in memoria. La memoria è

deallocata quando il programma termina o viene completata la chiamata della funzione/subroutine.

c. *Controllato*: una variabile *x* con l'attributo *controllato* viene allocata in memoria quando il programma esegue l'istruzione **new** *x*. La memoria è deallocata quando il programma esegue l'istruzione **release** *x*.

Discutere il metodo usato per allocare memoria alle variabili con ognuno di questi attributi. Commentare (i) l'efficienza nell'utilizzo della memoria e (ii) l'efficienza dell'esecuzione di questi metodi.

- 11.2. Un allocatore di memoria che utilizza la politica di allocazione best-fit ordina la sua free list in ordine crescente di dimensione delle aree libere. Questa organizzazione evita di dover esaminare l'intera free list per effettuare un'allocazione. Tuttavia, durante la gestione di una richiesta per  $n$  byte, l'allocatore non deve considerare gli elementi relativi ad aree di memoria con dimensione  $< n$  byte. Proporre un metodo per organizzare la free list che elimini l'overhead per esaminare e ignorare questi elementi.
- 11.3. Il kernel di un SO utilizza un allocatore di memoria separato per gestire le proprie richieste di memoria. Si rileva che questo allocatore di memoria riceve richieste per fornire e rilasciare aree di memoria esclusivamente per due dimensioni, ovvero 100 byte e 150 byte, a un tasso elevato. Commentare l'efficienza dell'utilizzo della memoria e la velocità di allocazione nel caso in cui l'allocatore di memoria sia
- un allocatore first-fit
  - un allocatore best-fit
  - un allocatore slab
- 11.4. Un allocatore di memoria utilizza la seguente politica per allocare una singola area di memoria contigua per le richieste di 1 KB e 2 KB: imposta separatamente un'area di memoria contigua di  $n$  KB per la gestione di tali richieste e divide quest'area di memoria in  $n$  aree di 1 KB ognuna. Per soddisfare una richiesta di 2 KB, unisce due aree libere contigue di 1 KB ognuna, se presenti e alloca l'area contigua risultante. Quando un'area di 2 KB viene rilasciata, considera l'area liberata come due aree libere di 1 KB ognuna. Mostrare che se l'allocatore ha 22 KB disponibili per l'allocazione, può non essere in grado di onorare richieste per un totale di 16 KB.
- 11.5. Ad un allocatore buddy system è allocata un'area di memoria di 64 KB. I blocchi di dimensione 2 KB, 11 KB, 120 byte e 20 KB sono allocati in quest'ordine.
- Mostrare lo stato dell'allocazione e le free list dell'allocatore. Quante divisioni sono effettuate?
  - Mostrare lo stato dell'allocazione e le free list dell'allocatore dopo che il blocco di 129 byte è stato liberato. Quante operazioni di fusione sono state effettuate?
- 11.6. Un allocatore potenza del 2 utilizza un blocco di dimensione minima di 16 byte e un blocco di dimensione massima di 32 KB. L'allocatore inizia il suo funzionamento con un blocco libero ognuno di dimensione 512 byte, 2 KB, 16 KB e 32 KB. Calcolare la frammentazione interna se l'allocatore elabora le stesse richieste Problema 11.5.
- 11.7. Quando un blocco di memoria viene rilasciato, un allocatore di memoria tenta di unirlo con uno o due dei suoi vicini. Siete d'accordo con la seguente affermazione? "Se le dimensioni dei blocchi vicini sono note, è sufficiente avere un tag a una sola estremità di ogni blocco. Tuttavia, se le dimensioni dei blocchi vicini non sono note, è essenziale avere tag a entrambe le estremità di ogni blocco."
- 11.8. Un buddy system organizza i tag dei blocchi in una bitmap, dato un array di tag monodimensionale. Commentare come sia possibile organizzare e utilizzare al meglio la bitmap. (*Suggerimento*: nota che i blocchi possono essere divisi e riuniti durante il funzionamento del buddy system.)
- 11.9. Se un buddy system binario inizia il suo funzionamento con un singolo blocco libero di dimensione  $2^z$  byte.
- Giustificare l'affermazione: "Quando un blocco viene rilasciato, il numero di blocchi liberi nel sistema può aumentare di 1, può rimanere uguale o può diminuire di un numero tra 1 e  $n$ , estremi inclusi, dove  $n < z$ ."
  - Determinare il valore di  $n$  se la dimensione minima del blocco nel buddy system è 16 byte.
- 11.10. Un buddy system di Fibonacci utilizza blocchi le cui dimensioni sono multipli dei termini della serie di Fibonacci, per esempio 16, 32, 48, 80, 128, ... Dunque la dimensione di un blocco è la somma delle dimensioni dei due blocchi immediatamente più piccoli. Questa formula governa la divisione e l'unione dei blocchi. Confrontare l'efficienza dell'esecuzione e l'efficienza della memoria del

- buddy system di Fibonacci rispetto al buddy system binario.
- 11.11. Un allocatore di memoria funziona come segue: le aree di memoria piccole sono allocate utilizzando un buddy system. Le aree di memoria grandi sono allocate utilizzando una free list e un allocatore first-fit. Commentare l'efficienza e l'utilizzo della memoria ottenuta da questo allocatore.
- 11.12. Un SO ha 110 MB disponibili per i processi utente. La richiesta massima di memoria di un processo per il proprio codice e dati è di 20 MB, mentre la richiesta media di memoria di un processo è di 10 MB. Se il SO usa l'allocazione contigua della memoria e non conosce le dimensioni dei singoli processi, qual è la frammentazione interna ed esterna media?
- 11.13. La regola del 50 per cento si applica ai seguenti allocatori?
- buddy system
  - allocatore potenza del 2
  - allocatore slab
- 11.14. Un SO riceve richieste per l'allocazione della memoria a un tasso elevato. Si verifica che una frazione elevata delle richieste riguarda aree di memoria di dimensione 100, 300 e 400 byte (chiamiamole dimensioni "standard"). Altre richieste sono effettuate per aree di varie dimensioni. Progettare uno schema di allocazione della memoria in cui non si verifichi frammentazione nell'allocazione delle aree di dimensione standard e non si verifichi frammentazione interna nell'allocazione delle aree di altre dimensioni.
- 11.15. Calcolare lo slack per ogni classe di buffer se fosse utilizzato un allocatore lazy buddy invece dell'allocatore buddy nel Problema 11.5.
- 11.16. Se il SO del Problema 11.12 utilizzasse la paginazione con una pagina di 2 KB, sarebbe possibile calcolare la frammentazione interna media nel sistema?

## Problemi avanzati

- 11.1. Si consideri un gestore della memoria basato su swapping che utilizza una lista per rappresentare lo stato di allocazione della memoria. Si supponga che la lista contenga le seguenti informazioni:

$$(A, 0, 12) \rightarrow (F, 12, 5) \rightarrow (A, 17, 6) \rightarrow (F, 23, 12) \rightarrow \\ \rightarrow (A, 35, 5) \rightarrow (F, 40, 9) \rightarrow (A, 49, 10)$$

dove il simbolo "A" denota una partizione di memoria occupata da un processo, mentre il simbolo "F" denota una partizione di memoria libera. Supponiamo che al gestore della memoria arrivino in tempi successivi due richieste di allocazione di memoria per due processi, il primo di dimensione 7 e il secondo di dimensione 10. Dire come si comporta il gestore della memoria nel caso si utilizzi la politica di allocazione *best-fit* e nel caso in cui si utilizzi una politica *first-fit* e mostrare in entrambi i casi la lista di allocazione della memoria risultante.

## Note bibliografiche

Linker e loader sono descritti in Dhamdhere (1999).

Knuth (1973) è il punto di partenza per lo studio della gestione della memoria continua. Descrive varie tecniche di allocazione della memoria e strutture dati efficienti per tenere traccia della memoria libera. Hoare e McKeag (1971) propongono varie tecniche di gestione della memoria. Randell (1969) è uno dei primi documenti riguardanti la motivazione dei sistemi con memoria virtuale. Denning (1970) descrive i fondamenti dei sistemi con memoria virtuale.

Vahalia (1996) descrive i vari allocatori di memoria utilizzati nei sistemi Unix. McKusick e Karels (1988) descrivono l'allocatore di memoria McKusick-Karels. Lee e Barkley (1989) descrivono l'allocatore lazy buddy. Entrambi questi allocatori sono utilizzati in Unix. Bonwick (1994) e Bonwick e Adams (2001) descrivono l'allocatore slab. Mauro e McDougall (2006) descrivono l'uso dell'allocatore slab in Solaris, mentre Beck et al. (2002), e Bovet e Cesati (2005) descrivono la sua implementazione in Linux.

Il kernel di Window usa diverse politiche di allocazione della memoria per le proprie richieste di memoria. Implementa l'allocazione sul modello del buddy system per blocchi di medie dimensioni e l'allocazione basata sull'heap per i blocchi di piccole dimensioni. Russinovich e Solomon (2005) descrivono sia l'allocazione heap sia l'allocazione della memoria del kernel in Windows.

1. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
2. Bonwick, J. (1994): "The slab allocator: An object-caching kernel memory allocator," *Proceedings of the Summer 1994 Usenix Technical Conference*, 87-98.
3. Bonwick, J., and J. Adams (2001): "Extending the slab allocator to many CPUs and arbitrary resources," *Proceedings of the 2001 USENIX Annual Technical Conference*, 15-34.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, Calif.
5. Denning, P.J. (1970): "Virtual Memory," *Computing Surveys*, **2** (3), 153-189.
6. Dhamdhere, D.M. (1999): *Systems Programming and Operating Systems*, 2nd revised ed., Tata McGraw-Hill, New Delhi.
7. Hoare, C.A.R., and R. M. McKeag (1971): "A survey of store management techniques," in *Operating Systems Techniques*, by C.A.R. Hoare and R.H. Perrott (eds.) Academic Press, London.
8. Knuth, D.E. (1973): *The Art of Computer Programming*, 2nd ed., Vol. I : Fundamental Algorithms, Addison-Wesley, Reading, Mass.
9. Kuck, D.J., and D.H. Lowrie (1970): "The use and performance of memory hierarchies," in *Software Engineering*, **1**, J.T. Tou (ed.), Academic Press, New York.
10. Lee, T.P., and R.E. Barkley (1989): "A watermark-based lazy buddy system for kernel memory allocation," *Proceedings of the Summer 1989 USENIX Technical Conference*, 1-13.
11. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
12. McKusick, M.K., and M.J. Karels (1988): "Design of a general-purpose memory allocator for the 4.3 BSD Unix kernel," *Proceedings of the Summer 1988 USENIX Technical Conference*, 295-303.
13. Peterson, J.L., and T.A. Norman (1977): "Buddy systems," *Communications of the ACM*, **20** (6), 421-431.
14. Randell, B. (1969): "A note on storage fragmentation and program segmentation," *Communications of the ACM*, **12** (7), 365-369.
15. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
16. Vahalia, U. (1996): *Unix Internals - The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.

---

# CAPITOLO 12

## Memoria virtuale

---

### Obiettivi di apprendimento

- Introduzione alla memoria virtuale
- Paginazione su richiesta
- Politiche di sostituzione delle pagine
- Allocazione della memoria ai processi
- Pagine condivise e file mappati in memoria
- Memoria virtuale con segmentazione
- Memoria virtuale nei sistemi operativi: Unix, Linux, Solaris, Windows

La memoria virtuale è una parte della gerarchia di memoria composta dalla memoria e da un disco rigido. In riferimento al principio delle gerarchie di memoria descritte nel [Capitolo 2](#), solo parti dello spazio di indirizzamento di un processo - cioè il suo codice e i suoi dati - risiedono in memoria a un certo istante; altre parti del suo spazio di indirizzamento si trovano su disco rigido e vengono caricate in memoria solo quando necessario durante l'esecuzione del processo. Il kernel utilizza la memoria virtuale per ridurre la richiesta di memoria da mettere a disposizione di un processo al fine di poter gestire un maggior numero di processi concorrentemente e poter gestire processi il cui spazio di indirizzamento è maggiore delle dimensioni della memoria a disposizione.

La memoria virtuale è implementata attraverso il *modello di allocazione della memoria non contigua* descritto precedentemente nel [Capitolo 11](#) e comprende sia componenti hardware che una componente software detta *virtual memory manager* (gestore della memoria virtuale). Le componenti hardware velocizzano la *traduzione degli indirizzi* e aiutano il gestore della memoria virtuale a eseguire i task in maniera più efficace. Il gestore della memoria virtuale decide quali porzioni dello spazio di indirizzamento di un processo devono essere presenti in memoria di volta in volta.

Le prestazioni della memoria virtuale dipendono dalla frequenza con cui parti dello spazio di indirizzamento di un processo devono essere caricate in memoria da un disco rigido e rimosse dalla memoria per liberare spazio al fine di caricare in memoria altre porzioni dello stesso processo o porzioni di processi differenti. Secondo il principio empirico della *località dei riferimenti*, è probabile che un processo acceda nuovamente a parti del suo spazio di indirizzamento referenziate di recente. Il gestore della memoria virtuale assicura buone prestazioni della memoria virtuale allocando un'adeguata quantità di memoria a un processo e utilizzando un *algoritmo di sostituzione* per rimuovere una porzione che non è stata referenziata di recente.

Cominciamo a illustrare il principio della località dei riferimenti e la sua importanza nelle prestazioni della memoria virtuale. Successivamente tratteremo le tecniche impiegate dal gestore della memoria virtuale per assicurare buone prestazioni.

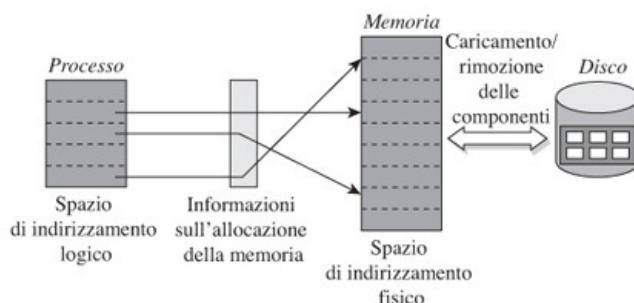
### 12.1 Principi di base della memoria virtuale

Gli utenti richiedono sempre un maggior numero di risorse e servizi a un sistema di elaborazione. La necessità di più risorse viene soddisfatta principalmente ottenendo un utilizzo più efficiente delle risorse esistenti oppure creando l'illusione che esistano più risorse nel sistema. Una *memoria virtuale* è ciò che il suo nome indica: è un'illusione di una memoria più grande di quella reale, ossia della RAM del sistema di elaborazione. Come abbiamo sottolineato nel Paragrafo 1.1, questa illusione fa parte della vista astratta della memoria da parte dell'utente. Un utente o la sua applicazione vede solo una memoria virtuale. Il kernel implementa tale illusione tramite una combinazione di mezzi hardware e software. Ci riferiremo alla memoria reale semplicemente con il termine memoria e ci riferiremo alla componente software della memoria virtuale con il termine *gestore della memoria virtuale*.

L'illusione di una memoria più grande di quella del sistema si evidenzia ogni volta che è attivato un processo la cui dimensione supera quella della memoria. Il processo può funzionare perché risiede nella sua interezza su disco mentre in memoria sono caricate di volta in volta solo le sue parti necessarie. La memoria virtuale si basa sul modello di *allocazione di memoria non contigua* descritto precedentemente nel Paragrafo 11.7. Assumiamo che lo spazio di indirizzamento di ciascun processo consista di parti chiamate *componenti*. Tali parti possono essere caricate in aree di memoria non adiacenti. L'indirizzo di ciascun operando o istruzione nel codice di un processo è un *indirizzo logico* del tipo  $(comp_i, byte_i)$ . L'*unità di gestione della memoria* (MMU) traduce l'indirizzo logico nell'indirizzo di memoria in cui l'operando o l'istruzione si trova effettivamente.

L'uso del modello di allocazione di memoria non contigua riduce la frammentazione di memoria, poiché un'area libera di memoria può essere riutilizzata anche se non è abbastanza grande da contenere l'intero spazio di indirizzamento di un processo. In questo modo, possono essere caricati in memoria più processi utente, un beneficio sia per gli utenti che per il SO. Il kernel estende ulteriormente questa idea, cioè anche i processi che potrebbero stare in memoria non vengono pienamente caricati. Questa strategia riduce la quantità di memoria allocata per ciascun processo, incrementando così ulteriormente il numero di processi che possono essere in esecuzione concorrentemente.

La [Figura 12.1](#) mostra uno schema di memoria virtuale. Lo spazio di indirizzamento logico del processo mostrato consiste di cinque componenti. Tre di queste componenti sono attualmente in memoria. L'informazione relativa alle aree di memoria in cui si trovano queste componenti è tenuta in una struttura dati del gestore della memoria virtuale. Questa informazione è utilizzata dalla MMU nella traduzione degli indirizzi. Quando un'istruzione nel processo fa riferimento a un dato o a un'istruzione che non è in memoria, viene caricata dal disco la componente che la contiene. Occasionalmente, il gestore della memoria virtuale rimuove alcune componenti dalla memoria per fare spazio ad altre componenti.



**Figura 12.1** Schema della memoria virtuale.

Lo schema mostrato in [Figura 12.1](#) è una gerarchia di memoria sul modello discusso nel Paragrafo 2.2.3 e illustrato in [Figura 2.4](#). La gerarchia comprende la memoria del sistema e un disco. La memoria è veloce, ma di piccole dimensioni, mentre il disco rigido è lento, ma ha una capacità di gran lunga maggiore.

La MMU e il gestore della memoria virtuale insieme gestiscono la gerarchia di memoria, in modo tale che l'istruzione corrente in un processo trova i suoi operandi in memoria.

Siamo pronti per definire la memoria virtuale.

**Definizione 12.1 Memoria virtuale** Una gerarchia di memoria, composta dalla memoria del sistema di elaborazione e da un disco, che consente a un processo di funzionare con soltanto alcune porzioni del suo spazio di indirizzamento in memoria.

### **Caricamento su richiesta di componenti di un processo**

Non appena manda in esecuzione un processo, il gestore della memoria virtuale carica solo una porzione dello spazio di indirizzamento di un processo in memoria, cioè quella porzione che contiene l'*indirizzo di start* del processo, cioè l'indirizzo dell'istruzione con

cui la sua esecuzione comincia. Successivamente, carica altre porzioni del processo solo quando sono necessarie. Questa tecnica è detta *caricamento su richiesta*. Per mantenere contenuta la quantità di memoria occupata da un processo, il gestore della memoria virtuale rimuove di tanto in tanto le componenti inutilizzate del processo dalla memoria. Queste porzioni saranno ricaricate di nuovo in memoria solo quando necessario.

Le prestazioni di un processo in memoria virtuale dipendono dalla frequenza con cui le sue componenti devono essere caricate in memoria. Il gestore della memoria virtuale sfrutta il principio della *località dei riferimenti* per raggiungere una bassa percentuale di caricamento delle componenti di processo. Tratteremo questo principio nel Paragrafo 12.2.1.

### **Paginazione e segmentazione**

Nel [Capitolo 11](#) abbiamo discusso di come questi due approcci all'implementazione della memoria virtuale differiscono nel modo in cui sono determinati i limiti e le dimensioni delle porzioni dello spazio di indirizzamento. La [Tabella 12.1](#) confronta i due approcci. Nella paginazione, ciascuna porzione di uno spazio di indirizzamento è chiamata *pagina*. Tutte le pagine hanno uguale dimensione, potenza di due. La dimensione della pagina è definita dall'hardware del computer e la suddivisione delle pagine nello spazio di indirizzamento di un processo viene eseguita automaticamente. Nella segmentazione ogni porzione di uno spazio di indirizzamento è chiamato *segmento*. Un programmatore dichiara alcune entità logiche rilevanti (per esempio, strutture dati o oggetti) in un processo come segmenti. Quindi l'identificazione delle porzioni è a carico del programmatore, e i segmenti possono avere dimensioni differenti. Questa differenza fondamentale conduce a implicazioni differenti per l'utilizzo efficiente della memoria e per la condivisione di programmi o dati. Alcuni sistemi usano un approccio ibrido segmentazione-con-paginazione per ottenere vantaggi da entrambi gli approcci.

| Oggetto                     | Confronto   |
|-----------------------------|---|
| Concetto                    | Una pagina è una porzione a misura fissa di uno spazio di indirizzamento di un processo identificato dall'hardware della memoria virtuale. Un segmento è un'entità logica all'interno di un programma, ossia una funzione, una struttura dati o un oggetto. I segmenti sono identificati dal programmatore. |
| Dimensione delle componenti | Tutte le pagine sono della stessa dimensione. I segmenti possono essere di dimensioni differenti.   |
| Frammentazione esterna      | Non si verifica durante la paginazione perché la memoria è divisa in page frame la cui dimensione è uguale alla dimensione delle pagine. Si verifica nella segmentazione perché un'area libera di memoria può essere piccola per ospitare un segmento.  |
| Frammentazione interna      | Si verifica nell'ultima pagina di un processo durante la paginazione. Non si verifica nella segmentazione perché un segmento è allocato in un'area di memoria la cui dimensione è uguale alla dimensione del segmento.  |
| Condivisione                | La condivisione (sharing) di pagine è realizzabile subordinatamente ai vincoli sulla condivisione delle pagine di codice descritti successivamente nel Paragrafo 12.6. La condivisione di segmenti è possibile liberamente.   |

**Tabella 12.1** Confronto tra paginazione e segmentazione.

## **12.2 Paginazione su richiesta**

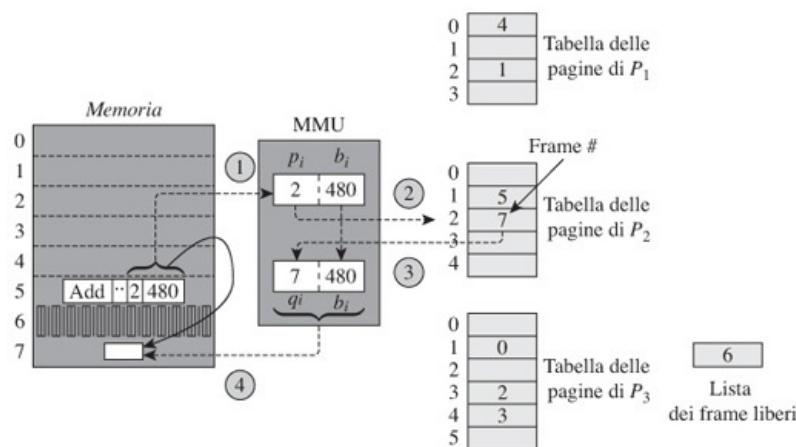
Come discusso precedentemente nel Paragrafo 11.8, si considera che un processo sia composto da pagine, numerate da 0 in avanti. Ogni pagina è di  $s$  byte, dove  $s$  è una potenza di 2. Si considera che la memoria di un sistema di elaborazione sia composta da *frame di pagina*, dove un frame di pagina è un'area di memoria che ha la stessa dimensione di una pagina. Questi sono numerati da 0 a  $\#frame-1$  dove  $\#frame$  è il numero dei frame di pagina di memoria. Dunque, lo spazio di indirizzamento fisico è composto da indirizzi da 0 a  $\#frame \times s - 1$ . In ogni momento, un frame di pagina può

essere libero, o può contenere una pagina di un processo. Ogni indirizzo logico usato in un processo è rappresentato da una coppia  $(p_i, b_i)$ , dove  $p_i$  è un numero di pagina e  $b_i$  è il numero di byte in  $p_i$ ,  $0 \leq b_i < s$ . L'indirizzo effettivo di memoria di un indirizzo logico  $(p_i, b_i)$  è calcolato come segue:

$$\begin{aligned} & \text{indirizzo effettivo di memoria dell'indirizzo logico } (p_i, b_i) \\ & = \text{indirizzo inizio del page frame che contiene la pagina } p_i + b_i \end{aligned} \quad (12.1)$$

La dimensione di una pagina è una potenza di 2, e così il calcolo dell'indirizzo effettivo è eseguito tramite concatenazione di bit, che è molto più veloce dell'addizione (Paragrafo 11.8).

La [Figura 12.2](#) mostra uno schema della memoria virtuale che usa la paginazione in cui si assume che la dimensione di pagina sia 1KB, dove 1KB = 1024 byte. I tre processi  $P_1$ ,  $P_2$  e  $P_3$  hanno alcune delle loro pagine in memoria. La memoria contiene 8 frame numerati da 0 a 7. Le informazioni sull'allocazione di memoria per un processo sono memorizzate in una *tabella delle pagine*. Ogni elemento nella tabella delle pagine contiene le informazioni sull'allocazione di memoria per una pagina di un processo. Contiene il numero del frame in cui una pagina risiede. Il processo  $P_2$  ha le sue pagine 1 e 2 in memoria. Esse occupano rispettivamente i frame 5 e 7. Il processo  $P_1$  ha le sue pagine 0 e 2 nei frame 4 e 1, mentre il processo  $P_3$  ha le sue pagine 1, 3 e 4 nei frame 0, 2 e 3, rispettivamente. La lista dei page frame liberi contiene una lista dei frame liberi. Attualmente solo il frame 6 è libero.



**Figura 12.2** Traduzione degli indirizzi nella memoria virtuale usando la paginazione.

Il processo  $P_2$  sta attualmente eseguendo l'istruzione 'Add .. 2528', quindi la MMU utilizza la tabella delle pagine di  $P_2$  per la traduzione degli indirizzi. La MMU vede l'indirizzo dell'operando 2528 come la coppia  $(2, 480)$  essendo  $2528 = 2 \times 1024 + 480$ . Poi accede all'elemento relativo alla pagina 2 nella tabella delle pagine di  $P_2$ . Questo elemento contiene il numero di frame 7, quindi la MMU calcola l'indirizzo effettivo  $7 \times 1024 + 480$  in base all'Equazione 12.1, e lo utilizza per accedere alla memoria. In effetti, accede al byte 480 nel frame 7.

### 12.2.1 Paginazione su richiesta: concetti preliminari

Se un'istruzione di  $P_2$  in [Figura 12.2](#) fa riferimento a un byte presente nella pagina 3, il gestore della memoria virtuale caricherà la pagina 3 in memoria e metterà il suo numero di frame nell'elemento 3 della tabella delle pagine di  $P_2$ . Queste azioni costituiscono il caricamento su richiesta di pagine o semplicemente *paginazione su richiesta*.

Per implementare la paginazione su richiesta, una copia dell'intero spazio di indirizzamento logico di un processo è mantenuta su un disco. L'area del disco utilizzata per memorizzare questa copia è chiamata *spazio di swap* del processo. Durante l'inizializzazione di un processo, il gestore della memoria virtuale alloca lo spazio di swap

per il processo e copia il suo codice e i suoi dati in tale spazio. Durante il funzionamento del processo, il gestore della memoria virtuale è avvisato quando il processo vuole utilizzare qualche dato o istruzione che è localizzata in una pagina che non è presente in memoria. Carica, allora, la pagina dallo spazio di swap in memoria. Questa operazione è detta operazione di *page-in*. Quando il gestore della memoria virtuale decide di rimuovere una pagina dalla memoria, la pagina è ricopiata nello spazio di swap del processo al quale essa appartiene, nel caso in cui l'ultima pagina sia stata modificata dall'ultima volta che era stata caricata in memoria. Questa operazione è detta operazione di *page-out*. In questo modo, lo spazio di swap di un processo contiene una copia aggiornata di ogni pagina del processo che non è presente in memoria. Un'operazione di sostituzione della pagina è quella che carica una pagina in un frame che già conteneva un'altra pagina. Questo può implicare un'operazione di page-out se la precedente pagina era stata modificata mentre occupava il page frame e implica un'operazione di page-in per caricare la nuova pagina.

In questo paragrafo descriviamo le strutture dati utilizzate dal gestore della memoria virtuale, e il modo in cui esegue le operazioni di page-in, page-out e sostituzione delle pagine. Discuteremo, poi, come il tempo effettivo di accesso alla memoria per un processo dipenda dall'overhead del gestore della memoria virtuale e dal tempo utilizzato dalle operazioni di page-in, page-out e sostituzione delle pagine.

### Tabella delle pagine

La tabella delle pagine di un processo facilita l'implementazione della traduzione degli indirizzi, del caricamento su richiesta e delle operazioni di sostituzione di pagine. La [Figura 12.3](#) mostra il formato di un elemento della tabella delle pagine. Il campo *bit di validità* contiene un valore booleano per indicare se la pagina esiste in memoria. Usiamo la seguente convenzione: 1 indica “residente in memoria” e 0 indica “non residente in memoria”. Il campo *frame #*, descritto precedentemente, facilita la traduzione degli indirizzi. Il campo *misc info* è diviso in quattro sottocampi. L'informazione nel campo *prot info* è usata per proteggere i contenuti della pagina contro le interferenze. Indica se il processo può leggere o scrivere dati nella pagina o eseguire istruzioni su di essa. *ref info* contiene informazioni riguardanti i riferimenti fatti alla pagina mentre era in memoria. Come discuteremo più avanti, questa informazione è usata per prendere decisioni sulla sostituzione della pagina.

| Informazioni varie |         |           |          |            |            |  |
|--------------------|---------|-----------|----------|------------|------------|--|
| Bit di validità    | Frame # | Prot info | Ref info | Modificato | Altre info |  |
|                    |         |           |          |            |            |  |
|                    |         |           |          |            |            |  |
|                    |         |           |          |            |            |  |

| Campo           | Descrizione  |
|-----------------|--|
| Bit di validità | Indica se la pagina descritta dall'elemento attualmente è presente in memoria. Questo bit è anche chiamato bit di presenza.      |
| Frame #         | Indica quale frame di memoria è occupato dalla pagina.   |
| Prot info       | Indica le modalità di utilizzo del contenuto della pagina, se in scrittura, lettura o esecuzione.                                |
| Ref info        | Fornisce informazioni riguardanti i riferimenti fatti alla pagina mentre era in memoria.   |
| Modificato      | Indica se la pagina è stata modificata mentre era in memoria, ovvero se è dirty. Questo campo è un singolo bit, detto dirty bit. |
| Altre info      | Altre informazioni utili relative alla pagina, per esempio, la sua posizione nello spazio di swap.                               |

[Figura 12.3](#) Campi di un elemento della tabella delle pagine.

Il bit *Modificato* indica se la pagina è stata modificata, ossia se è *dirty*. È usato per decidere se è necessaria un'operazione di page-out durante la sostituzione della pagina. Il campo *Altre info* contiene informazioni quali l'indirizzo del blocco del disco nello spazio di swap dove è conservata una copia della pagina.

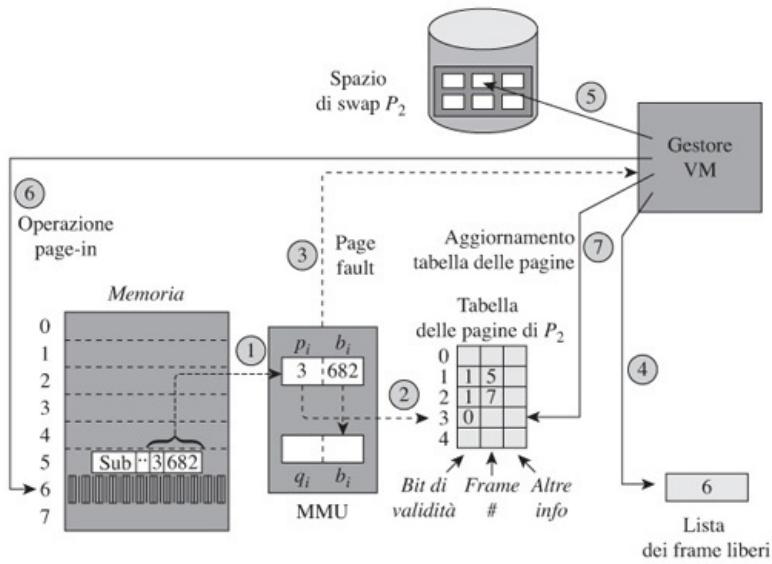
### Page fault e caricamento su richiesta delle pagine

La [Tabella 12.2](#) sintetizza i passi della traduzione degli indirizzi da parte della MMU. Nella traduzione degli indirizzi di un indirizzo logico  $(p_i, b_i)$ , la MMU controlla il bit di validità dell'elemento della tabella delle pagine di  $p_i$  (Passo 2 nella [Tabella 12.2](#)). Se il bit indica che  $p_i$  non è presente in memoria, la MMU genera un interrupt chiamato *missing page interrupt* o *page fault*, che è un interrupt di programma (Paragrafo 2.2.5). La routine di servizio degli interrupt per gli interrupt di programma verifica che l'interrupt è stato causato da un page fault, quindi invoca il gestore della memoria virtuale con il numero di pagina che ha provocato il page fault, cioè,  $p_i$ , come parametro. Il gestore della memoria virtuale carica, quindi, la pagina  $p_i$  in memoria e aggiorna la sua entry nella tabella delle pagine. Di conseguenza, la MMU e il gestore della memoria virtuale interagiscono per decidere *quando* una pagina di un processo deve essere caricata in memoria.

| Passo  | Descrizione   |
|--|---|
| <b>1.</b> Ricavare numero di pagina e numero di byte in una pagina | Un indirizzo logico è visto come una coppia $(p_i, b_i)$ , dove $b_i$ è dato dagli $n_b$ bit meno significativi dell'indirizzo e $p_i$ dato dagli $n_p$ bit più significativi (Paragrafo 11.8).                         |
| <b>2.</b> Ricercare nella tabella delle pagine                     | $p_i$ è usato per indicizzare la tabella delle pagine. Un page fault viene generato se il <i>bit di validità</i> dell'elemento della tabella delle pagine contiene uno 0, ossia se la pagina non è presente in memoria. |
| <b>3.</b> Formare l'indirizzo effettivo di memoria                 | Il campo <i>frame #</i> dell'elemento della tabella delle pagine contiene un numero di frame rappresentato con un numero a $n_f$ bit. Se è concatenato con $b_i$ si ottiene l'effettivo indirizzo di memoria del byte.  |

**Tabella 12.2** Passi nella traduzione degli indirizzi da parte della MMU.

La [Figura 12.4](#) è uno schema delle azioni intraprese dal gestore della memoria virtuale nel caricamento su richiesta di una pagina. Le frecce tratteggiate indicano le azioni della MMU, mentre quelle continue indicano gli accessi alle strutture dati, alla memoria e al disco da parte del gestore della memoria virtuale quando si verifica un page fault. I numeri nei cerchi indicano i passi della traduzione degli indirizzi, nel richiedere e nel gestire i page fault (i Passi 1-3 sono stati descritti precedentemente nella [Tabella 12.2](#)). Il processo  $P_2$  della [Figura 12.2](#) è in esecuzione. Traducendo l'indirizzo logico (3, 682), la MMU genera un page fault perché il bit di validità dell'elemento relativo della pagina 3 è 0. Quando il gestore della memoria virtuale prende il controllo, sa che un riferimento alla pagina 3 ha determinato il page fault. Il campo *Misc info* dell'elemento nella tabella delle pagine della pagina 3 contiene l'indirizzo del blocco del disco nello spazio di swap di  $P_2$  che contiene la pagina 3. Il gestore della memoria virtuale ottiene questo indirizzo. A seguire, consulta la lista dei frame liberi e trova che il page frame 6 è attualmente libero, quindi alloca questo page frame per la pagina 3 e inizia un'operazione di I/O per caricare la pagina 3 nel page frame 6. Quando l'operazione di I/O termina, il gestore della memoria virtuale aggiorna la entry relativa alla pagina 3 nella tabella delle pagine settando il bit di validità a 1 e ponendo 6 nel campo *frame #*. L'esecuzione dell'istruzione "Sub .. (3, 682)", che ha causato il page fault, viene ripristinata. L'indirizzo logico (3, 682) è tradotto nell'effettivo indirizzo del numero di byte 682 nel page frame 6, ossia,  $6 \times 1024 + 682$ .



**Figura 12.4** Richiesta di caricamento di una pagina.

### Operazioni di page-in, page-out e sostituzione delle pagine

La Figura 12.4 ha mostrato come viene eseguita un'operazione di page-in per una pagina richiesta quando si verifica un page fault in un processo ed è disponibile in memoria un page frame libero. Se nessun page frame è libero, il gestore della memoria virtuale esegue un'operazione di sostituzione della pagina per sostituire una delle pagine presenti in memoria con la pagina il cui riferimento aveva determinato il page fault. L'esecuzione avviene come segue: il gestore della memoria virtuale utilizza un *algoritmo di sostituzione delle pagine* per selezionare una delle pagine attualmente in memoria per la sostituzione, accede alla entry della tabella delle pagine relativo alla pagina selezionata per marcarla come "non presente" in memoria, e inizia un'operazione di page-out nel caso in cui il bit *Modificato* della sua entry nella tabella delle pagine indichi che è una pagina *dirty*. Al passo successivo, il gestore della memoria virtuale inizia un'operazione di page-in per caricare la pagina richiesta nel page frame che era occupato dalla pagina selezionata. Al termine dell'operazione di page-in, aggiorna la entry della pagina nella tabella delle pagine per registrare il numero di frame, marca la pagina come "presente", e si predisponde per ripristinare l'esecuzione del processo. Il processo riesegue ora la sua istruzione corrente. Questa volta, la traduzione dell'indirizzo logico nell'istruzione corrente termina senza page fault.

Le operazioni di page-in e page-out richieste per implementare la paginazione su richiesta costituiscono il *page I/O*; usiamo il termine *traffico di pagine* per descrivere i movimenti delle pagine all'interno e all'esterno della memoria. Notare che il page I/O è diverso dalle operazioni di I/O eseguite dai processi, che chiamiamo *program I/O*. Lo stato di un processo che va incontro a un page fault è cambiato a *bloccato* finché la pagina richiesta è caricata in memoria; quindi, la sua prestazione risente del page fault. Il kernel può commutare la CPU a un altro processo per salvaguardare le prestazioni del sistema.

### Tempo effettivo di accesso alla memoria

Il tempo effettivo di accesso alla memoria per un processo nella paginazione su richiesta è il tempo medio di accesso alla memoria atteso dal processo. Dipende da due fattori: il tempo usato dalla MMU per la traduzione degli indirizzi e il tempo medio usato dal gestore della memoria virtuale nella gestione di un page fault. Usiamo la seguente notazione per calcolare il tempo di accesso effettivo alla memoria:

- $pr_1$  probabilità che una pagina esista in memoria
- $t_{\text{mem}}$  tempo di accesso alla memoria
- $t_{\text{pfh}}$  overhead di tempo nella gestione del page fault

$pr_1$  rappresenta la percentuale di successo di trovare un riferimento alla memoria ed è chiamato *memory hit ratio*.  $t_{\text{pfh}}$  è di qualche ordine di grandezza maggiore di  $t_{\text{mem}}$  poiché comprende l'I/O del disco - è richiesta un'operazione di I/O del disco se è sufficiente una sola operazione di page-in, ne sono richieste due se è necessaria una sostituzione di pagina.

Quando un processo è in esecuzione, la sua tabella delle pagine è in memoria. Di conseguenza, per accedere a un operando con l'indirizzo logico  $(p_i, b_i)$  vengono utilizzati due cicli di memoria se la pagina  $p_i$  è presente in memoria: uno per accedere alla entry della tabella delle pagine relativo a  $p_i$  per la traduzione degli indirizzi, e l'altro per accedere all'operando in memoria utilizzando l'indirizzo di memoria effettivo di  $(p_i, b_i)$ . Se la pagina non è presente in memoria, viene generato un page fault dopo aver fatto riferimento alla entry della tabella delle pagine relativo a  $p_i$ , ossia dopo un ciclo di memoria. Viene, ora, caricata in memoria la pagina richiesta e viene aggiornata la sua entry nella tabella delle pagine per registrare il numero di frame dove è stata caricata. Quando viene ripresa l'esecuzione del processo, sono richiesti altri due riferimenti in memoria, cioè uno per accedere alla tabella delle pagine e l'altro per accedere effettivamente all'operando. Di conseguenza, il tempo di accesso effettivo alla memoria è calcolato come segue:

$$\begin{aligned} \text{tempo di accesso effettivo alla memoria} = & pr_1 \times 2 \times t_{\text{mem}} \\ & + (1 - pr_1) \times (t_{\text{mem}} + t_{\text{pfh}} + 2 \times t_{\text{mem}}) \end{aligned} \quad (12.2)$$

Il tempo di accesso effettivo alla memoria può essere migliorato riducendo il numero di page fault. Un modo per farlo è caricare le pagine prima che siano necessarie al processo. Il sistema operativo Windows esegue in maniera speculativa un caricamento simile; in pratica, quando si verifica un page fault, carica la pagina richiesta e anche alcune pagine del processo adiacenti. Quest'azione incrementa il tempo medio di accesso alla memoria se viene riferita dal processo una pagina precaricata. Il sistema operativo Linux consente a un processo di specificare quali pagine dovrebbero essere precaricate. Un programmatore può utilizzare questa caratteristica per migliorare il tempo di accesso effettivo alla memoria.

### **Sostituzione delle pagine**

La sostituzione di pagina diventa necessaria quando si verifica un page fault e non ci sono page frame liberi in memoria. Tuttavia, si potrebbe verificare un altro page fault se la pagina sostituita fosse nuovamente riferita. Di conseguenza, è importante sostituire una pagina che probabilmente non sarà riferita nell'immediatezza. Ma come fa il gestore della memoria virtuale a sapere quale pagina probabilmente non sarà riferita subito dopo?

Il principio empirico della *località dei riferimenti* afferma che gli indirizzi logici usati da un processo in un breve intervallo di tempo durante il suo funzionamento tendono a essere raggruppati in alcune porzioni del suo spazio di indirizzamento logico. I processi manifestano questo comportamento per due motivi. L'esecuzione delle istruzioni in un processo è per lo più sequenziale in natura, perché solo il 10-20 per cento delle istruzioni eseguite da un processo sono istruzioni di salto. I processi tendono, inoltre, a eseguire operazioni simili su diversi tipi di dati non scalari come gli array. A causa dell'effetto combinato di questi due motivi, i riferimenti a istruzioni e dati fatti da un processo tendono a essere strettamente prossimi ai riferimenti alle istruzioni e ai dati fatti precedentemente.

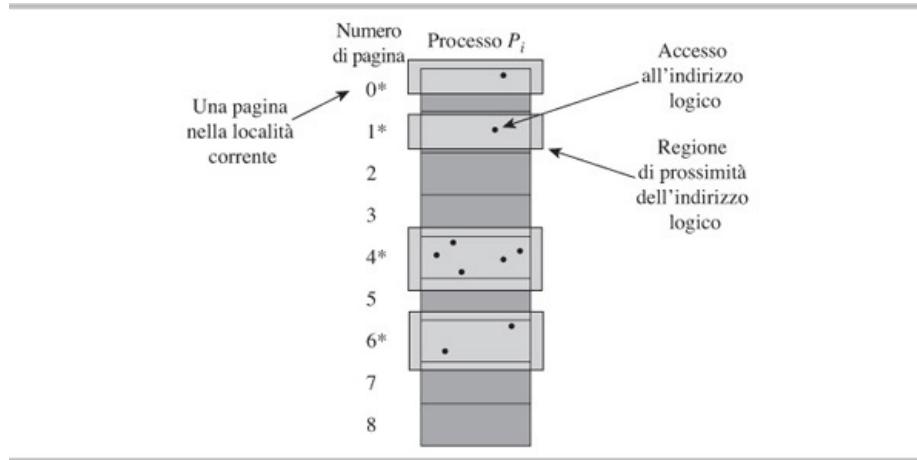
Definiamo *località corrente* di un processo l'insieme delle pagine riferite nelle sue poche istruzioni precedenti. Quindi, il principio di località indica che l'indirizzo logico usato in una istruzione probabilmente si riferisce a una pagina che si trova nella località corrente del processo. Come menzionato nel Paragrafo 2.2.3, il computer sfrutta il principio di località per assicurare hit ratio elevati nella cache. Il gestore della memoria virtuale può sfruttare il principio di località per ottenere un effetto analogo, dovrebbero cioè verificarsi meno page fault assicurando che le pagine che sono nella località corrente di un processo siano presenti in memoria.

Notare che la località dei riferimenti non implica un'assenza di page fault. Supponiamo che una *regione di prossimità* di un indirizzo logico  $a_i$  contenga tutti gli indirizzi logici

che sono strettamente prossimi ad  $a_i$ . Si possono verificare page fault per due motivi. Primo, la regione di prossimità di un indirizzo logico può non entrare in una pagina; in questo caso, il prossimo indirizzo può stare in una pagina adiacente che non è inclusa nella località corrente del processo. Secondo, un'istruzione o dato riferito da un processo può non essere in prossimità dei precedenti riferimenti. Chiamiamo questa situazione *shift nella località* di un processo. Tipicamente si verifica quando un processo fa una transizione da un'azione nel suo spazio logico a un altro. Il prossimo esempio illustra la località di un processo.

### Esempio 12.1 Località corrente di un processo

In [Figura 12.5](#), i cerchietti pieni indicano i pochi indirizzi logici usati durante l'esecuzione di un processo  $P_i$ . I riquadri tratteggiati mostrano le regioni di prossimità di questi indirizzi logici. Notare che la regione di prossimità di un indirizzo logico si può estendere oltre la frontiera della pagina. Le regioni di prossimità degli indirizzi logici possono inoltre sovrapporsi. Mostriamo le regioni di prossimità cumulative in [Figura 12.5](#); ossia le regioni di prossimità degli indirizzi logici riferiti nella pagina 4 cumulativamente coprono l'intera pagina 4 e parti delle pagine 3 e 5. Quindi, le regioni di prossimità si trovano nelle pagine 0, 1, 3, 4, 5, 6 e 7; comunque, la località corrente di  $P_i$  è l'insieme delle pagine i cui numeri sono marcati con i simboli \* in [Figura 12.5](#), ossia l'insieme delle pagine {0, 1, 4, 6}.



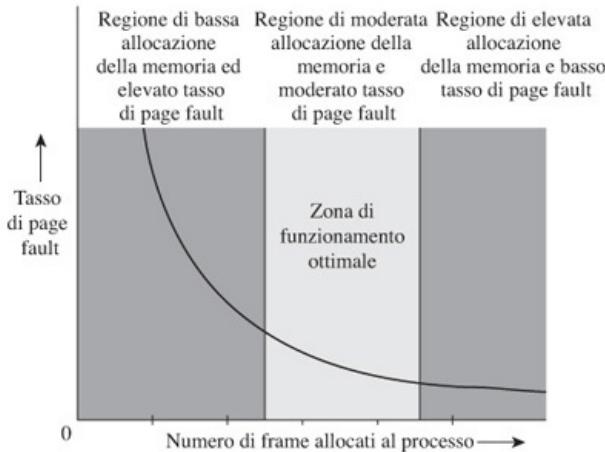
**Figura 12.5** Regioni di prossimità dei riferimenti precedenti e località corrente di un processo.

Il principio di località aiuta a scegliere quale pagina debba essere sostituita quando si verifica un page fault. Supponiamo che il numero di page frame allocati a un processo  $P_i$  sia costante; di conseguenza, ogni volta che si verifica un page fault durante l'esecuzione di  $P_i$ , una delle pagine di  $P_i$  esistenti in memoria deve essere sostituita. Siano  $t_1$  e  $t_2$  i periodi di tempo per cui le pagine  $p_1$  e  $p_2$  non sono state riferite durante l'esecuzione di  $P_i$ . Sia  $t_1 > t_2$ , il che implica che alcuni byte della pagina  $p_2$  sono stati riferiti o eseguiti (come istruzione) più recentemente di alcuni byte della pagina  $p_1$ . Di conseguenza, la pagina  $p_2$  è più probabile che sia parte della località corrente del processo rispetto alla pagina  $p_1$ ; cioè, un byte della pagina  $p_2$  è più probabile che sia stato riferito o eseguito rispetto a un byte della pagina  $p_1$ . Utilizziamo questo elemento per scegliere la pagina  $p_1$  per la sostituzione quando si verifica un page fault. Se esistono molte pagine di  $P_i$  in memoria, possiamo attribuire un peso in base al numero dei loro ultimi riferimenti e sostituire la pagina che è stata meno recentemente riferita. Questa politica di sostituzione di pagina è detta *sostituzione delle pagine LRU*.

### Allocazione di memoria per un processo

La [Figura 12.6](#) mostra come la percentuale dei page fault di un processo vari in funzione della quantità di memoria a esso allocata. La percentuale dei page fault è maggiore se una piccola quantità di memoria è allocata per il processo; comunque, diminuisce se viene allocata più memoria per il processo. Questa caratteristica sui page fault di un processo è interessante perché consente al gestore della memoria virtuale di

intraprendere azioni correttive quando verifica che un processo ha un'alta percentuale di page fault; può, per esempio, comportare una riduzione nella percentuale di page fault aumentando la memoria allocata per il processo. Come discuteremo nel Paragrafo 12.4, la politica di sostituzione di pagina LRU ha una caratteristica sui page fault che è simile alla curva di [Figura 12.6](#) perché sostituisce una pagina che è meno probabile che sia nella località corrente del processo rispetto ad altre pagine del processo che sono in memoria.



**Figura 12.6** Variazione ottimale della percentuale di page fault rispetto all'allocazione della memoria.

Quanta memoria il gestore della memoria virtuale dovrebbe allocare per un processo? Due fattori opposti influenzano questa decisione. Dalla [Figura 12.6](#), vediamo che un'allocazione sovradimensionata di memoria per un processo implica una bassa percentuale di page fault per il processo; questo assicura buone prestazioni del processo. Tuttavia, un basso numero di processi entrerebbe in memoria e ciò potrebbe causare che la CPU sia idle e, pertanto, scarse prestazioni di sistema. Un'allocazione sottodimensionata di memoria per un processo implica un'alta percentuale di page fault, il che condurrebbe a scarse prestazioni per il processo. La zona operativa ottimale evidenziata in [Figura 12.6](#) evita le regioni di sovradimensionamento e sottodimensionamento della memoria.

Il problema principale nello stabilire quanta memoria allocare per un processo è legato alle caratteristiche dei page fault; per esempio, l'andamento della curva e la frequenza dei page fault, in [Figura 12.6](#), varia tra i processi. Anche per lo stesso processo, le caratteristiche dei page fault possono essere differenti se il processo viene eseguito con dati diversi. Di conseguenza, la quantità di memoria da allocare a un processo deve essere determinata dinamicamente considerando le caratteristiche attuali dei page fault del processo. Questo argomento è affrontato nel Paragrafo 12.5.

### Thrashing

Consideriamo un processo che è in esecuzione nella regione a bassa allocazione di memoria e ad alta frequenza di page fault in [Figura 12.6](#). A causa dell'alta frequenza di page fault, questo processo consuma molto del suo tempo in stato *bloccato*. Un tale processo non sta utilizzando la CPU in maniera efficace. Ciò causa, inoltre, un elevato overhead dovuto all'alta frequenza di page fault e alla commutazione di processi causata dai page fault. Se tutti i processi nel sistema fossero in esecuzione nella regione ad alta percentuale di page fault, la CPU sarebbe impegnata la maggior parte del tempo nel traffico di pagine e nella commutazione di processi. L'efficienza della CPU sarebbe scarsa e la prestazione di sistema, misurata o in termini di tempo medio di risposta o come throughput, sarebbe scarsa. Questa situazione è detta *thrashing*.

**Definizione 12.2 Thrashing** Una condizione in cui coincidono un elevato traffico di pagine e una scarsa efficienza di CPU.

Da notare che la scarsa efficienza della CPU si può verificare anche per altre cause, ossia, se esistono in memoria troppo pochi processi o tutti i processi in memoria eseguono operazioni di I/O frequentemente. La situazione del thrashing è differente nel senso che *tutti* i processi hanno un progresso scarso a causa delle alte percentuali di page fault.

Dalla [Figura 12.6](#), possiamo dedurre che la causa del thrashing è un'allocazione sottostimata di memoria per ogni processo. Questo si può risolvere incrementando l'allocazione di memoria per ciascun processo. Ciò si potrebbe realizzare rimuovendo alcuni processi dalla memoria, cioè riducendo il grado di multiprogrammazione. Un processo può avere individualmente un'alta percentuale di page fault senza il thrashing di sistema. La stessa analisi può essere applicata anche al processo; esso risente di un'allocazione sottostimata di memoria, per cui la soluzione è incrementare la memoria a esso allocata.

### **Dimensione di pagina ottimale**

La dimensione di una pagina è definita dall'hardware del computer. Ciò determina il numero di bit richiesti per rappresentare il numero di byte di una pagina. La dimensione di una pagina influisce, inoltre, sui seguenti elementi:

1. spreco di memoria dovuto a frammentazione interna;
2. dimensione della tabella delle pagine per un processo;
3. frequenze di page fault quando è allocata una quantità fissata di memoria per un processo.

Consideriamo un processo  $P_i$  di dimensione  $z$  byte. La dimensione di pagina di  $s$  byte comporta che il processo abbia  $n$  pagine, dove  $n = \lceil z/s \rceil$  è il valore di  $z/s$  arrotondato per eccesso. La frammentazione interna media è  $s/2$  byte perché l'ultima pagina dovrebbe essere mezza vuota in media. Il numero di elementi nella tabella delle pagine è  $n$ . Quindi, la frammentazione interna varia direttamente con la dimensione della pagina, mentre quest'ultima varia inversamente ad essa.

Va notato che la percentuale di page fault varia anche con la dimensione di pagina se viene allocata una quantità fissata di memoria per  $P_i$ . Ciò può essere spiegato in questo modo: il numero di pagine di  $P_i$  in memoria varia inversamente alla dimensione della pagina. Di conseguenza, esisterebbero in memoria due o più pagine di  $P_i$  se la dimensione di pagina fosse  $s/2$ . Supponiamo ora che la regione di prossimità di un'istruzione o byte di dati, come definito nel [Paragrafo 12.2](#), sia piccola rispetto a  $s/2$ ; per esempio, si può assumere che questa regione entri all'interno della pagina che contiene il byte. Quando la dimensione di pagina è  $s/2$ , la memoria contiene due o più regioni di prossimità di indirizzi logici recenti come quando la dimensione di pagina è  $s$  byte. Dalla caratteristica del page fault di [Figura 12.6](#), le percentuali di page fault dovrebbero essere più piccole per dimensioni di pagina inferiori.

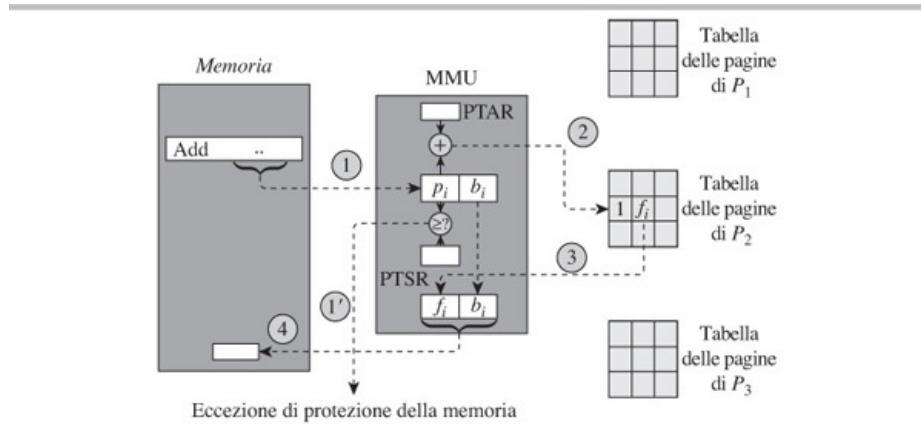
Possiamo calcolare la dimensione di pagina che minimizza il totale di penalizzazione della memoria dovuto alla frammentazione interna e all'allocazione di memoria per la tabella delle pagine. Se  $s \ll z$  e ogni entry della tabella delle pagine occupa 1 byte di memoria, il valore ottimale di  $s$  è  $\sqrt{2z}$ . Quindi, la dimensione di pagina ottimale è solo 400 byte per una dimensione di processo di 80 KB, ed è 800 byte per un processo di 320 KB. Tuttavia, i sistemi di elaborazione tendono a utilizzare dimensioni di pagina maggiori (per esempio, Pentium e MIPS usano dimensioni di pagina di 4 KB o più, Sun Ultrasparc usa dimensioni di pagina di 8 KB o più e il PowerPC usa una dimensione di pagina di 4 KB) per le seguenti ragioni:

1. le entry della tabella delle pagine tendono a occupare più di 1 byte;
2. i costi dell'hardware sono alti per dimensioni di pagina più piccole. Per esempio, il costo della traduzione degli indirizzi aumenta se viene usato un maggior numero di bit per rappresentare un numero di pagina;
3. i dischi, che sono usati come dispositivi per la paginazione, funzionano meno efficientemente per dimensioni di blocchi del disco più piccoli.

La decisione di utilizzare dimensioni di pagina più grandi rispetto al valore ottimale comporta, in qualche modo, maggiori percentuali di page fault per un processo. Questo fatto rappresenta un compromesso tra il costo dell'hardware e l'esecuzione efficiente di un processo.

## 12.2.2 Supporto hardware alla paginazione

La Figura 12.7 illustra la traduzione degli indirizzi in un sistema multiprogrammato. Le tabelle delle pagine per molti processi sono presenti in memoria. La MMU contiene un registro speciale detto *registro degli indirizzi della tabella delle pagine* (PTAR) per puntare all'inizio di una tabella delle pagine. Per un indirizzo logico ( $p_i, b_i$ ), la MMU calcola  $<\text{PTAR}> + p_i \times l_{\text{PT\_entry}}$  per ottenere l'indirizzo della entry della pagina  $p_i$  nella tabella delle pagine, dove  $l_{\text{PT\_entry}}$  è la lunghezza di una entry della tabella delle pagine e  $<\text{PTAR}>$  denota il contenuto della PTAR. La PTAR deve essere caricata con l'indirizzo corretto quando viene schedulato un processo. Per facilitare questo, il kernel può memorizzare l'indirizzo della tabella delle pagine di ogni processo nel suo blocco di controllo del processo (PCB).



**Figura 12.7** Traduzione degli indirizzi in un sistema multiprogrammato.

La Tabella 12.3 riassume le funzioni eseguite dal paging hardware. Descriviamo le tecniche usate nell'implementazione di queste funzioni e nominiamo alcuni sistemi di elaborazione moderni che le usano.

| Funzione                                  | Descrizione  |
|---|--|
| Protezione della memoria                  | Assicurare che un processo acceda solo a quelle aree di memoria allocate per esso.   |
| Efficiente traduzione degli indirizzi     | Fornire una soluzione per eseguire la traduzione degli indirizzi in maniera efficiente.  |
| Supporto per la sostituzione delle pagine | Collezionare informazioni relative ai riferimenti fatti alle pagine. Il gestore della memoria virtuale utilizza questa informazione per decidere quale pagina sostituire quando si verifica un page fault. |

**Tabella 12.3** Funzioni di supporto hardware fornito alla paginazione.

### Protezione della memoria

Un interrupt di *violazione di protezione della memoria* viene generato allorquando un processo tenta di accedere a una pagina non esistente, oppure se va oltre i suoi privilegi di accesso, accedendo a una pagina a cui non può accedere. La MMU fornisce un registro speciale chiamato il *registro della dimensione della tabella delle pagine* (PTSR) per individuare violazioni del primo tipo. Il kernel registra il numero di pagine contenute in un processo nel suo *process control block* (PCB) e carica questo numero dal PCB nel PTSR quando viene schedulato il processo. Viene generata una violazione della protezione di memoria se il numero di pagina in un indirizzo logico non è inferiore rispetto al contenuto del PTSR; questo controllo è analogo a quello che utilizza il registro della dimensione nello schema di protezione della memoria del Capitolo 2.

I privilegi di accesso di un processo a una pagina sono memorizzati nel campo *prot info* della entry della pagina nella tabella delle pagine. Durante la traduzione degli indirizzi, la MMU controlla il tipo di accesso effettuato alla pagina attraverso questa

informazione e genera una violazione di protezione della memoria se i due non sono compatibili. L'informazione nel campo *prot info* può essere bit-encoded per un accesso efficiente: ogni bit nel campo corrisponde a un tipo di accesso alla pagina (per esempio, read, write, ecc.) ed è settato a "on" solo se il processo possiede il corrispondente privilegio di accesso alla pagina.

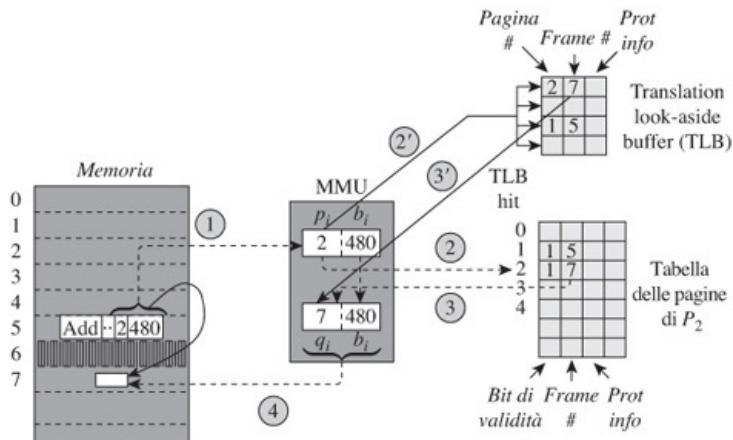
### Traduzione degli indirizzi e generazione dei page fault

La MMU segue i passi della [Tabella 12.2](#) per eseguire la traduzione degli indirizzi. Per un indirizzo logico  $(p_i, b_i)$ , esso accede alla entry della tabella delle pagine di  $p_i$  utilizzando  $p_i \times l_{PT\_entry}$  come offset nella tabella delle pagine, dove  $l_{PT\_entry}$  è la lunghezza di una entry nella tabella delle pagine.  $l_{PT\_entry}$  è tipicamente una potenza di 2, quindi  $p_i \times l_{PT\_entry}$  può essere calcolato efficientemente shiftando il valore di  $p_i$  di pochi bit.

### Buffer di traduzione degli indirizzi

Un riferimento alla tabella delle pagine durante la traduzione degli indirizzi utilizza un ciclo di memoria perché la tabella delle pagine è memorizzata in memoria. Il *translation look-aside buffer* (TLB) è una memoria associativa piccola e veloce usata per eliminare il riferimento alla tabella delle pagine, velocizzando così la traduzione degli indirizzi. Il TLB contiene entry nella forma (pagina #, page frame #, protection info) per poche pagine di un programma a cui è stato fatto accesso di recente e che sono in memoria. Durante la traduzione degli indirizzi di un indirizzo logico  $(p_i, b_i)$ , il supporto hardware cerca nel TLB una entry relativa alla pagina  $p_i$ . Se la trova, viene utilizzato il frame # dalla entry per completare la traduzione degli indirizzi per l'indirizzo logico  $(p_i, b_i)$ .

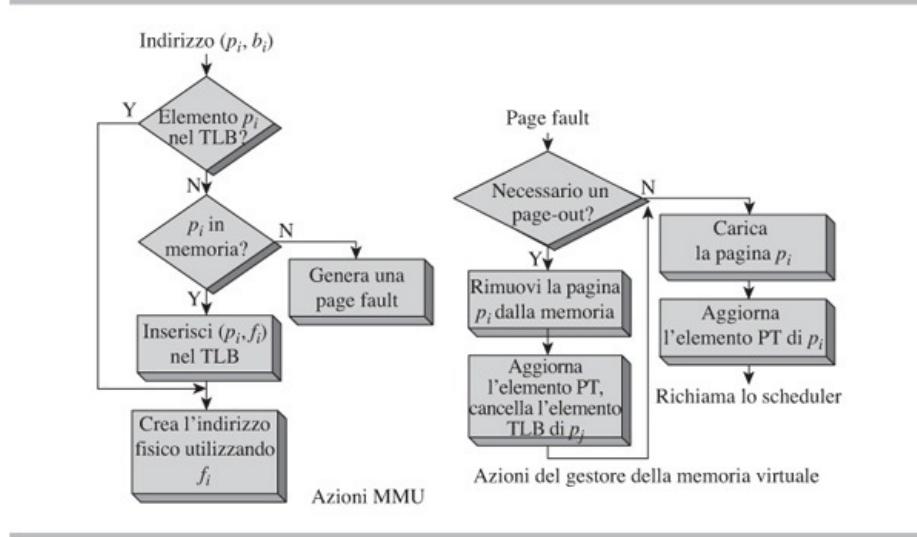
La [Figura 12.8](#) illustra il funzionamento del TLB. Le frecce evidenziate 2' e 3' indicano il TLB lookup. Il TLB contiene le entry per le pagine 1 e 2 del processo  $P_2$ . Se  $p_i$  è 0 o 1 oppure 2, il TLB lookup guadagna un "hit", quindi la MMU prende il numero di page frame dal TLB e completa la traduzione degli indirizzi. Si verifica un TLB "miss" se  $p_i$  è qualche altra pagina, e di conseguenza la MMU accede alla tabella delle pagine e completa la traduzione degli indirizzi se la pagina  $p_i$  è presente in memoria; altrimenti, genera un page fault, che attiva il gestore della memoria virtuale per caricare  $p_i$  in memoria.



**Figura 12.8** Traduzione degli indirizzi utilizzando il translation look-aside buffer e la tabella delle pagine.

La [Figura 12.9](#) sintetizza le azioni della MMU e i passi software nella traduzione degli indirizzi e nella gestione del page fault per un indirizzo logico  $(p_i, b_i)$ . Le azioni MMU riguardanti l'uso del TLB e della tabella delle pagine sono state descritte precedentemente. Il gestore della memoria virtuale è attivato da un page fault. Se non è disponibile un frame di pagina vuoto per caricare la pagina  $p_i$ , inizia un'operazione di page-out di qualche pagina  $p_j$  per liberare il frame, per esempio il page frame  $f_j$ , da esso occupato. La entry nella tabella delle pagine di  $p_j$  viene aggiornata per indicare che non

è più presente in memoria. Se  $p_j$  ha una entry nel TLB, il gestore della memoria virtuale la cancella eseguendo un'istruzione di "erase TLB entry". Quest'azione è essenziale per evitare traduzioni errate di indirizzi al successivo riferimento a  $p_j$ . Viene eseguita successivamente un'operazione di page-in per caricare  $p_i$  nel frame  $f_j$ , e la entry nella tabella delle pagine di  $p_i$  viene aggiornata quando è completata l'operazione di page-in. L'esecuzione dell'istruzione che ha causato il page fault è ripetuta quando il processo viene nuovamente schedulato. Questa volta  $p_i$  non ha una entry nel TLB ma esiste in memoria e, quindi, la MMU usa l'informazione nella tabella delle pagine per completare la traduzione degli indirizzi. Viene a questo punto inserita una entry per  $p_i$  nel TLB.



**Figura 12.9** Schema di traduzione degli indirizzi di  $(p_i, b_i)$  (PT = tabella delle pagine).

Nuove entry nel TLB possono essere inserite o dall'hardware o dal gestore della memoria virtuale. La gestione hardware del TLB è più efficiente; l'hardware può creare una nuova entry nel TLB ogni volta che deve completare una traduzione di indirizzi tramite un riferimento alla tabella delle pagine. Quando il TLB è gestito dal gestore della memoria virtuale, la MMU genera un interrupt "missing TLB entry" ogni volta che non riesce a trovare una entry per la pagina richiesta nel TLB, e il gestore della memoria virtuale esegue diverse istruzioni per crearla nel TLB. Secondo questo approccio, la MMU esegue traduzioni di indirizzi esclusivamente tramite il TLB, e la tabella delle pagine è utilizzata solo dal gestore della memoria virtuale. Questa soluzione consente flessibilità perché il gestore della memoria virtuale può usare differenti organizzazioni della tabella delle pagine per risparmiare memoria (Paragrafo 12.2.3). Il PowerPC e le architetture Intel 80 × 86 usano TLB hardware-managed, mentre i MIPS, Sparc, Alpha e le architetture PA-RISC usano TLB software-managed.

Alcune caratteristiche sono comuni a entrambi gli approcci. Viene usato un algoritmo di sostituzione per decidere quale entry nel TLB dovrebbe essere sovrascritta quando una nuova entry deve essere creata. L'uso del TLB può inficiare la protezione se la MMU esegue traduzioni di indirizzi tramite le entry nel TLB create durante l'esecuzione di qualche altro processo. Questo problema è analogo al problema di protezione nella cache discusso precedentemente nel Paragrafo 2.2.3. Di conseguenza, anche le soluzioni sono analoghe. Ogni entry nel TLB può contenere l'identificativo del processo in esecuzione quando la entry è stata creata; cioè, ogni entry nel TLB può avere la forma (id di processo, page #, frame #, info protezione), in modo che la MMU possa evitare di usarla quando qualche altro processo è in esecuzione. Alternativamente, il kernel deve ripulire il TLB durante l'esecuzione della commutazione dei processi (process switching).

Usiamo la seguente notazione per calcolare il tempo effettivo di accesso alla memoria quando viene usato un TLB:

$pr_1$  probabilità che una pagina esista in memoria;

$pr_2$  probabilità che una entry di pagina esista nel TLB;

- $t_{\text{mem}}$  tempo di accesso in memoria;
- $t_{\text{TLB}}$  tempo di accesso al TLB;
- $t_{\text{pfh}}$  overhead di tempo per la gestione dei page fault.

Come menzionato precedentemente nel Paragrafo 12.2.1,  $pr_1$  è il memory hit ratio e  $t_{\text{mem}}$  è di qualche ordine di grandezza inferiore rispetto a  $t_{\text{pfh}}$ . Tipicamente  $t_{\text{TLB}}$  è almeno un ordine di grandezza inferiore rispetto a  $t_{\text{mem}}$ .  $pr_2$  è detto *TLB hit ratio*. Quando non è usato il TLB, il tempo effettivo di accesso in memoria è dato dall'Equazione 12.2. Si deve eseguire l'accesso alla tabella delle pagine solo se la pagina a cui si fa riferimento non ha una entry nel TLB. Di conseguenza, un riferimento a una pagina impiega un tempo ( $t_{\text{TLB}} + t_{\text{mem}}$ ) se la pagina ha una entry nel TLB, e un tempo ( $t_{\text{TLB}} + 2 \times t_{\text{mem}}$ ) se non ha una entry nel TLB ma esiste in memoria. La probabilità della seconda situazione è  $(pr_1 - pr_2)$ . Quando il TLB è usato,  $pr_2$  rappresenta la probabilità che una entry per la pagina richiesta esista nel TLB. La probabilità che un riferimento alla tabella delle pagine è sia necessaria che sufficiente per la traduzione degli indirizzi è  $(pr_1 - pr_2)$ . Il tempo impiegato da ogni riferimento di questo tipo è  $(t_{\text{TLB}} + 2 \times t_{\text{mem}})$  poiché la ricerca nel TLB senza successo precederebbe quella nella tabella delle pagine. La probabilità di un page fault è  $(1 - pr_1)$ . Esso si verifica dopo che sono stati esaminati il TLB e la tabella delle pagine; ciò richiede un tempo ( $t_{\text{pfh}} + t_{\text{TLB}} + 2 \times t_{\text{mem}}$ ) se supponiamo che l'entry nel TLB sia creata per la pagina mentre si sta calcolando l'indirizzo effettivo di memoria. Di conseguenza, il tempo effettivo di accesso alla memoria è

$$\begin{aligned} \text{Tempo effettivo di accesso alla memoria} = \\ pr_2 \times (t_{\text{TLB}} + t_{\text{mem}}) + (pr_1 - pr_2) \times (t_{\text{TLB}} + 2 \times t_{\text{mem}}) \\ + (1 - pr_1) \times (t_{\text{TLB}} + t_{\text{mem}} + t_{\text{pfh}} + t_{\text{TLB}} + 2 \times t_{\text{mem}}) \end{aligned} \quad (12.3)$$

Per realizzare accessi efficienti alla memoria durante l'elaborazione del kernel, la maggior parte dei computer forniscono *wired TLB entry* per le pagine del kernel. Queste entry non sono mai modificate dagli algoritmi di sostituzione.

### **Superpagine**

La dimensione delle memorie dei computer e dei processi sono cresciute rapidamente dagli anni 1990. Le dimensioni dei TLB non sono andate di pari passo perché i TLB sono costosi a causa della loro natura associativa; le loro dimensioni sono cresciute da circa otto negli anni 1960 a solo circa un migliaio nel 2005. Di conseguenza il *TLB reach*, ovvero il prodotto del numero di entry in un TLB per la dimensione della pagina, è cresciuto marginalmente, ma il suo rapporto rispetto alla dimensione della memoria è diminuito di un fattore oltre 1000. Di conseguenza, gli hit ratio dei TLB sono scarsi e i tempi medi di accesso alla memoria sono elevati (Equazione 12.3). Le cache del processore sono diventate maggiori del TLB reach, il che impatta sulle prestazioni della cache nella quale la ricerca avviene tramite indirizzi fisici perché l'accesso ai contenuti della cache può essere rallentato dai miss e lookup dei TLB attraverso la tabella delle pagine. Un modo generale per tenere in considerazione questi problemi è utilizzare una dimensione maggiore di pagina, in modo che i TLB reach diventino maggiori. Tuttavia, ciò porta a una maggiore frammentazione interna e più pagine I/O. In assenza di una soluzione generale, sono state sviluppate tecniche per rispondere a problemi particolari creati dal basso TLB reach. La ricerca nella cache tramite indirizzi logici escluse il TLB dal percorso dalla CPU alla cache, il che evitò il rallentamento della ricerca nella cache dovuto a limitati TLB reach. Comunque, scarsi TLB hit ratio continuarono a degradare le prestazioni della memoria virtuale.

Le superpagine sono sorte come soluzione generale ai problemi causati da basso TLB reach. Una *superpagina* è come una pagina di un processo, la sua dimensione è una potenza di 2 multiplo della dimensione di una pagina, e il suo indirizzo iniziale in entrambi gli spazi di indirizzamento (sia logico sia fisico) è allineato a un multiplo della sua dimensione. Questa caratteristica aumenta il TLB reach senza aumentare la dimensione del TLB, e aiuta a ottenere un maggiore TLB hit ratio. La maggior parte delle moderne architetture ammettono alcune dimensioni standard per le superpagine e prevedono un campo ulteriore nel TLB entry per indicare la dimensione della superpagina alla quale si può accedere tramite la entry.

Il gestore della memoria virtuale supporta la tecnica delle superpagine adattando la dimensione e il numero delle superpagine in un processo alle sue caratteristiche di esecuzione. Mette insieme alcune pagine di un processo in una superpagina di una dimensione appropriata, le pagine ad accesso frequente e che soddisfano il requisito di contiguità e l'allineamento degli indirizzi nello spazio di indirizzamento logico. Quest'azione è detta *promotion*. Il gestore della memoria virtuale può dover spostare le singole pagine in memoria durante la promotion per assicurare la contiguità e l'allineamento degli indirizzi in memoria. La promotion incrementa il TLB reach e libera alcune delle TLB entry che erano assegnate a pagine singole della nuova superpagina.

Se il gestore della memoria virtuale verifica che alcune pagine in una superpagina non sono ad accesso frequente, può decidere di suddividere la superpagina in pagine singole. Quest'azione, chiamata *demotion*, libera memoria che può essere usata per caricare altre pagine. Così, è possibile ridurre la frequenza dei page fault.

### **Supporto per la sostituzione delle pagine**

Il gestore della memoria virtuale ha bisogno di due tipi di informazioni per minimizzare i page fault e il numero delle operazioni di page-in e page-out durante la sostituzione di pagina:

1. l'ultimo istante di utilizzo di una pagina;
2. se una pagina è *dirty*, ossia se un'operazione di scrittura è stata eseguita su qualche byte nella pagina. (Una pagina è *clean* se non è *dirty*.)

L'istante di ultimo utilizzo indica quanto di recente una pagina è stata usata da un processo; è utile per selezionare un candidato per la sostituzione di pagina. Tuttavia, fornire un sufficiente numero di bit in una entry di una tabella delle pagine per questo scopo è costoso, quindi la maggior parte dei sistemi di elaborazione prevedono un singolo bit chiamato bit di *riferimento*. Il bit di *modifica* in una entry della tabella delle pagine è usato per indicare se una pagina è *clean* oppure *dirty*. Se una pagina è *clean*, la sua copia nello spazio di swap del processo è ancora attuale, quindi non è necessaria alcuna operazione di page-out; la pagina che si sta caricando può semplicemente sovrascrivere tale pagina in memoria. Per una pagina *dirty* è necessaria un'operazione di page-out perché la sua copia nello spazio di swap è vecchia. Un'operazione di page-in per la nuova pagina da caricare può essere avviata solo dopo che l'operazione di page-out è stata completata.

### **12.2.3 Organizzazione pratica delle tabelle delle pagine**

Un processo con un ampio spazio di indirizzamento richiede un'ampia tabella delle pagine. Di conseguenza, il gestore della memoria virtuale deve allocare un grosso quantitativo di memoria per ogni tabella delle pagine. Per esempio, in un sistema di elaborazione che usa indirizzi logici a 32 bit e una dimensione di pagina di 4 KB, un processo può arrivare ad avere 1 milione di pagine. Se la dimensione di una entry della tabella delle pagine è 4 byte, la tabella delle pagine ha una dimensione di 4 MB. Quindi, il gestore della memoria virtuale potrebbe mettere insieme alcune centinaia di megabyte di memoria per la memorizzazione delle tabelle delle pagine dei processi! Le richieste di memoria sarebbero anche maggiori nel caso in cui si utilizzassero indirizzi logici a 64 bit. Vengono seguiti due approcci per ridurre la dimensione della memoria assegnata alle tabelle delle pagine.

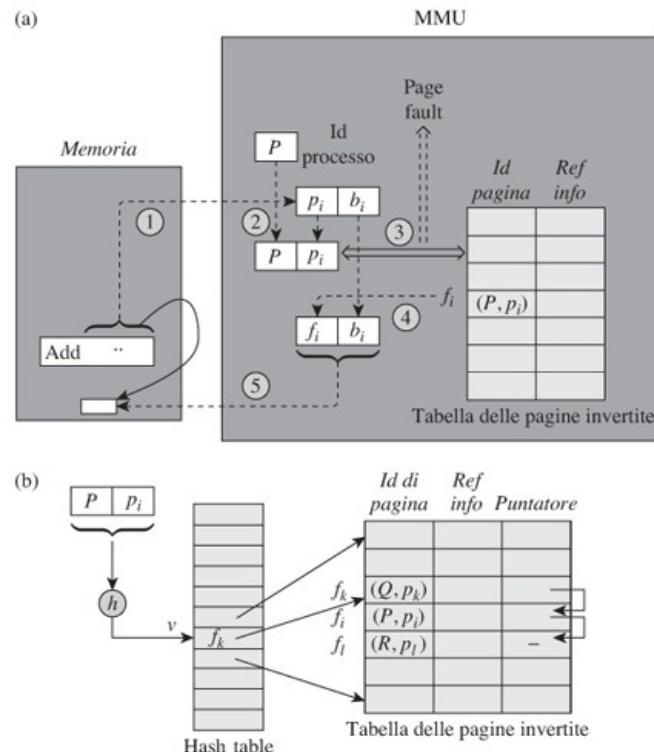
- *Tabella delle pagine invertita*: la tabella delle pagine invertita (IPT) ha una entry per ogni frame di pagina in memoria che indica quale pagina, se esiste, occupa il page frame; la tabella viene così denominata perché l'informazione in essa è l'"inverso" dell'informazione presente nella tabella delle pagine. La dimensione di una tabella delle pagine invertita dipende dalla dimensione della memoria, così è indipendente dal numero e dalle dimensioni dei processi. Comunque, non si può accedere direttamente all'informazione relativa a una pagina come in una tabella delle pagine; essa deve essere ricercata nella IPT.
- *Tabella delle pagine multilivello*: la tabella delle pagine di un processo è essa stessa paginata; non è, perciò, necessario che l'intera tabella delle pagine sia presente sempre in memoria. Viene utilizzata una tabella delle pagine di alto livello per accedere alle pagine della tabella delle pagine. Se la tabella delle pagine di alto livello è grande, potrebbe essa stessa essere paginata e così via. In questa organizzazione, l'accesso alla

entry della tabella delle pagine di una pagina deve essere effettuato attraverso le entry delle tabelle delle pagine di alto livello.

In entrambi gli approcci, il TLB viene usato per ridurre il numero di riferimenti di memoria necessari per eseguire la traduzione degli indirizzi.

### Tabella delle pagine invertita

La [Figura 12.10\(a\)](#) illustra la traduzione degli indirizzi usando una *tabella delle pagine invertita* (IPT). Ogni entry nella tabella delle pagine invertita è una coppia ordinata formata da un identificativo di processo (id del processo) e un numero di pagina. Quindi, una coppia  $(R, p_i)$  nella  $f_i$ -esima entry indica che il frame  $f_i$  è occupato dalla pagina  $p_i$  di un processo  $R$ . Nello scheduling di un processo, lo scheduler copia l'id del processo dal suo PCB in un registro della MMU. Sia  $P$  questo id. La MMU esegue la traduzione degli indirizzi per un indirizzo logico  $(p_i, b_i)$  in un processo  $P$ , usando i seguenti passi.



**Figura 12.10** Tabella delle pagine Invertita: (a) concetto; (b) implementazione usando una hash table.

- Separare le componenti  $p_i$  e  $b_i$  dell'indirizzo logico.
- Formare la coppia  $(P, p_i)$  con l'id del processo  $P$ .
- Cercare la coppia  $(P, p_i)$  nella IPT. Generare un page fault se la coppia non esiste nella IPT.
- Se la coppia  $(P, p_i)$  esiste nella entry  $f_i$  della IPT, copiare il numero di page frame  $f_i$  per utilizzarlo nella traduzione degli indirizzi.
- Calcolare l'indirizzo effettivo di memoria usando  $f_i$  e  $b_i$ .

Questi passi sono rappresentati con i numeri da 1 a 5 nei cerchietti in [Figura 12.10\(a\)](#).

La ricerca di  $(P, p_i)$  nel Passo 3 deve essere condotta in maniera efficiente, altrimenti viene rallentata la traduzione degli indirizzi. Di conseguenza, viene usata una *hash table* per velocizzare la ricerca nella tabella delle pagine invertita. La [Figura 12.10\(b\)](#) mostra una soluzione chiamata *hash-with-chaining*, che funziona in questo modo: ogni entry della tabella delle pagine invertita contiene un ulteriore campo *puntatore*, che punta a un'altra entry nella stessa tabella. Per interpretare una coppia  $(P, p_i)$ , prima

concateniamo le stringhe di bit che rappresentano  $P$  e  $p_i$  per ottenere una stringa di bit più lunga. Poi interpretiamo questa stringa di bit come un numero intero  $x$ , e applichiamo a esso la seguente funzione di hash  $h$ :

$$h(x) = \text{resto} \left( \frac{x}{a} \right)$$

dove  $a$  è la dimensione della hash table, che è tipicamente un numero primo.  $h(x)$ , che è nell'intervallo  $0, \dots, a - 1$ , è un numero di una entry nella hash table. Sia  $v$  il suo valore. L'hashing di molte coppie del tipo (id di processo, id di pagina) può produrre lo stesso valore  $v$ , perché il numero totale di pagine di tutti i processi in memoria è molto più grande della dimensione della hash table. Le entry di tutte queste coppie nella tabella delle pagine invertita sono collegate dal campo *puntatore*.

La tabella delle pagine invertita è costruita e manutenuta dal gestore della memoria virtuale in questo modo: quando la pagina  $p_i$  del processo  $P$  è caricata nel page frame  $f_i$  in memoria, il gestore della memoria virtuale memorizza la coppia  $(P, p_i)$  nella entry  $f_i$ -esima della tabella delle pagine invertita. Poi interpreta questa coppia per ottenere un numero di entry, diciamo  $v$ , e aggiunge la entry  $f_i$ -esima della tabella delle pagine invertita nella catena cominciando dalla entry  $v$ -esima della hash table in questo modo: copia il valore trovato nella entry  $v$ -esima della hash table nel campo *puntatore* della entry  $f_i$ -esima della tabella delle pagine invertita, e inserisce  $f_i$  nella entry  $v$ -esima della hash table. Quando questa pagina è rimossa dalla memoria, il gestore della memoria virtuale cancella la sua entry dalla catena che comincia dalla entry  $v$ -esima della hash table. In [Figura 12.10\(b\)](#), le pagine  $(R, p_l)$ ,  $(P, p_i)$  e  $(Q, p_k)$  sono caricate nei frame  $f_l$ ,  $f_i$  e  $f_k$  rispettivamente, e sono tutte codificate nella entry  $v$ -esima della hash table. L'Esempio 12.2 descrive come la MMU usa la tabella delle pagine invertita durante la traduzione degli indirizzi.

### **Esempio 12.2 Ricerca nella tabella delle pagine invertita**

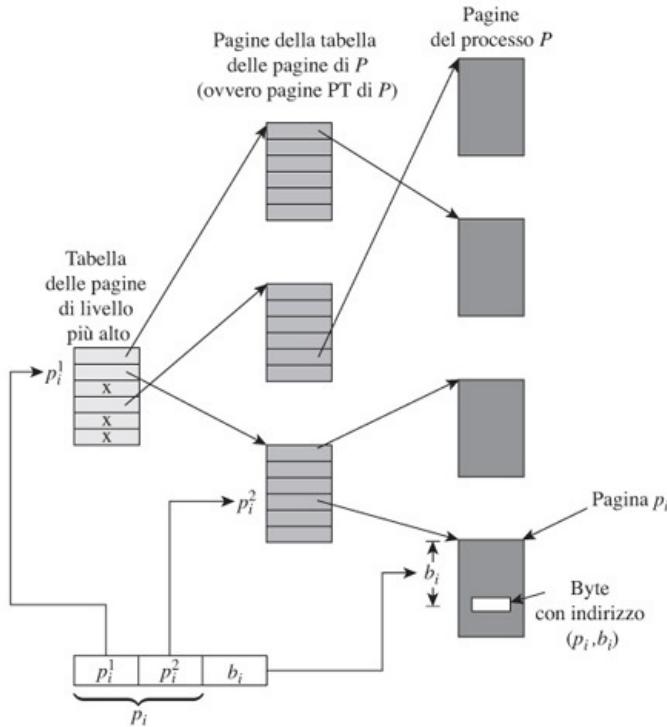
L'indirizzo logico  $(p_i, b_i)$  deve essere tradotto usando la tabella delle pagine invertita di [Figura 12.10\(b\)](#). La coppia  $(P, p_i)$  è interpretata per ottenere un numero di entry  $v$  nella hash table. Viene cercata la catena che comincia con questa entry. La coppia  $(P, p_i)$  non corrisponde alla coppia  $(Q, p_k)$  trovata nel campo *id di pagina* della prima entry della catena. Perciò, la MMU usa il campo *puntatore* di questa entry per localizzare la prossima entry nella catena. La coppia in questa entry corrisponde a  $(P, p_i)$ , quindi la MMU usa il numero di questa entry, ossia,  $f_i$ , come numero di frame per formare l'indirizzo fisico  $(f_i, b_i)$ .

Il numero medio di confronti richiesto per individuare la entry di una coppia  $(P, p_i)$  nella tabella delle pagine invertita dipende dalla lunghezza media della catena che inizia da una entry della hash table. Aumentando la dimensione della hash table,  $a$ , si riduce la lunghezza media della catena. Un valore di  $a > 2 \times \#frame$  assicura che il numero medio delle entry in una lista linkata è minore di 2. La tabella delle pagine invertita contiene esattamente  $\#frame$  entry. Notare che la tabella delle pagine invertita non contiene alcuna informazione relativamente alle pagine che non sono presenti in memoria; una tabella delle pagine convenzionale dovrebbe essere mantenuta su disco per contenere tali informazioni. Le tabelle delle pagine invertite sono state usate nei sistemi IBM RS 6000 e AS 400, nei PowerPC e nelle architetture PA-RISC. Sono state usate anche nei SO Solaris per architetture Sparc.

### **Tabella delle pagine multilivello**

La richiesta di memoria per la tabella delle pagine di un processo è ridotta dalla paginazione della stessa tabella delle pagine e dal caricamento delle pagine su richiesta proprio come per le pagine dei processi. Questo approccio richiede una soluzione di indirizzamento a due livelli in cui una tabella delle pagine di alto livello contiene le entry che riportano le informazioni riguardanti le pagine della tabella delle pagine mentre la tabella delle pagine contiene le informazioni riguardanti le pagine del processo. Le informazioni in ognuna di queste tabelle sono simili a quelle contenute in una tabella delle pagine convenzionale. La [Figura 12.11](#) illustra il concetto di una tabella delle pagine a due livelli. La memoria ora contiene due tipi di pagine, cioè pagine dei processi e pagine delle tabelle delle pagine dei processi, che noi chiameremo *PT page*. Solo tre PT

page di un processo  $P$  sono attualmente in memoria. Per la traduzione degli indirizzi di un indirizzo logico  $(p_i, b_i)$  nel processo  $P$ , dovrebbero essere presenti in memoria sia la pagina  $p_i$  del processo  $P$  che la PT page che contiene la entry relativa alla pagina  $p_i$ .

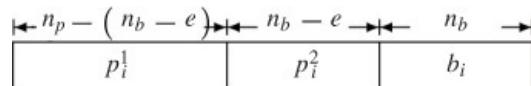


**Figura 12.11** Organizzazione della tabella delle pagine a due-livelli.

Come menzionato nel Paragrafo 12.2, il numero di pagina e il numero di byte in un indirizzo logico  $(p_i, b_i)$  sono rappresentati in  $n_p$  e  $n_b$  bit. La dimensione di ogni entry di una tabella delle pagine è una potenza di 2, quindi il numero delle entry in una tabella delle pagine che è incluso in una PT page è anch'esso una potenza di 2. Se la dimensione di una entry è 2<sup>e</sup> byte, il numero delle entry della tabella delle pagine in una PT page è  $2^{n_b}/2^e$ , ossia  $2^{n_b-e}$ . Perciò, il numero di pagina  $p_i$  nell'indirizzo logico stesso consiste di due parti: l'id della PT page che contiene la entry della tabella delle pagine di  $p_i$ , e un numero di entry all'interno della PT page. Come mostrato in [Figura 12.11](#), chiamiamo queste due parti  $p_i^1$  e  $p_i^2$ , rispettivamente. Dalla discussione precedente,  $p_i^2$  è contenuto negli  $n_b - e$  bit di  $p_i$  meno significativi. Poiché la rappresentazione binaria di  $p_i$  contiene  $n_p$  bit,  $p_i^1$  è contenuto negli  $n_p - (n_b - e)$  bit più significativi.

La [Figura 12.11](#) illustra la traduzione degli indirizzi per un indirizzo logico  $(p_i, b_i)$ . Consiste dei seguenti passi:

1. l'indirizzo  $(p_i, b_i)$  è raggruppato in tre campi:



I contenuti di questi campi sono  $p_i^1$ ,  $p_i^2$  e  $b_i$ , rispettivamente;

2. la PT page con il numero  $p_i^1$  contiene l'entry della tabella delle pagine per  $p_i$ . La MMU controlla se questa pagina è presente in memoria e genera un page fault se non è presente. Il page fault è gestito dal gestore della memoria virtuale per caricare la PT page in memoria;
3.  $p_i^2$  è il numero della entry per  $p_i$  nella PT page. La MMU usa l'informazione in questa

entry per controllare se la pagina  $p_i$  è presente in memoria e genera un page fault se non è presente. Il gestore della memoria virtuale gestisce il page fault e carica la pagina  $p_i$  in memoria;

4. i contenuti della entry della tabella delle pagine di  $p_i$  vengono usati per eseguire la traduzione degli indirizzi.

Quindi, la traduzione degli indirizzi richiede due accessi in memoria, uno per accedere alla tabella delle pagine di più alto livello e l'altro per accedere alla tabella delle pagine del processo  $P$ . Ciò può essere velocizzato attraverso il TLB usando due tipi di entry: quelle della forma  $(P, p_i^1, \text{numero frame, info protezione})$  aiutano a eliminare gli accessi alla tabella delle pagine di più alto livello del processo  $P$  e quelle della forma  $(P, p_i^1, p_i^2, \text{numero frame, info protezione})$  aiutano a eliminare gli accessi alla tabella delle pagine di  $P$ .

Quando la dimensione della tabella delle pagine di alto livello in una organizzazione con tabella delle pagine a due livelli è molto grande, la tabella delle pagine di alto livello può essa stessa essere paginata. Questa soluzione si traduce in una struttura di tabella delle pagine a tre livelli. La traduzione degli indirizzi che usa la tabella delle pagine a tre livelli è eseguita da un'ovvia estensione della traduzione degli indirizzi in tabella delle pagine a due livelli. Un indirizzo logico  $(p_i, b_i)$  è diviso in quattro componenti  $p_i^1, p_i^2, p_i^3$  e  $b_i$  e le prime tre componenti sono utilizzate per indirizzare i tre livelli della tabella delle pagine. Così la traduzione degli indirizzi richiede fino a tre accessi in memoria. In sistemi di elaborazione che usano gli indirizzi a 64 bit, anche la tabella delle pagine di più alto livello in una organizzazione con tabella delle pagine a tre livelli può diventare troppo grande. Vengono utilizzate le tabelle delle pagine a quattro livelli per superare questo problema.

L'architettura Intel 80386 usava una tabella delle pagine a due livelli. Le tabelle delle pagine a tre e quattro livelli sono state utilizzate nelle architetture Sun Sparc e Motorola 68030, rispettivamente.

#### 12.2.4 Operazioni di I/O in ambiente paginato

Un processo per eseguire operazioni di I/O effettua una chiamata di sistema. Due dei suoi parametri sono il numero di byte da trasferire e l'indirizzo logico dell'*area dati*, che è l'area di memoria coinvolta nel trasferimento dati. La chiamata attiva il *gestore di I/O* nel kernel. Il sottosistema I/O non contiene una MMU; usa gli indirizzi fisici per implementare il trasferimento dati per e dalla memoria. Di conseguenza, il gestore di I/O deve eseguire alcune azioni preparatorie prima di iniziare l'operazione di I/O. La prima di queste è sostituire l'indirizzo logico dell'area dati con il suo indirizzo fisico, usando l'informazione dalla tabella delle pagine del processo. Deve eseguire anche altre azioni per risolvere i due problemi che discuteremo di seguito.

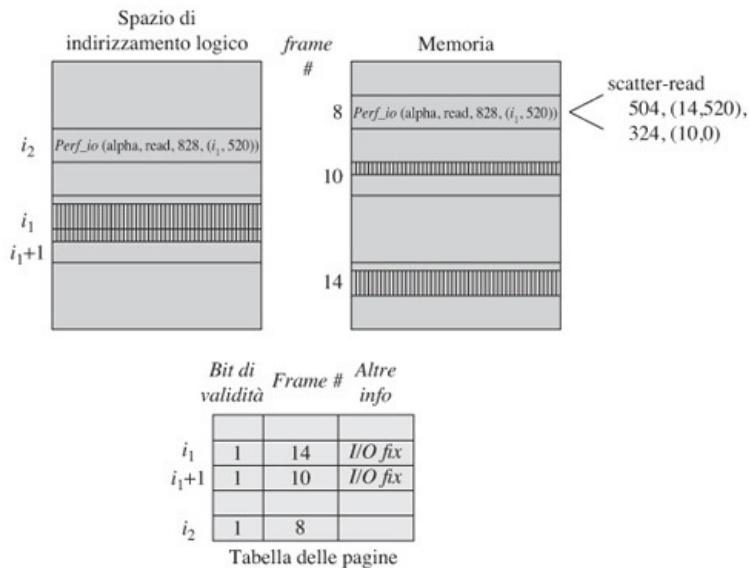
L'area dati in un'operazione di I/O può estendersi su molte pagine del processo. Un page fault durante l'accesso a una pagina dell'area dati interromperebbe bruscamente l'operazione di I/O, quindi tutte queste pagine devono rimanere in memoria durante l'esecuzione dell'I/O. Il gestore I/O soddisfa questo requisito caricando tutte le pagine dell'area dati in memoria e mettendo un *I/O fix* su ogni pagina per avvertire il gestore della memoria virtuale che queste pagine non dovranno essere sostituite finché non sarà rimosso il fix al termine dell'operazione di I/O. Può cominciare ora l'operazione di I/O. Un modo facile per implementare l'operazione di I/O fixing delle pagine è aggiungere un bit relativo all'I/O fix nel campo *Misc info* di ciascuna entry della tabella delle pagine.

Poiché il sottosistema di I/O funziona senza una MMU, ci si aspetta che l'area dei dati occupi un'area contigua di memoria. Tuttavia, il processo è paginato, di conseguenza le pagine dell'area dati possono non avere indirizzi fisici contigui. Questa situazione può essere risolta in due modi. La maggior parte dei sottosistemi di I/O hanno la caratteristica *scatter/gather*, che consente di depositare parti di dati di un'operazione di I/O in aree di memoria non contigue. Per esempio, i primi pochi byte da un record di I/O possono essere letti in un frame situato in una parte della memoria e i rimanenti byte possono essere letti in un altro frame situato in una parte diversa della memoria. Analogamente, una "gather write" può prendere i dati dell'operazione di I/O da aree di memoria non contigue e scriverle in un record su un dispositivo di I/O. L'Esempio 12.3 illustra come viene utilizzata un'operazione scatter-read per implementare un'operazione di I/O che si estende su due pagine in un processo. Se un sottosistema di I/O non ha la

caratteristica scatter/gather, il gestore di I/O può risolvere la situazione in due modi: può istruire il gestore della memoria virtuale a mettere le pagine contenenti l'area dati in maniera contigua in memoria, oppure può prima leggere i dati in un'area del kernel che ha indirizzi fisici contigui e poi copiarli nell'area dei dati del processo. Analoghe scelte possono essere fatte per supportare un'operazione di write.

### Esempio 12.3 Operazione di I/O in memoria virtuale

La pagina  $i_2$  di un processo  $P_i$  contiene una chiamata di sistema “*perf\_io* (*alpha*, read, 828, ( $i_1, 520$ ))”, dove *alpha* è un file, 828 è il totale dei byte di dati da leggere, e ( $i_1, 520$ ) è l'indirizzo logico dell'inizio dell'area dati. La Figura 12.12 illustra come è implementata l'operazione di I/O. La dimensione della pagina è 1 KB, e così l'area dei dati è situata nelle pagine  $i_1$  e  $i_1 + 1$  del processo. Prima di iniziare l'operazione di I/O, il gestore di I/O invoca il gestore della memoria virtuale per caricare le pagine  $i_1$  e  $i_1 + 1$  in memoria. Vengono caricate nei frame 14 e 10 della memoria. Il gestore di I/O mette un I/O fix su queste pagine settando i bit relativi all'I/O fix nel campo *Misc info* delle loro entry nella tabella delle pagine. Queste pagine non sono sostituite finché l'I/O fix non verrà rimosso al termine dell'operazione di I/O. Il gestore di I/O genera ora un'operazione scatter-read per leggere i primi 504 byte cominciando al byte numero 520 nel frame 14, e i rimanenti 324 byte cominciando al byte numero 0 nel frame 10. Rimuove poi l'I/O fix sulle pagine 14 e 10 quando l'operazione di I/O termina.



**Figura 12.12** Un'operazione di I/O in memoria virtuale.

## 12.3 Il gestore della memoria virtuale

Il gestore della memoria virtuale usa due strutture dati: la tabella delle pagine, il cui formato della entry è mostrato in Figura 12.3, e la lista dei frame liberi. I campi *Ref info* e *Modificato* nella entry della tabella delle pagine sono tipicamente determinati dal supporto hardware alla paginazione. Tutti gli altri campi sono determinati dallo stesso gestore della memoria virtuale. La Tabella 12.4 schematizza le funzioni del gestore della memoria virtuale. Discutiamo le prime quattro funzioni in questo paragrafo. Altre funzioni, quali sostituzione di pagine, allocazione di memoria per i processi e implementazione della condivisione di pagine, saranno discusse nei prossimi paragrafi.

| Funzione   | Descrizione  |
|--|--|
| Gestione spazio di indirizzamento logico             | Set up dello spazio di swap di un processo. Organizzare il suo spazio di indirizzamento logico in memoria attraverso le operazioni di page-in e page-out, e gestire la sua tabella delle pagine.                           |
| Gestione della memoria                               | Tenere traccia dei frame occupati e liberi in memoria.   |
| Implementare la protezione di memoria                | Conservare le informazioni utili per la protezione di memoria.   |
| Raccogliere informazioni sui riferimenti alle pagine | Il supporto hardware alla paginazione fornisce le informazioni circa i riferimenti alle pagine. Queste informazioni sono tenute in appropriate strutture dati per l'uso da parte dell'algoritmo di sostituzione di pagine. |
| Eseguire sostituzione di pagine                      | Eseguire la sostituzione di una pagina quando viene generato un fault e tutti i page frame in memoria, o tutti i frame allocati per un processo, sono occupati.  |
| Allocare memoria fisica                              | Decidere quanta memoria allocare a un processo e rivedere questa decisione di volta in volta per adattarsi alle necessità del processo e del SO.   |
| Implementare la condivisione delle pagine            | Organizzare la condivisione delle pagine da elaborare.   |

**Tabella 12.4** Funzioni del gestore della memoria virtuale.

#### **Gestione dello spazio di indirizzamento logico di un processo**

Il gestore della memoria virtuale gestisce lo spazio di indirizzamento logico di un processo attraverso le seguenti sottofunzioni.

1. Creare una copia delle istruzioni e dei dati del processo nel suo spazio di swap.
2. Gestire la tabella delle pagine.
3. Eseguire le operazioni di page-in e page-out.
4. Eseguire l'inizializzazione del processo.

Come menzionato precedentemente nel Paragrafo 12.2, una copia dell'intero spazio di indirizzamento logico di un processo è tenuta nello spazio di swap del processo. Quando un riferimento a una pagina determina un page fault, la pagina viene caricata dallo spazio di swap mediante un'operazione di page-in. Quando una pagina dirty deve essere rimossa dalla memoria, viene eseguita un'operazione di page-out per copiarla dalla memoria in un blocco di disco nello spazio di swap. In tal modo la copia di una pagina nello spazio di swap è attuale se quella pagina non è in memoria, o è in memoria ma non è stata modificata da quando è stata caricata. Per le altre pagine, la copia nello spazio di swap è obsoleta, mentre in memoria è valida.

Un problema nella gestione dello spazio di swap è la sua dimensione per un processo. La maggior parte delle implementazioni della memoria virtuale consentono allo spazio di indirizzamento logico di un processo di crescere dinamicamente durante la sua esecuzione. Ciò può accadere per svariate ragioni. La dimensione dello stack o dell'area dati PCD può crescere (Paragrafo 11.4.2), oppure il processo può collegare dinamicamente più moduli o può eseguire il mapping in memoria di un file (Paragrafo 12.7). Un modo per gestire la crescita dinamica degli spazi di indirizzamento è quello di allocare spazio di swap dinamicamente e in maniera non contigua. Tuttavia, questo approccio va incontro al problema che il gestore della memoria virtuale può andare in esecuzione al di fuori dello spazio di swap durante l'esecuzione di un processo.

Per avviare un processo, i gestori hanno bisogno solo che sia caricata in memoria la pagina contenente il suo *indirizzo di avvio* (start address), ovvero l'indirizzo della prima istruzione. Le altre pagine sono caricate su richiesta. I dettagli relativi alla tabella delle pagine e alle operazioni di page-in e page-out sono stati descritti precedentemente nel Paragrafo 12.2.

#### **Gestione della memoria**

La lista dei frame liberi è aggiornata costantemente. Un frame viene eliminato dalla lista di quelli liberi per caricare una nuova pagina; un frame viene, invece, aggiunto quando viene eseguita un'operazione di page-out. Quando un processo termina, tutti i page frame allocati per il processo sono aggiunti alla lista dei frame liberi.

### Protezione

Durante la creazione di un processo, il gestore della memoria costruisce la sua tabella delle pagine e memorizza le informazioni relative allo start address della tabella delle pagine e alla sua dimensione nel PCB del processo. Il gestore della memoria virtuale registra i privilegi di accesso del processo per una pagina nel campo *Prot info* della sua entry nella tabella delle pagine. Durante l'inizializzazione del processo, il kernel carica lo start address della tabella delle pagine del processo e la sua dimensione della tabella delle pagine nei registri della MMU. Durante la traduzione di un indirizzo logico  $(p_i, b_i)$ , la MMU assicura che la entry della pagina  $p_i$  esiste nella tabella delle pagine e contiene gli appropriati privilegi di accesso nel campo *Prot info*.

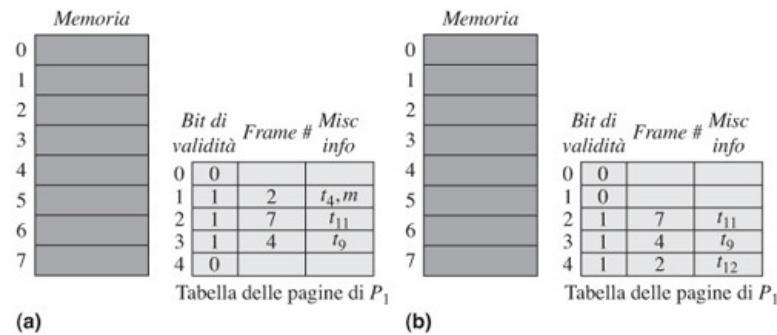
### Raccolta di informazioni per la sostituzione di pagina

Il campo *Ref info* della entry della tabella delle pagine di una pagina indica l'ultimo riferimento alla pagina, il campo *Modificato* indica se è stata modificata dall'ultima volta che è stata in memoria. L'informazione di riferimento della pagina è utile solo finché la pagina rimane in memoria; è reinizializzata quando viene eseguita un'operazione di page-in per la pagina. La maggior parte dei computer prevedono un solo bit nel campo *Ref info* per raccogliere informazioni di riferimento di pagina. Questa informazione non è adeguata per selezionare il miglior candidato per la sostituzione di pagina. Di conseguenza, il gestore della memoria virtuale può periodicamente resettare il bit usato per memorizzare questa informazione. Discutiamo questo aspetto nel Paragrafo 12.4.1.

#### Esempio 12.4 Sostituzione di pagina

La memoria di un computer è composta da otto frame. Sia  $P_1$  un processo di cinque pagine numerate da 0 a 4. Solo le pagine 1, 2 e 3 sono in memoria attualmente; occupano i page frame 2, 7 e 4, rispettivamente. I rimanenti frame sono stati allocati ad altri processi e nel sistema non ci sono frame liberi.

La Figura 12.13(a) illustra la situazione del sistema all'istante di tempo  $t_{11}^+$ , ossia un po' dopo  $t_{11}$ . In figura è mostrata solo la tabella delle pagine di  $P_1$  poiché è stato schedulato il processo  $P_1$ . Nel campo *Misc info* sono riportati i contenuti dei campi *Ref info* e *Modified*. Le pagine 1, 2 e 3 sono state referenziate per ultimo negli istanti di tempo, rispettivamente,  $t_4$ ,  $t_{11}$  e  $t_9$ . La pagina 1 è stata modificata saltuariamente dopo essere stata caricata per ultima. Di conseguenza, il campo *Misc info* della sua entry nella tabella delle pagine contiene l'informazione  $t_4, m$ .



**Figura 12.13** Strutture dati del gestore della memoria virtuale: (a) prima e (b) dopo una sostituzione di pagina.

All'istante di tempo  $t_{12}$ , il processo  $P_1$  dà luogo a un page fault per la pagina 4. Poiché sono occupati tutti i page frame in memoria, il gestore della memoria virtuale decide di sostituire la pagina 1 del processo. Il simbolo *m* nel campo *Misc info* della entry nella tabella delle pagine di pagina 1 indica che è stata modificata da quando è stata caricata per ultima, quindi è necessaria un'operazione di page-out. Il campo

Frame # della entry nella tabella delle pagine di pagina 1 indica che la pagina esiste nel page frame 2. Il gestore della memoria virtuale esegue un'operazione di page-out per scrivere i contenuti del frame 2 nell'area di swap riservata alla pagina 1 di  $P_1$ , e modifica il bit di *validità* della entry nella tabella delle pagine di pagina 1 per indicare che non è presente in memoria. Un'operazione di page-in è ora iniziata per la pagina 4 di  $P_1$ . Al termine dell'operazione, viene modificata la entry nella tabella delle pagine di pagina 4 per indicare che è in memoria nel frame 2. Viene ripristinata l'esecuzione di  $P_1$ . Viene ora effettuato un riferimento alla pagina 4, e quindi l'informazione del riferimento della pagina 4 indica che è stata riferita per ultimo all'istante  $t_{12}$ . La Figura 12.13(b) indica la tabella delle pagine di  $P_1$  all'istante di tempo  $t_{12}^+$ .

### 12.3.1 Panoramica sul funzionamento del gestore della memoria virtuale

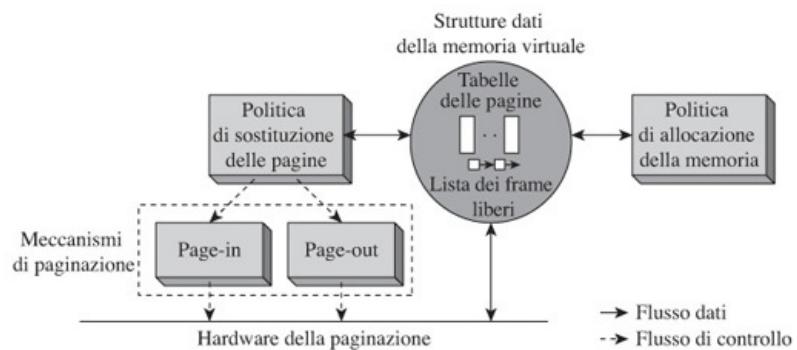
Il gestore della memoria virtuale prende due importanti decisioni durante il suo funzionamento:

- quando si verifica un page fault durante l'esecuzione di qualche processo  $proc_i$ , decide quale pagina deve essere sostituita;
- periodicamente decide quanta memoria, ossia quanti frame devono essere allocati a ciascun processo.

Come discusso nei precedenti paragrafi, queste decisioni vengono prese indipendentemente l'una dall'altra. Quando si verifica un page fault, il gestore della memoria virtuale sostituisce semplicemente una pagina dello stesso processo se sono occupati tutti i page frame allocati al processo. Quando decide di aumentare o diminuire la memoria allocata a un processo, specifica semplicemente il nuovo numero di frame che dovrebbero essere allocati a ciascun processo.

La Figura 12.14 raffigura l'organizzazione della politica e i moduli di funzionamento del gestore della memoria virtuale. La politica di sostituzione delle pagine utilizza l'informazione relativa al riferimento di pagina disponibile nelle strutture dati del gestore della memoria virtuale, e aggiorna le tabelle delle pagine affinché rispecchino le sue decisioni. L'implementazione utilizza operazioni di page-in e page-out come meccanismi. I meccanismi di page-in e page-out interagiscono con il supporto hardware alla paginazione per implementare le loro funzionalità. L'hardware della paginazione aggiorna le informazioni di riferimento di pagina mantenute nelle tabelle del gestore della memoria virtuale. La politica di allocazione della memoria usa le informazioni nelle tabelle delle pagine e nella lista dei frame liberi per decidere periodicamente se e come variare la memoria allocata a ogni processo. Usiamo la seguente notazione per la memoria allocata a ogni processo:

$alloc_i$  Numero di page frame allocati al processo  $proc_i$



**Figura 12.14** I moduli del gestore della memoria virtuale.

Omettiamo il pedice di  $alloc_i$  quando è preso in considerazione un solo processo.

## 12.4 Politiche di sostituzione delle pagine

Come discusso precedentemente nel Paragrafo 12.2.1, una politica di sostituzione delle pagine dovrebbe sostituire solo quelle pagine che probabilmente non saranno riferite nell'immediato. Valutiamo le seguenti tre politiche di sostituzione delle pagine per vedere come si comportano rispetto all'obiettivo proposto.

- Politica di sostituzione delle pagine ottimale.
- Politica di sostituzione delle pagine First-In, First-Out (FIFO).
- Politica di sostituzione delle pagine Least Recently Used (LRU).

Per la nostra analisi di queste politiche di sostituzione di pagina, ci basiamo sul concetto di *stringa dei riferimenti alle pagine*. Una stringa dei riferimenti alle pagine di un processo è una sequenza di pagine a cui un processo ha fatto accesso durante la sua esecuzione. Può essere costruita monitorando l'esecuzione di un processo, e formando una sequenza di numeri di pagina che appaiono negli indirizzi logici generati da esso. La stringa dei riferimenti alle pagine di un processo dipende dai dati di input, quindi l'uso di dati differenti determina stringhe diverse per un processo.

Per convenienza, associamo una *stringa dei tempi dei riferimenti*  $t_1, t_2, t_3, \dots$  a ogni stringa dei riferimenti alle pagine. In questo modo, assumiamo che il  $k$ -esimo riferimento di pagina nella stringa dei riferimenti alle pagine dovrà verificarsi all'istante di tempo  $t_k$ . (In effetti, supponiamo che un *orologio logico* sia attivo solo quando un processo è in stato *running* e avanzi solo quando il processo fa riferimento a un indirizzo logico.) L'Esempio 12.5 illustra la stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti associata a un processo.

### Esempio 12.5 Stringa dei riferimenti alle pagine

Un computer supporta istruzioni di lunghezza 4 byte, e usa una dimensione di pagina di 1 KB. Esegue il seguente programma, privo di senso, in cui i simboli *A* e *B* sono nelle pagine 2 e 5, rispettivamente:

|      |       |          |
|------|-------|----------|
|      | START | 2040     |
|      | READ  | B        |
| LOOP | MOVER | AREG, A  |
|      | SUB   | AREG, B  |
|      | BC    | LT, LOOP |
|      | ...   |          |
|      | STOP  |          |
| A    | DS    | 2500     |
| B    | DS    | 1        |
|      | END   |          |

La stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti per il processo sono le seguenti:

stringa dei riferimenti alle pagine 1, 5, 1, 2, 2, 5, 2, 1, ...  
stringa dei tempi dei riferimenti  $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, \dots$

L'indirizzo logico della prima istruzione è 2040, e quindi si trova nella pagina 1. Il primo riferimento di pagina nella stringa, perciò, è 1. Siamo all'istante di tempo  $t_1$ . *B*, l'operando dell'istruzione, è situato nella pagina 5, e quindi il secondo riferimento di pagina nella stringa è 5, all'istante  $t_2$ . La prossima istruzione è situata nella pagina 1 e si riferisce ad *A*, che si trova nella pagina 2, e quindi i due successivi riferimenti di pagina sono alle pagine 1 e 2. Le successive due istruzioni sono situate alla pagina 2, e l'istruzione con la label *LOOP* si trova nella pagina 1. Perciò, se il valore dell'input *B* all'istruzione *READ* è maggiore del valore *A*, i successivi quattro riferimenti di pagina sarebbero alle pagine 2, 5, 2 e 1, rispettivamente; altrimenti, i successivi quattro riferimenti di pagina sarebbero alle pagine 2, 5, 2 e 2, rispettivamente.

### Sostituzione ottimale delle pagine

La sostituzione *ottimale* consiste nel prendere decisioni relativamente alla sostituzione di

pagine in modo tale che il numero totale di page fault durante l'esecuzione di un processo sia il minimo possibile; ossia, nessun'altra sequenza di decisioni relative alla sostituzione di pagine può determinare un numero inferiore di page fault. Per raggiungere la sostituzione di pagine ottimale, a ogni page fault, la politica di sostituzione di pagine deve considerare tutte le decisioni alternative di sostituzione di pagine, analizzare le loro implicazioni per i page fault successivi, e selezionare la migliore alternativa. Naturalmente, una politica del genere non è praticabile in realtà: il gestore della memoria virtuale non può conoscere il comportamento futuro di un processo. Come strumento analitico, comunque, questa politica fornisce un utile confronto, con il senso di poi, relativamente alle prestazioni delle altre politiche di sostituzione di pagina (Esempio 12.6 e Problema 12.5).

Sebbene la sostituzione ottimale delle pagine sembri richiedere un'eccessiva analisi, Belady (1966) mostrò che è equivalente alla seguente semplice regola: al verificarsi di un page fault, sostituire la pagina il cui prossimo riferimento è il più lontano nella stringa dei riferimenti di pagina.

### **Sostituzione delle pagine FIFO**

Ad ogni page fault, la politica di sostituzione delle pagine FIFO sostituisce la pagina che è stata caricata in memoria prima di ogni altra pagina del processo. Per facilitare la sostituzione delle pagine FIFO, il gestore della memoria virtuale registra il tempo di caricamento di una pagina nel campo *Ref info* della sua entry nella tabella delle pagine. Quando si verifica un page fault, questa informazione è usata per determinare  $p_{earliest}$ , la pagina che era stata caricata prima di ogni altra pagina del processo. Questa è la pagina che sarà sostituita con la pagina il cui riferimento ha determinato il page fault.

### **Sostituzione di pagine LRU**

La politica LRU usa il principio della località dei riferimenti come base delle sue decisioni per la sostituzione. Il suo funzionamento può essere descritto come segue: a ogni page fault la pagina *utilizzata meno di recente* (least recently used – LRU) è sostituita dalla pagina richiesta. La entry nella tabella delle pagine di una pagina memorizza l'istante in cui la pagina era stata riferita per ultimo. Questa informazione è inizializzata quando una pagina è caricata, ed è aggiornata ogni volta che la pagina è riferita. Quando si verifica un page fault, questa informazione è utilizzata per localizzare la pagina  $p_{LRU}$  alla quale è stato fatto l'ultimo riferimento in un tempo precedente rispetto a quelli di tutte le altre pagine. Questa pagina è sostituita con la pagina il cui riferimento ha determinato il page fault.

### **Analisi delle politiche di sostituzione di pagine**

L'Esempio 12.6 illustra il funzionamento delle politiche di sostituzione di pagine ottimale, FIFO e LRU.

#### **Esempio 12.6 Funzionamento delle politiche di sostituzione di pagine**

Si considerino la seguente stringa dei riferimenti alle pagine e la stringa dei tempi dei riferimenti per un processo  $P$ :

$$\text{stringa dei riferimenti alle pagine} \quad 0, 1, 0, 2, 0, 1, 2, \dots \quad (12.4)$$

$$\text{stringa dei tempi dei riferimenti} \quad t_1, t_2, t_3, t_4, t_5, t_6, t_7, \dots \quad (12.5)$$

La [Figura 12.15](#) illustra il funzionamento delle politiche di sostituzione delle pagine ottimale, FIFO e LRU per questa stringa dei riferimenti alle pagine con  $alloc = 2$ . Per convenienza, mostriamo solo due campi della tabella delle pagine, il *bit di validità* e *ref info*. Nell'intervallo tra  $t_0$  e  $t_3$  (incluso), vengono riferite solo due pagine distinte: le pagine 0 e 1. Possono stare entrambe in memoria nello stesso istante perché  $alloc = 2$ .  $t_4$  è il primo istante di tempo in cui un page fault determina una sostituzione di pagina.

| Istante | Rif.<br>pagina | Ottimale                |                      | FIFO                          |                      | LRU                           |                      |
|---------|----------------|-------------------------|----------------------|-------------------------------|----------------------|-------------------------------|----------------------|
|         |                | Bit di<br>validità info | Sostitu-<br>zione    | Bit di<br>validità info       | Sostitu-<br>zione    | Bit di<br>validità info       | Sostitu-<br>zione    |
| $t_1$   | 0              | 0 1<br>1 0<br>2 0       | -                    | 0 1 $t_1$<br>1 0<br>2 0       | -                    | 0 1 $t_1$<br>1 0<br>2 0       | -                    |
| $t_2$   | 1              | 0 1<br>1 1<br>2 0       | -                    | 0 1 $t_1$<br>1 1 $t_2$<br>2 0 | -                    | 0 1 $t_1$<br>1 1 $t_2$<br>2 0 | -                    |
| $t_3$   | 0              | 0 1<br>1 1<br>2 0       | -                    | 0 1 $t_1$<br>1 1 $t_2$<br>2 0 | -                    | 0 1 $t_1$<br>1 1 $t_2$<br>2 0 | -                    |
| $t_4$   | 2              | 0 1<br>1 0<br>2 1       | Sostuisci<br>1 con 2 | 0 0<br>1 1 $t_2$<br>2 1 $t_4$ | Sostuisci<br>0 con 2 | 0 1 $t_3$<br>1 0<br>2 1 $t_4$ | Sostuisci<br>1 con 2 |
| $t_5$   | 0              | 0 1<br>1 0<br>2 1       | -                    | 0 1 $t_5$<br>1 0<br>2 1 $t_4$ | Sostuisci<br>1 con 0 | 0 1 $t_5$<br>1 0<br>2 1 $t_4$ | -                    |
| $t_6$   | 1              | 0 0<br>1 1<br>2 1       | Sostuisci<br>0 con 1 | 0 1 $t_5$<br>1 1 $t_6$<br>2 0 | Sostuisci<br>2 con 1 | 0 1 $t_5$<br>1 1 $t_6$<br>2 0 | Sostuisci<br>2 con 1 |
| $t_7$   | 2              | 0 0<br>1 1<br>2 1       | -                    | 0 0<br>1 1 $t_6$<br>2 1 $t_7$ | Sostuisci<br>0 con 2 | 0 0<br>1 1 $t_6$<br>2 1 $t_7$ | Sostuisci<br>0 con 2 |

**Figura 12.15** Confronto delle politiche di sostituzione di pagina con  $alloc = 2$ .

La colonna di sinistra mostra i risultati per la sostituzione di pagina ottimale. L'informazione sui riferimenti di pagina non è mostrata nella tabella delle pagine poiché l'informazione riguardante i riferimenti passati non serve per la sostituzione ottimale delle pagine. Quando si verifica un page fault all'istante di tempo  $t_4$ , la pagina 1 è sostituita perché il suo riferimento successivo è più lontano nella stringa dei riferimenti rispetto a quello di pagina 0. All'istante  $t_6$  la pagina 1 sostituisce la pagina 0 perché il riferimento successivo della pagina 0 è più lontano rispetto a quello di pagina 2.

La colonna di centro della [Figura 12.15](#) mostra i risultati per la politica di sostituzione FIFO. Quando si verifica un page fault all'istante di tempo  $t_4$ , il campo *ref info* mostra che la pagina 0 era stata caricata prima della pagina 1, e così la pagina 0 è sostituita dalla pagina 2.

L'ultima colonna della [Figura 12.15](#) mostra i risultati per la politica di sostituzione LRU. Il campo *Ref info* della tabella delle pagine indica quando una pagina era stata riferita l'ultima volta. All'istante di tempo  $t_4$ , la pagina 1 è sostituita dalla pagina 2 perché l'ultimo riferimento della pagina 1 è precedente rispetto all'ultimo riferimento della pagina 0.

Il numero totale di page fault che si verificano con le politiche ottimale, FIFO e LRU sono 4, 6 e 5, rispettivamente. Per definizione, nessun'altra politica ha meno page fault rispetto alla politica di sostituzione ottimale delle pagine.

Quando ci domandiamo perché la politica LRU funzioni meglio della politica FIFO nell'Esempio 12.6, osserviamo che la politica FIFO rimuoveva la pagina 0 all'istante  $t_4$  mentre LRU non faceva altrettanto perché la pagina 0 era stata riferita successivamente rispetto alla pagina 1. Questa decisione rispetta il principio della località dei riferimenti, che stabilisce che, poiché la pagina 0 era stata riferita molto più recentemente rispetto alla pagina 1, ha maggiore probabilità di essere riferita di nuovo rispetto alla pagina 1.

La politica LRU mostra prestazioni migliori poiché la pagina 0 è stata difatti riferita prima rispetto alla pagina 1, successivamente all'istante  $t_4$ . Tuttavia, la politica di sostituzione di pagina LRU non può avere prestazioni migliori rispetto alla politica FIFO se i riferimenti in una stringa dei riferimenti non rispettano il principio della località. Per esempio, per  $alloc = 3$ , le politiche LRU e FIFO dovrebbero avere prestazioni identiche per la stringa dei riferimenti 0, 1, 2, 3, 0, 1, 2, 3, mentre la politica LRU dovrebbe avere prestazioni peggiori rispetto alla politica FIFO per la stringa 0, 1, 2, 0, 3, 1. Comunque, tali situazioni non accadono di frequente.

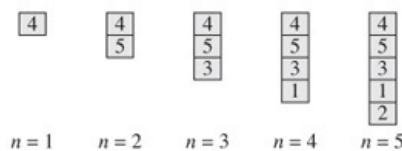
Per raggiungere la situazione ottimale presentata in [Figura 12.6](#) relativamente ai page fault, una politica di sostituzione di pagina deve possedere la *stack property* (anche detta *proprietà di inclusione*). È definita usando la seguente notazione:

$\{p_i\}_n^k$  insieme di pagine che esistono in memoria all'istante di tempo  $t_k^+$  se  $alloc_i = n$  per tutta l'esecuzione del processo  $proc_i$  ( $t_k^+$  è un istante successivo a  $t_k$  e precedente a  $t_{k+1}$ ).

**Definizione 12.3** Una politica di sostituzione di pagina possiede la *proprietà dello stack* se

$$\{p_i\}_n^k \subseteq \{p_i\}_m^k \text{ per tutti } n, m \text{ tali che } n < m.$$

La [Figura 12.16](#) illustra  $\{p_i\}_n^k$  per differenti valori di  $n$  per una politica di sostituzione di pagina. Troviamo che  $\{p_i\}_n^k \subseteq \{p_i\}_{n+1}^k$  per  $n = 1, \dots, 4$ . Di conseguenza l'algoritmo possiede la proprietà dello stack.



**Figura 12.16**  $\{p_i\}_n^k$  per differenti  $n$  per una politica di sostituzione di pagina che implementa la proprietà dello stack.

Per capire come la proprietà dello stack assicuri il minor numero di page fault di [Figura 12.6](#), consideriamo due esecuzioni del processo  $proc_i$ , una con  $alloc_i = n$  per tutta l'esecuzione, e un'altra con  $alloc_i = m$ , tale che  $n < m$ . Se la politica di sostituzione delle pagine possiede la proprietà dello stack, allora in punti identici durante queste esecuzioni del  $proc_i$  (ossia, a istanti identici) tutte le pagine che erano in memoria quando  $alloc_i = n$  dovrebbero essere in memoria quando  $alloc_i = m$ . Inoltre, la memoria contiene anche altre  $m - n$  pagine del processo. Se qualcuna di queste pagine viene riferita nei primi riferimenti di pagina successivi del  $proc_i$ , si verificano page fault se  $alloc_i = n$ , ma non se  $alloc_i = m$ . Quindi, la percentuale di page fault è maggiore se  $alloc_i = n$  rispetto ad  $alloc_i = m$ . Questo soddisfa la caratteristica dei page fault di [Figura 12.6](#). La percentuale di page fault sarà identica se queste  $m - n$  pagine non sono riferite nei riferimenti di pagina immediatamente successivi. Tuttavia, in nessun caso la percentuale di page fault si incrementerà quando l'allocazione della memoria per un processo viene incrementata. Se una politica di sostituzione di pagina non rispetta la proprietà dello stack, allora  $\{p_i\}_m^k$  può non contenere alcune pagine contenute in  $\{p_i\}_n^k$ . I riferimenti a queste pagine determineranno dei page fault. Di conseguenza, la percentuale di page fault può aumentare quando aumenta l'allocazione della memoria per un processo.

L'Esempio 12.7 illustra che la politica di sostituzione di pagina FIFO non rispetta la proprietà dello stack. Si può provare, invece, che la politica di sostituzione di pagina LRU rispetta la proprietà dello stack (Problema 12.9).

### Esempio 12.7 Problemi nella sostituzione delle pagine FIFO

Consideriamo le seguenti stringhe dei riferimenti e dei tempi dei riferimenti per un processo:

stringa dei riferimenti alle pagine 5, 4, 3, 2, 1, 4, 3, 5, 4, 3, 2, 1, 5, ...  
 stringa dei tempi dei riferimenti  $t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, \dots$

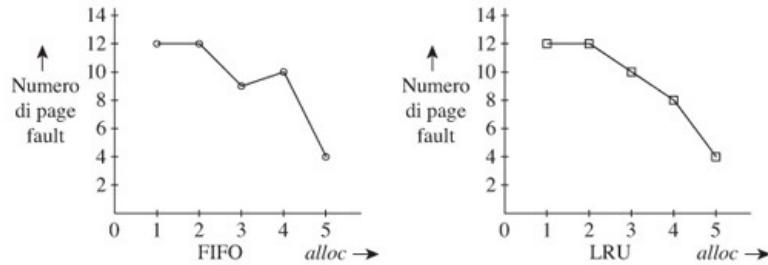
La [Figura 12.17](#) mostra l'esecuzione delle politiche di sostituzione delle pagine FIFO e LRU per questa stringa dei riferimenti alle pagine. I riferimenti di pagina che causano i page fault e determinano le sostituzioni di pagina vengono contrassegnati con il simbolo \*. È associata una colonna di quadratini a ogni istante di tempo. Ogni quadratino è un page frame; il numero in esso contenuto indica quale pagina lo occupa dopo l'esecuzione dei riferimenti di memoria segnati sotto la colonna.

|      |               |   |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
|------|---------------|---|----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|---|---|----|----|----|---|----|----|----|----|---|---|----|----|----|----|----|----|
| FIFO | $alloc_i = 3$ | <table border="1"> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td><td>5</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5</td></tr> </table>  |    |    | 3  | 3  | 3  | 4  | 4  | 4  | 4  | 4  | 2 | 2 | 2 | 5 | 4 | 4 | 4 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 5 | 4* | 3* | 2* | 1* | 4* | 3* | 5* | 4 | 3 | 2* | 1* | 5  |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      |               | 3   | 3  | 3  | 4  | 4  | 4  | 4  | 4  | 2  | 2  | 2  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4             | 4   | 4  | 1  | 1  | 1  | 5  | 5  | 5  | 5  | 5  | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 5             | 5   | 2  | 2  | 2  | 3  | 3  | 3  | 3  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4*            | 3*  | 2* | 1* | 4* | 3* | 5* | 4  | 3  | 2* | 1* | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      | $alloc_i = 4$ | <table border="1"> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td></tr> <tr><td>4</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td>5</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>5*</td></tr> </table> |    |    | 2  | 2  | 2  | 2  | 2  | 2  | 2  | 3  | 3 | 3 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 1 | 1 | 5 | 5  | 5  | 5  | 1  | 1  | 1  | 1  | 1 | 1 | 2  | 2  | 2  | 5 | 4* | 3* | 2* | 1* | 4 | 3 | 5* | 4* | 3* | 2* | 1* | 5* |
|      |               | 2   | 2  | 2  | 2  | 2  | 2  | 2  | 3  | 3  | 3  | 3  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 4    | 3             | 3   | 3  | 3  | 3  | 3  | 3  | 4  | 4  | 4  | 4  | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4             | 4   | 4  | 4  | 4  | 4  | 5  | 5  | 5  | 5  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 5             | 5   | 5  | 1  | 1  | 1  | 1  | 1  | 1  | 2  | 2  | 2  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4*            | 3*  | 2* | 1* | 4  | 3  | 5* | 4* | 3* | 2* | 1* | 5* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      | Tempo         | $t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \ t_8 \ t_9 \ t_{10} \ t_{11} \ t_{12} \ t_{13}$   |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| LRU  | $alloc_i = 3$ | <table border="1"> <tr><td></td><td></td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>5</td><td>4</td><td>4</td><td>4</td><td>1</td><td>1</td><td>1</td><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>2</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>5</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4*</td><td>3*</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </table>   |    |    | 3  | 3  | 3  | 4  | 4  | 4  | 4  | 4  | 1 | 1 | 1 | 5 | 4 | 4 | 4 | 1 | 1 | 1 | 5 | 5 | 5 | 2 | 2 | 2 | 5 | 5 | 5 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 4* | 3* | 2* | 1* | 4* | 3* | 5* | 4 | 3 | 2* | 1* | 5* |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      |               | 3   | 3  | 3  | 4  | 4  | 4  | 4  | 4  | 1  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4             | 4   | 4  | 1  | 1  | 1  | 5  | 5  | 5  | 2  | 2  | 2  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 5             | 5   | 2  | 2  | 2  | 3  | 3  | 3  | 3  | 3  | 3  | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4*            | 3*  | 2* | 1* | 4* | 3* | 5* | 4  | 3  | 2* | 1* | 5* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      | $alloc_i = 4$ | <table border="1"> <tr><td></td><td></td><td>2</td><td>2</td><td>2</td><td>2</td><td>2</td><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td></tr> <tr><td>4</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>3</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td>5</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>5</td></tr> <tr><td>5</td><td>5</td><td>5</td><td>5</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td><td>2</td><td>2</td></tr> <tr><td>5</td><td>4*</td><td>3*</td><td>2*</td><td>1*</td><td>4</td><td>3</td><td>5*</td><td>4</td><td>3</td><td>2*</td><td>1*</td><td>5*</td></tr> </table>   |    |    | 2  | 2  | 2  | 2  | 2  | 5  | 5  | 5  | 5 | 1 | 1 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5  | 5  | 5  | 1  | 1  | 1  | 1  | 1 | 1 | 2  | 2  | 2  | 5 | 4* | 3* | 2* | 1* | 4 | 3 | 5* | 4  | 3  | 2* | 1* | 5* |
|      |               | 2   | 2  | 2  | 2  | 2  | 5  | 5  | 5  | 5  | 1  | 1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 4    | 3             | 3   | 3  | 3  | 3  | 3  | 3  | 4  | 4  | 4  | 4  | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4             | 4   | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 4  | 5  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 5             | 5   | 5  | 1  | 1  | 1  | 1  | 1  | 1  | 2  | 2  | 2  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
| 5    | 4*            | 3*  | 2* | 1* | 4  | 3  | 5* | 4  | 3  | 2* | 1* | 5* |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |
|      | Tempo         | $t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \ t_8 \ t_9 \ t_{10} \ t_{11} \ t_{12} \ t_{13}$   |    |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |   |   |    |    |    |   |    |    |    |    |   |   |    |    |    |    |    |    |

**Figura 12.17** Prestazioni delle politiche di sostituzione delle pagine FIFO e LRU.

Per la sostituzione delle pagine FIFO, abbiamo  $\{p_i\}_4^{12} = \{2, 1, 4, 3\}$ , mentre  $\{p_i\}_3^{12} = \{1, 5, 2\}$ . Quindi, la sostituzione di pagina FIFO non soddisfa la proprietà dello stack. Ciò comporta un page fault all'istante  $t_{13}$  quando  $alloc_i = 4$ , ma non quando  $alloc_i = 3$ . Così, si verificano un totale di 10 page fault in 13 istanti di tempo quando  $alloc_i = 4$ , mentre, quando  $alloc_i = 3$ , si verificano 9 page fault. Per LRU, vediamo che  $\{p_i\}_3 \subseteq \{p_i\}_4$  per tutti gli istanti.

La [Figura 12.18](#) illustra come variano i page fault per gli algoritmi di sostituzione di pagina FIFO e LRU per la stringa dei riferimenti (12.6). Per semplicità, l'asse verticale mostra il numero totale di page fault piuttosto che la frequenza dei page fault. La [Figura 12.18\(a\)](#) illustra un'anomalia nel comportamento dell'algoritmo FIFO, cioè il numero di page fault aumenta quando aumenta l'allocazione della memoria per il processo. Questo comportamento anomalo fu riportato per primo da Belady ed è perciò noto come *anomalia di Belady*.

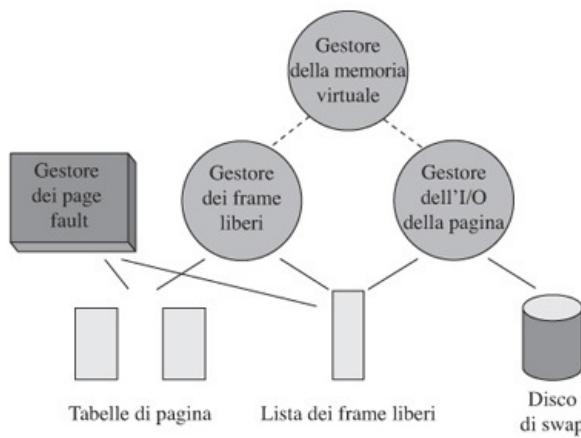


**Figura 12.18** (a) Anomalia di Belady nella sostituzione delle pagine FIFO; (b) comportamento dei page fault nella sostituzione delle pagine LRU.

Il gestore della memoria virtuale non può usare la sostituzione di pagina FIFO perché incrementando l'allocazione per un processo può aumentare la frequenza dei page fault del processo. Questa caratteristica renderebbe difficile affrontare il thrashing nel sistema. Tuttavia, quando viene usata la sostituzione di pagina LRU, il numero di page fault è una funzione non crescente di *alloc*. Di conseguenza è possibile combattere il thrashing incrementando il valore di *alloc* per ogni processo.

#### 12.4.1 Esempi realistici di politiche di sostituzione delle pagine

La Figura 12.19 mostra uno schema di esempio per il gestore della memoria virtuale. Il gestore della memoria virtuale mantiene una lista di frame liberi e prova a tenere pochi frame in questa lista a ogni istante. Il gestore della memoria virtuale consiste di due thread demoni. Il thread chiamato *gestore di frame liberi* è attivato dal gestore della memoria virtuale quando il numero di frame liberi va al di sotto di una soglia predefinita dallo stesso gestore della memoria virtuale. Il gestore di frame liberi scandisce le pagine presenti in memoria per identificare alcune pagine che possono essere liberate, e aggiunge i frame occupati da queste pagine alla lista dei frame liberi. Se la pagina contenuta in un frame aggiunta di recente è dirty, contrassegna la il frame come *dirty* nella lista dei frame liberi. Inoltre, azzera il bit di *validità* di questa pagina nella tabella delle pagine del processo al quale appartiene. Il gestore di frame liberi si mette esso stesso in uno stato di sleep quando il numero di page frame liberi supera un'altra soglia sempre stabilita dal gestore della memoria virtuale. Il thread chiamato *page I/O manager* esegue operazioni di page-out sui frame dirty nella lista dei frame liberi; esso azzera il *dirty* bit del frame quando la sua operazione di page-out è completata.



**Figura 12.19** Sostituzione delle pagine in pratica.

Il gestore dei page fault funziona come il gestore degli eventi del kernel. È attivato quando si verifica un page fault. Prima controlla se la pagina richiesta esiste in qualcuno dei frame nella lista dei frame liberi. In caso affermativo, semplicemente richiama la

pagina rimuovendo il suo frame dalla lista di quelli liberi, settando il bit di *validità* della pagina nella tabella delle pagine del processo e copiando il valore del *dirty* bit della page frame nel bit *modificato* della pagina. Questa operazione rende disponibile la pagina richiesta senza dover eseguire un'operazione di page-in. Se la pagina richiesta non esiste in alcun page frame, prende un page frame clean al di fuori della lista dei frame liberi e comincia l'operazione di page-in su di esso.

Effettivamente, la politica di sostituzione di pagina è implementata nel gestore dei frame liberi del gestore della memoria virtuale; tuttavia, nel seguito discuteremo la sostituzione di pagina come se fosse fatta direttamente dal gestore della memoria virtuale. La politica di sostituzione di pagina LRU sarebbe la scelta automatica di implementazione in un gestore della memoria virtuale perché soddisfa la proprietà dello stack. Tuttavia, la sostituzione di pagina LRU non è fattibile perché i computer non hanno sufficienti bit nel campo *Ref info* per memorizzare il tempo di ultimo riferimento. Infatti, la maggior parte dei computer hanno un singolo bit di riferimento per raccogliere informazioni relative ai riferimenti di pagina. Per tale motivo, le politiche di sostituzione delle pagine devono essere implementate usando solo il bit di riferimento. Questo requisito ha portato a una classe di politiche chiamate politiche NRU *non usato di recente* (*not recently used*), in cui il bit di riferimento è utilizzato per determinare se una pagina è stata riferita di recente, e qualche pagina che non è stata riferita di recente viene sostituita.

Una semplice politica NRU è la seguente: il bit di riferimento di una pagina è inizializzato a 0 quando la pagina è caricata, ed è impostato a 1 quando la pagina viene riferita. Quando si rende necessaria una sostituzione di pagina, se il gestore della memoria virtuale verifica che i bit di riferimento di tutte le pagine sono diventati 1, resetta i bit di tutte le pagine a 0 e arbitrariamente seleziona una delle pagine per la sostituzione; altrimenti, sostituisce una pagina il cui bit di riferimento è 0. Successive sostituzioni di pagina dipenderanno da quali pagine sono state riferite dopo il reset dei bit di riferimento. Gli algoritmi di sostituzione delle pagine detti *algoritmi di clock* forniscono la migliore discriminazione tra le pagine resettando i bit di riferimento delle pagine periodicamente, piuttosto che solo quando tutte diventano 1; in tal modo risulta possibile sapere se una pagina è stata riferita nell'immediato passato, per esempio nell'ambito delle ultime 100 istruzioni, piuttosto che dal momento in cui tutti i bit di riferimento sono stati resettati a 0.

### **Algoritmi di clock**

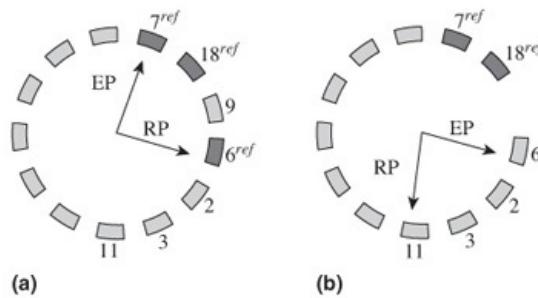
Negli algoritmi di clock, le pagine di tutti i processi in memoria sono inseriti in una lista circolare e i puntatori usati dagli algoritmi si muovono sulle pagine ripetutamente. Il nome attribuito a questi algoritmi deriva dal fatto che il movimento dei puntatori è analogo al movimento delle lancette sul quadrante di un orologio. La pagina puntata dal puntatore è quella in esame e, a seguito dell'azione su di essa, il puntatore viene avanzato per puntare alla pagina successiva. Gli algoritmi di clock possono essere anche applicati a livello di singolo processo quando l'allocazione di memoria per un processo deve essere diminuita. In questo caso, il gestore della memoria virtuale manterrà una lista separata di pagine per ogni processo e gli algoritmi di clock esamineranno solo la lista dei processi dei quali deve essere diminuita l'allocazione di memoria.

Nell'*algoritmo di clock a una lancetta*, una esecuzione è composta da due passi su ogni pagina. Nel primo passo, il gestore della memoria virtuale semplicemente resetta il bit di riferimento della pagina puntata dal puntatore. Nel secondo passo cerca tutte le pagine i cui bit di riferimento sono ancora off e li aggiunge alla lista dei frame liberi. Nell'*algoritmo di clock a due lancette*, sono utilizzati due puntatori. Un puntatore, che chiameremo il puntatore di resettaggio o *resetting pointer* (RP), servirà a resettare i bit di riferimento e l'altro puntatore, che chiameremo puntatore di controllo o *examining pointer* (EP), sarà utilizzato per controllare i bit di riferimento. Entrambi i puntatori vengono incrementati simultaneamente. Il frame al quale il puntatore di controllo punta è aggiunto alla lista dei frame liberi se il suo bit di riferimento è *off*. L'Esempio 12.8 descrive il funzionamento dell'algoritmo di clock a due lancette.

#### **Esempio 12.8 Algoritmo di clock a due lancette**

La [Figura 12.20](#) illustra il funzionamento dell'algoritmo di clock a due lancette utilizzato dal gestore dei frame liberi di [Figura 12.19](#). Il simbolo *ref* su una pagina implica che il bit di riferimento della pagina è settato ad 1; l'assenza di questo simbolo implica che il bit di riferimento è 0. Quando viene attivato il gestore dei frame liberi,

esamina la pagina 7, che è puntata dall'examining pointer [Figura 12.20(a)]. Il suo bit di riferimento è 1, quindi vengono avanzati entrambi i puntatori, il resetting e l'examining. In questo istante, il bit di riferimento della pagina 6 è resettato a 0 perché RP stava puntando a esso. Il puntatore examining si sposta su pagina 18 (e il resetting si sposta su pagina 2) perché, anch'esso ha il suo bit di riferimento settato ad 1. Ora resta sulla pagina 9. La pagina 9 ha il suo bit di riferimento a 0, quindi è rimossa dalla lista delle pagine in memoria e aggiunta a quella dei frame liberi. I puntatori resetting e examining puntano ora alle pagine 6 e 11, rispettivamente [Figura 12.20(b)]. La distanza tra i puntatori resetting ed examining fornisce proprietà diverse agli algoritmi di sostituzione di pagina. Se i puntatori sono molto vicini, sarà esaminata una pagina subito dopo il reset del suo bit di riferimento; di conseguenza solo le pagine usate di recente rimarranno in memoria. Se i puntatori del clock sono abbastanza lontani, solo le pagine che non sono state utilizzate da molto tempo saranno rimosse.



**Figura 12.20** Funzionamento dell'algoritmo di clock a due lancette.

## 12.5 Controllo dell'allocazione di memoria per un processo

Nel Paragrafo 12.2 abbiamo illustrato come un sovrdimensionamento di memoria per i processi abbia influenza sulle prestazioni di sistema a causa del basso grado di multiprogrammazione, mentre un sottodimensionamento di memoria per i processi porta al *thrashing*, che è caratterizzato da un page di I/O alto, scarsa efficienza di CPU, e prestazioni basse dei processi e del sistema. Mantenendo l'allocazione di memoria per un processo all'interno della zona operativa ottimale mostrata in Figura 12.6 si evitano sia il sovrdimensionamento che il sottodimensionamento di memoria per un processo. Tuttavia, non è chiaro il modo in cui il gestore della memoria virtuale decide il giusto numero di frame da allocare a ogni processo, ossia il giusto valore di *alloc* per ogni processo.

Vengono utilizzati due approcci per controllare l'allocazione di memoria per un processo:

- *allocazione di memoria fissa*: l'allocazione di memoria per un processo è fissa. Di conseguenza, la prestazione di un processo è indipendente dagli altri processi nel sistema. Quando si verifica un page fault in un processo, viene sostituita una delle sue pagine. Questo approccio è detto *sostituzione di pagina locale*;
- *allocazione di memoria variable*: l'allocazione di memoria per un processo può essere variata in due modi. Quando si verifica un page fault, tutte le pagine di tutti i processi che sono presenti in memoria possono essere prese in considerazione per la sostituzione. Ciò è identificato come *sostituzione globale delle pagine*. Alternativamente, il gestore della memoria virtuale può rivedere l'allocazione della memoria per un processo periodicamente sulla base della sua località e sul comportamento riguardo ai page fault, ma quando si verifica un page fault esegue una sostituzione locale di pagina.

Nell'allocazione di memoria fissa, le decisioni riguardanti l'allocazione di memoria sono eseguite staticamente. La memoria da allocare a un processo è determinata in base ad alcuni criteri quando il processo viene inizializzato. Per fare un semplice esempio, la memoria allocata per un processo potrebbe essere una frazione fissa della sua dimensione. La sostituzione di pagina è sempre eseguita localmente. L'approccio è semplice da implementare, e l'overhead della sostituzione di pagina è accettabile, poiché

solo le pagine del processo in esecuzione vengono considerate in una decisione di sostituzione di pagina. Tuttavia, questo metodo risente di tutti i problemi connessi a una decisione statica. Un sottodimensionamento o un sovradimensionamento di memoria per un processo può influenzare la prestazione del processo stesso e quella del sistema. Inoltre, il sistema può andare incontro al thrashing.

Nell'allocazione di memoria variabile, che utilizza la sostituzione globale delle pagine, l'allocazione per il processo attualmente in esecuzione può diventare troppo grande. Per esempio, se è utilizzata una politica di sostituzione delle pagine LRU o NRU, il gestore della memoria virtuale sarà impegnato per la maggior parte del tempo nella sostituzione delle pagine di altri processi perché i loro ultimi riferimenti precederanno quelli delle pagine del processo attualmente in esecuzione. L'allocazione di memoria per un processo bloccato andrà diminuendo, e quindi il processo andrà incontro a elevati tassi di page fault quando sarà rischedulato.

Nell'allocazione di memoria variabile che utilizza la sostituzione locale delle pagine, il gestore della memoria virtuale determina il giusto valore di *alloc* per un processo, di volta in volta. Nel seguito, illustriamo come può essere fatto in pratica.

### **Modello a working set**

Il concetto di un *working set* fornisce una base per decidere quanti e quali pagine di un processo dovrebbero essere in memoria per ottenere una buona prestazione di processo. Un gestore della memoria virtuale che segue il modello a working set si dice che usa un *allocatore di memoria working set*.

**Definizione 12.4 Working set** L'insieme delle pagine di un processo che sono state riferite nelle precedenti  $\Delta$  istruzioni del processo, dove  $\Delta$  è un parametro di sistema.

Diciamo che le precedenti  $\Delta$  istruzioni costituiscono la *finestra del working set*. Introduciamo la seguente notazione:

|                    |   |
|--------------------|---|
| $WS_i(t, \Delta)$  | working set per il processo $proc_i$ all'istante $t$ con dimensione della finestra $\Delta$ ;   |
| $WSS_i(t, \Delta)$ | dimensione del working set $WS_i(t, \Delta)$ , ossia il numero di pagine in $WS_i(t, \Delta)$ . |

Notare che  $WSS_i(t, \Delta) \leq \Delta$  perché una pagina può essere riferita più di una volta in una finestra di un working set. Omettiamo  $(t, \Delta)$  quando  $t$  e  $\Delta$  non sono importanti oppure il loro valore è chiaro dal contesto.

Un allocatore di memoria a working set tiene l'intero working set di un processo in memoria, o sospende il processo. Così, a ogni istante di tempo  $t$ , un processo  $proc_i$  o ha  $WS_i$  in memoria e  $alloc_i = WSS_i$ , oppure ha  $alloc_i = 0$ . Questa strategia aiuta ad assicurare un buon tasso di successo di accesso in memoria grazie al principio della località dei riferimenti. Esso, inoltre, evita il sottodimensionamento di memoria per un processo, evitando così il thrashing.

L'allocatore di memoria a working set deve variare il grado di multiprogrammazione in base ai cambiamenti nelle dimensioni del working set dei processi. Per esempio, se  $\{proc_k\}$  è l'insieme dei processi in memoria, il grado di multiprogrammazione dovrebbe essere decrementato se  $\sum_k WSS_k > \#frames$  dove  $\#frames$  è il numero totale di page frame in memoria. L'allocatore di memoria a working set rimuove alcuni processi dalla memoria finché  $\sum_k WSS_k \leq \#frames$ . Il grado di multiprogrammazione dovrebbe essere incrementato se  $\sum_k WSS_k < \#frames$  ed esiste un processo  $proc_g$  tale che  $WSS_g \leq (\#frames - \sum_k WSS_k)$ ,  $proc_g$  dovrebbe avere  $WSS_g$  page frame allocati e la sua esecuzione dovrebbe essere ripristinata.

Le variazioni nel grado di multiprogrammazione sono implementate come segue. Il gestore della memoria virtuale tiene due elementi di informazione per ciascun processo:  $alloc_i$  e  $WSS_i$ . Quando il grado di multiprogrammazione deve essere ridotto, il gestore della memoria virtuale seleziona il processo da sospendere, per esempio  $proc_i$ . Ora esegue un'operazione di page-out per ciascuna pagina modificata di  $proc_i$  e cambia lo

stato di tutti i frame allocati a esso a *libero*.  $alloc_i$  è settato a 0; tuttavia, il valore di  $WSS_i$  rimane immutato. Quando il grado di multiprogrammazione deve essere incrementato e il gestore della memoria virtuale decide di ripristinare  $proc_i$ , settterà  $alloc_i = WSS_i$  e allocherà un numero di frame pari al valore di  $alloc_i$ . Carica ora la pagina di  $proc_i$  che contiene la prossima istruzione da eseguire. Quando si verificano dei page fault, dovrebbero essere caricate altre pagine. In alternativa, il gestore della memoria virtuale carica tutte le pagine di  $WS_i$  quando viene ripristinata l'esecuzione di  $proc_i$ . Tuttavia, questo approccio può portare a caricamenti ridondanti di pagine perché alcune pagine in  $WS_i$  possono essere riferite di nuovo.

Le prestazioni di un allocatore di memoria a working set è dipendente dal valore di  $\Delta$ . Se  $\Delta$  è troppo grande, la memoria conterrà alcune pagine che probabilmente non saranno più riferite. In questo caso, si parla di sovrdimensionamento di memoria per i processi. Un valore troppo alto di  $\Delta$  forza anche il gestore della memoria virtuale a ridurre il grado di multiprogrammazione, influenzando così la prestazione del sistema. Se  $\Delta$  è troppo piccolo, c'è pericolo di sottodimensionamento di memoria per i processi, che determina un incremento nella frequenza dei page fault e la possibilità di thrashing.

### Implementazione di un allocatore di memoria a working set

L'uso di un allocatore di memoria a working set risente di una difficoltà pratica. È costoso determinare  $WS_i(t, \Delta)$  e  $alloc_i$  a ogni istante di tempo  $t$ . Per risolvere questa difficoltà, un gestore della memoria virtuale che utilizza un allocatore di memoria a working set può determinare i working set dei processi periodicamente piuttosto che a ogni istante. I working set determinati alla fine di un intervallo sono utilizzati per decidere i valori di  $alloc$  da usare nell'intervallo successivo. L'esempio successivo illustra questo approccio.

### Allocatore di memoria a working set

#### Esempio 12.9 Allocatore di memoria a working set

Un gestore della memoria virtuale ha 60 frame disponibili per l'allocazione ai processi utente. Ricalcola i working set di tutti i processi agli istanti di tempo  $t_{j \times 100}^+, j = 1, 2, \dots$ . Seguendo il calcolo dei working set, gestisce ciascun processo  $P_i$  come segue: setta  $alloc_i = WSS_i$  se può allocare a esso  $WSS_i$  frame, altrimenti setta  $alloc_i = 0$  e rimuove tutte le pagine di  $P_i$  dalla memoria. Il valore di  $alloc$  assegnato in  $t_{j \times 100}^+$  è tenuto costante fino a  $t_{(j+1) \times 100}^+$ .

La Figura 12.21 illustra il funzionamento dell'allocatore di memoria a working set. Mostra i valori di  $alloc$  e  $WSS$  per tutti i processi agli istanti di tempo  $t_{100}^+, t_{200}^+, t_{300}^+$  e  $t_{400}^+$ . A  $t_{100}^+$ ,  $WSS_4 = 10$ ,  $alloc_4 = 0$ , e  $\sum_{i=1, \dots, 3} WSS_i = 52$ . Ciò comporta che la dimensione del working set di  $P_4$  è 10 frame; tuttavia, il suo funzionamento è stato sospeso perché solo  $60 - 52 = 8$  frame sono liberi.

| Processo | $t_{100}$ |       | $t_{200}$ |       | $t_{300}$ |       | $t_{400}$ |       |
|----------|-----------|-------|-----------|-------|-----------|-------|-----------|-------|
|          | WSS       | alloc | WSS       | alloc | WSS       | alloc | WSS       | alloc |
| $P_1$    | 14        | 14    | 12        | 12    | 14        | 14    | 13        | 13    |
| $P_2$    | 20        | 20    | 24        | 24    | 11        | 11    | 25        | 25    |
| $P_3$    | 18        | 18    | 19        | 19    | 20        | 20    | 18        | 18    |
| $P_4$    | 10        | 0     | 10        | 0     | 10        | 10    | 12        | 0     |

**Figura 12.21** Funzionamento di un allocatore di memoria a working set.

A  $t_{200}^+$ , i valori di  $WSS_i$ ,  $i = 1, \dots, 3$  sono ricalcolati. Il valore di  $WSS_4$  è trasportato da  $t_{100}^+$  poiché  $P_4$  non è stato eseguito nell'intervallo  $t_{100} - t_{200}$ . Vengono assegnati ora a  $alloc_i$ ,  $i = 1, \dots, 3$  nuovi valori.  $P_4$  ancora non può essere ricaricato per mancanza di memoria poiché  $\sum_{i=1, \dots, 3} WSS_i = 55$ , quindi solo cinque frame sono liberi e  $WSS_4 = 10$ . A  $t_{300}^+$ ,  $P_4$  è ricaricato; tuttavia è scaricato ancora a  $t_{400}^+$ . Notare che nell'intervallo  $t_{100} - t_{400}$  l'allocazione minima per  $P_2$  è 11 frame e l'allocazione massima è 25 frame. Questa variazione è eseguita per adeguare l'allocazione di memoria del processo al suo

comportamento più recente.

L'espansione e la contrazione di  $alloc$  è eseguita come segue: in  $t_{200}$ , il gestore della memoria virtuale decide di ridurre  $alloc_1$  da 14 frame a 12 frame, così usa una politica tipo NRU per rimuovere due pagine di  $P_1$ . In  $t_{300}$ , incrementa  $alloc_1$  a 14 page frame. Alloca ulteriori due frame ad  $alloc_1$ . Questi frame saranno usati quando si verificheranno page fault durante il funzionamento di  $P_1$ .

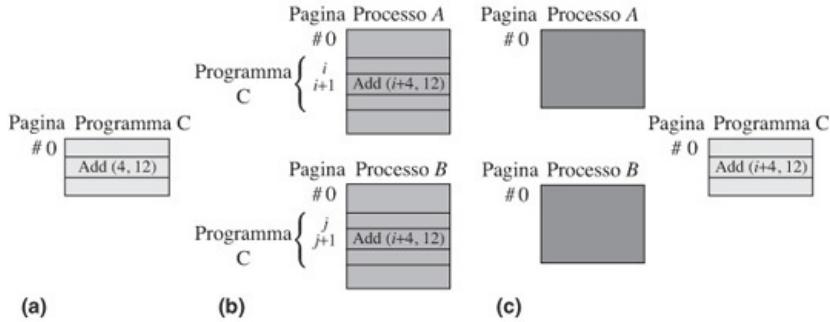
Il gestore della memoria virtuale può usare i bit di riferimento forniti dal supporto hardware di paginazione per determinare i working set. I bit di riferimento di tutte le pagine in memoria possono essere *off* quando vengono determinati i working set. Questi bit saranno di nuovo *on* appena queste pagine sono riferite nell'intervallo successivo. Eseguendo sostituzioni di pagina, il gestore della memoria virtuale può registrare quali tra le pagine sostituite avevano il loro bit di riferimento *on*. Il working set alla fine del prossimo intervallo sarà composto da queste pagine e tutte le pagine in memoria i cui bit di riferimento sono *on*.

L'implementazione dei working set, in questo modo, va incontro a un problema. Il reset dei bit di riferimento alla fine di un intervallo interferisce con le decisioni di sostituzione di pagina. Se si verifica un page fault in un processo subito dopo aver determinato il working set, i bit di riferimento della maggior parte delle pagine del processo in memoria saranno *off*. Di conseguenza il gestore della memoria virtuale non può differenziare queste pagine per la sostituzione di pagina. Se alcuni processi rimangono bloccati oppure non hanno un'opportunità di completare l'esecuzione in un intervallo, le loro allocazioni diminuiranno. Questo aspetto rende difficile la decisione relativa alla giusta dimensione di  $\Delta$ , la finestra del working set.

Un'alternativa è usare una finestra di working set per ciascun processo individualmente. Tuttavia, questo complicherebbe il gestore della memoria virtuale e aggiungerebbe overhead. Non risolverebbe, inoltre, il problema dell'interferenza con le decisioni di sostituzione di pagina. Per queste ragioni, i sistemi operativi non determinano effettivamente i working set dei processi in accordo con la Definizione 12.4. Nel Paragrafo 12.8.4 descriviamo come il sistema operativo Windows usa la nozione di working set di un processo.

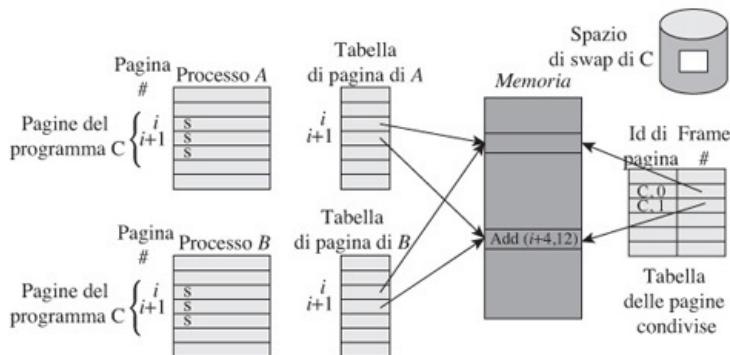
## 12.6 Pagine condivise

La condivisione dei programmi è stata discussa nel Paragrafo 11.3.3. La *condivisione statica* è conseguenza del binding statico eseguito da un linker o da un loader prima dell'esecuzione dell'inizio di un programma (Paragrafo 11.3.3). La [Figura 12.22\(a\)](#) mostra lo spazio di indirizzamento logico di un programma C. L'istruzione *Add* (4,12) nella pagina 1 ha il suo operando nella pagina 4. Con il binding statico, se due processi A e B condividono staticamente il programma C, allora C è incluso nel codice sia di A che di B. Supponiamo che la pagina 0 di C diventi la pagina  $i$  del processo A [[Figura 12.22\(a\)](#)]. L'istruzione *Add* (4,12) nella pagina 1 del programma C sarebbe rilocata per usare l'indirizzo  $(i + 4, 12)$ . Se la pagina 0 di C diventa la pagina  $j$  nel processo B, l'istruzione *Add* sarebbe rilocata per diventare *Add*  $(j + 4, 12)$ . Quindi, ogni pagina del programma C ha due copie negli spazi di indirizzamento di A e B. Queste copie possono essere presenti in memoria allo stesso istante se i processi A e B sono in esecuzione simultaneamente.



**Figura 12.22** Condivisione del programma C da parte dei processi A e B: (a) il programma C; (b) binding statico di C ai codici dei processi A e B; e (c) binding dinamico di C.

Il *binding dinamico* (Paragrafo 11.2) può essere usato per conservare memoria collegando la stessa copia di un programma o di dati a diversi processi. In questo caso, il programma o i dati da condividere dovrebbero conservare la loro identità [Figura 12.22(c)]. Si ottiene che il gestore della memoria virtuale mantiene una *tabella delle pagine condivise* per tenere informazioni relative alle pagine condivise in memoria. Il processo A effettua una chiamata di sistema per fare il binding del programma C come programma condiviso cominciando in una specifica pagina, diciamo, pagina  $i$ , nel suo spazio di indirizzamento logico. Il kernel invoca il gestore della memoria virtuale, che crea le entry nella tabella delle pagine di A per le pagine del programma C, e setta un flag  $s$  in ciascuna di queste entry per indicare che esso riguarda una pagina condivisa. Controlla poi, se le pagine del programma C hanno le entry nella *tabella delle pagine condivise*. In caso negativo, le crea, alloca lo spazio di swap per il programma C, e invoca il linker dinamico, che dinamicamente collega il programma C al codice del processo A. Durante il binding, le istruzioni address-sensitive di C vengono rilocate. Così, l'istruzione *Add* nella pagina 1 del programma C diventa *Add(i + 4, 12)* [Figura 12.22(c)]. Quando un riferimento a un indirizzo nel programma C va in page fault, il gestore della memoria virtuale verifica che è una pagina condivisa e verifica la tabella delle pagine condivise per controllare se la pagina richiesta è già in memoria (il che potrebbe accadere se un altro processo l'ha usata di recente). In tal caso, copia il numero di page frame della pagina dalla tabella delle pagine condivise nella entry di quella pagina nella tabella delle pagine di A; altrimenti, carica la pagina in memoria e aggiorna la sua entry nella tabella delle pagine di A e in quella delle *pagine condivise*. Vengono intraprese azioni simili quando il processo B dinamicamente collega il programma C allo start address della pagina  $i$  e i riferimenti alle pagine di C nelle istruzioni del processo B causano i page fault. La Figura 12.23 mostra la soluzione risultante.



**Figura 12.23** Condivisione dinamica del programma C da parte dei processi A e B.

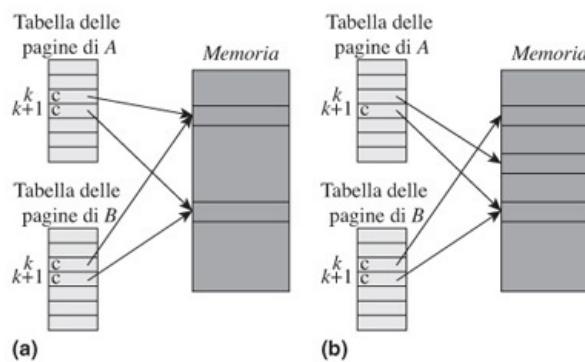
Affinché il binding dinamico di programmi funzioni, dovrebbero essere soddisfatte due condizioni. Il programma da condividere dovrebbe essere codificato come un programma rientrante in modo tale che possa essere invocato da più processi allo stesso istante

(Paragrafo 11.3.3). Il programma dovrebbe essere, inoltre, collegato agli identici indirizzi logici in ogni processo che lo condivide. Ciò assicurerrebbe che un'istruzione come *Add* ( $i + 4, 12$ ) nella pagina  $i + 1$  di [Figura 12.23](#) funzionerà correttamente in ognuno dei processi. Queste condizioni non sono necessarie quando i dati, piuttosto che un programma, sono collegati dinamicamente a diversi processi; tuttavia, i processi che condividono i dati dovrebbero essere sincronizzati nell'accesso ai dati condivisi per evitare conflitti.

Quando è implementata la condivisione di pagine facendo in modo che le entry della tabella delle pagine dei processi condivisi puntino allo stesso frame, le informazioni sui riferimenti di pagine per le pagine condivise saranno distribuite in molte tabelle delle pagine. L'algoritmo di sostituzione di pagina dovrà recuperare questa informazione per avere un quadro d'insieme dei riferimenti alle pagine condivise. Questo è piuttosto ostico. Un metodo migliore sarebbe conservare le informazioni riguardanti le pagine condivise nella *tabella delle pagine condivise* e mettere insieme le informazioni sui riferimenti di pagina per le pagine condivise nelle entry di questa tabella. Questa soluzione permette, inoltre, un diverso criterio di sostituzione delle pagine da usare per gestire le pagine condivise. Nel Paragrafo 12.8.4 descriviamo una tecnica usata nei sistemi operativi Windows.

### 12.6.1 Copy-on-write

La funzionalità copy-on-write è utilizzata per risparmiare memoria quando i dati nelle pagine condivise potrebbero essere modificati ma i valori modificati sono riservati a un processo. Quando i processi  $A$  e  $B$  si collegano dinamicamente a tali dati, il gestore della memoria virtuale implementa la soluzione mostrata in [Figura 12.24\(a\)](#), che è analoga alla soluzione illustrata in [Figura 12.23](#) a meno di un flag *copy-on-write* in ciascuna entry della tabella delle pagine, che indica se la proprietà *copy-on-write* è da utilizzare per quella pagina. Il simbolo  $c$  in una entry della tabella delle pagine in [Figura 12.23](#) indica che il flag *copy-on-write* è impostato per quella pagina. Se il processo  $A$  tenta di modificare la pagina  $k$ , la MMU genera un page fault vedendo che la pagina  $k$  è una pagina di *copy-on-write*. Il gestore della memoria virtuale fa, poi, una copia privata della pagina  $k$  per il processo  $A$ , cambia di conseguenza il numero di frame memorizzato nella entry della pagina  $k$  nella tabella delle pagine di  $A$ , e inoltre imposta ad off il flag *copy-on-write* in questa entry [[Figura 12.24\(b\)](#)]. Altri processi che condividono la pagina  $k$  continueranno a utilizzare la copia originale della pagina  $k$  in memoria; se la modificano, ognuna di esse avrà una copia privata della pagina.



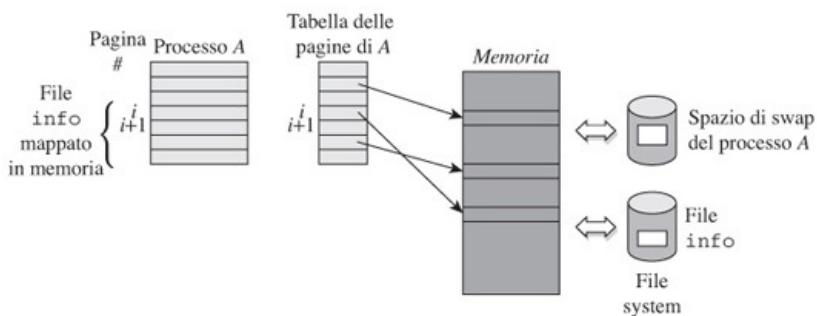
**Figura 12.24** Implementazione del copy-on-write: (a) prima e (b) dopo che il processo  $A$  modifichi la pagina  $k$ .

Nei sistemi Unix, un processo figlio inizia con il codice e i dati del processo padre; tuttavia, può modificare i dati e i valori modificati sono privati. La funzionalità *copy-on-write* è utilizzata per gli interi spazi di indirizzamento dei processi padre e figlio. Velocizza la creazione di un processo. Evita, inoltre, di copiare pagine di codice perché non sono mai modificate; saranno copiate solo le pagine di dati se modificate.

### 12.7 File mappati in memoria

La *mappatura in memoria* di un file da parte di un processo lega quel file a una parte dello spazio di indirizzamento logico del processo. Questo binding viene eseguito quando il processo effettua una chiamata di sistema *memory map*; essa è analoga al binding dinamico di programmi e dati discusso precedentemente nel Paragrafo 12.6. Dopo la mappatura in memoria di un file, il processo si riferisce ai dati nel file come se fossero dati localizzati nelle pagine del suo spazio di indirizzamento, e il gestore della memoria virtuale si coordina con il file system per caricare a richiesta parti del file, di dimensione di pagina, in memoria. Quando il processo aggiorna i dati contenuti in queste pagine, i bit *modificati* delle pagine sono settati ad *on* ma i dati non vengono immediatamente riportati nel file; le pagine dirty di dati verranno riportate nel file quando i frame che li contengono devono essere liberati. Quando il processo effettua una chiamata di *memory unmap*, il gestore della memoria virtuale scrive tutte le pagine dirty che contengono ancora i dati del file e cancella il file dallo spazio di indirizzamento logico del processo.

La [Figura 12.25](#) mostra la soluzione usata per la mappatura in memoria del file *info* da parte del processo *A*. Notare che le operazioni di page-in e page-out su queste pagine del processo *A* che non appartengono al file *info* coinvolgono lo spazio di swap del processo e sono eseguite dal gestore della memoria virtuale. La lettura e la scrittura dei dati dal file *info* sono eseguite dal file system congiuntamente al gestore della memoria virtuale. Se diversi processi mappano in memoria lo stesso file, abbiamo una soluzione analoga a quella mostrata in [Figura 12.23](#); questi processi condivideranno effettivamente il file mappato in memoria.



**Figura 12.25** Mappatura in memoria del file *info* da parte del processo *A*.

La [Tabella 12.5](#) sintetizza i vantaggi della mappatura in memoria dei file. La mappatura in memoria rende i file come se fossero dei record accessibili attraverso l'hardware della memoria virtuale. Questo è intrinsecamente più efficiente. Le operazioni di copia da memoria-a-memoria vengono in questo modo evitate: quando un processo accede ad alcuni dati in un file di input non mappato in memoria, il file system prima copia il record in un'area di memoria usata come file buffer o cache del disco ([Capitolo 14](#)). Il processo copia, poi, i dati dal buffer o dalla cache del disco in un suo spazio di indirizzamento (ossia, in qualche variabile) per accedervi. Così, vengono eseguite un'operazione di copia da disco-alla-memoria e una da memoria-a-memoria. Quando, invece, un file è mappato in memoria, l'operazione di copia (da memoria a memoria) dal buffer allo spazio di indirizzamento del processo non è necessaria poiché il dato è già una parte dello spazio di indirizzamento del processo. Analogamente, vengono eseguite poche operazioni di copia quando è modificato un file di dati. Al dato localizzato in una pagina che era stato letto o scritto durante una precedente operazione sul file, si può accedere senza I/O di disco, per cui la mappatura in memoria riduce il numero di operazioni di I/O eseguite durante l'elaborazione del file.

| Vantaggio                          | Descrizione  |
|------------------------------------|--|
| Dati del file come pagine          | L'accesso ai dati nel file è visto come accesso alle pagine, intrinsecamente più efficiente perché gestito dall'hardware della memoria virtuale.   |
| Evitare copie da memoria-a-memoria | I dati del file costituiscono una parte dello spazio di processo. Di conseguenza, il processo non deve copiarlo in una variabile per l'elaborazione.   |
| Meno operazioni di read/write      | I dati nel file sono letti e scritti una pagina alla volta, piuttosto che a ogni operazione, e quindi singole operazioni di lettura e scrittura possono essere sufficienti per varie operazioni su file. |
| Prefetching di dati                | Per letture sequenziali, i dati saranno già in memoria se la pagina che contiene i dati era stata letta nell'operazione precedente sul file.   |
| Accesso efficiente ai dati         | Può essere eseguito l'accesso ai dati del file in maniera efficiente indipendentemente dall'organizzazione dei file.   |

**Tabella 12.5** Vantaggi dei file mappati in memoria.

L'ultimo vantaggio, rappresentato dall'accesso efficiente ai dati in un file indipendentemente dalla sua organizzazione, deriva dal fatto che l'accesso ai dati in un file avviene attraverso l'hardware della memoria virtuale. Di conseguenza, è possibile accedere a qualsiasi parte dei dati in maniera ugualmente efficiente, mentre, come discusso nel [Capitolo 13](#), l'efficienza dell'accesso agli stessi dati in un file tramite le operazioni sui file dipenderà dall'organizzazione del file.

La mappatura in memoria dei file pone alcuni problemi prestazionali. Le operazioni di *apertura* e *chiusura* su di un file mappato in memoria comportano maggiore overhead rispetto alle operazioni di *apertura* e *chiusura* sui file normali. Ciò è causato dall'aggiornamento della tabella delle pagine e delle entry del TLB durante l'inizializzazione e la dismissione di quella parte dello spazio di indirizzamento del processo in cui il file è mappato. Il gestore della memoria virtuale deve, inoltre, distinguere tra pagine mappate in memoria e altre pagine nello spazio di indirizzamento. Le pagine di dati dirty dello spazio di indirizzamento sono scritte su disco solo quando si verifica un memory crunch, mentre le pagine dirty di un file mappato in memoria devono essere scritte su disco periodicamente per motivi di affidabilità del file. Quindi, il gestore della memoria virtuale deve creare un thread particolare che provveda alla scrittura delle pagine dirty dei file mappati in memoria.

## 12.8 Casi di studio

### 12.8.1 Memoria virtuale in Unix

Unix è stato portato su sistemi di elaborazione con diverse progettazioni hardware. Sono stati definiti differenti schemi per supportare le caratteristiche dell'hardware di paginazione di differenti macchine host. Questo paragrafo descrive alcune proprietà comuni a tutte le implementazioni della memoria virtuale in Unix e alcune interessanti tecniche usate nelle diverse versioni di Unix. Lo scopo è di fornire una panoramica delle problematiche pratiche nelle implementazioni della memoria virtuale piuttosto che studiare il gestore della memoria virtuale di una specifica versione Unix in dettaglio. Dove possibile, sostituiamo la terminologia Unix con quella che abbiamo usato nei precedenti paragrafi di questo capitolo.

#### **Spazio di indirizzamento logico e spazio di swap**

La tabella delle pagine di un processo si differenzia in tre tipi di pagine: residenti, non accessibili e pagine swapped-out. Una pagina residente è attualmente in memoria; una pagina non accessibile è una pagina alla quale non è ancora stato fatto un accesso durante l'elaborazione del processo e perciò non è mai stata caricata in memoria. Sarà caricata quando verrà fatto un riferimento a essa che genera un page fault. Come descritto successivamente, la pagina è presente o in un file o nello spazio di swap, a

seconda che sia una pagina di testo, ossia contiene istruzioni, oppure che sia una pagina di dati. Una pagina swapped-out, invece, è una pagina che è attualmente nello spazio di swap; a un page fault, viene ricaricata in memoria dalla sua posizione nello spazio di swap.

Una pagina non accessibile può essere una pagina di testo o una pagina di dati. Una pagina di testo viene caricata da un file eseguibile esistente nel file system. Localizzare una pagina del genere nel file system può richiedere letture di diversi blocchi di disco nell'inode e la tabella di allocazione dei file (Paragrafo 13.14.1). Per evitare questo overhead, il gestore della memoria virtuale gestisce le informazioni sulle pagine di testo in una tabella separata e fa riferimento a essa quando una pagina ha bisogno di essere caricata. Come descritto successivamente, il gestore della memoria virtuale 4.3BSD tiene questa informazione nella entry della stessa tabella delle pagine. Tale informazione viene sovrascritta dal numero di frame quando la pagina viene caricata in memoria, e quindi non è disponibile se la pagina viene rimossa dalla memoria e deve essere ricaricata. Per superare questa difficoltà, il gestore della memoria virtuale scrive una pagina di testo nello spazio di swap quando è rimossa dalla memoria per la prima volta, e poi la carica dallo spazio di swap a richiesta. Una pagina di dati è detta pagina zero-fill, cioè è riempita con degli zero quando il suo primo uso genera un page fault; poi, o è una pagina residente o una pagina swapped-out.

Una pagina di testo può rimanere in memoria anche se è contrassegnata come non residente nella sua entry nella tabella delle pagine. Questa situazione si verifica se qualche altro processo sta usando la pagina (o l'ha usata nel passato). Quando si verifica un page fault per una pagina di testo, il gestore della memoria virtuale prima controlla se la pagina esiste già in memoria. In caso affermativo, semplicemente mette l'informazione del frame nella sua entry della tabella delle pagine e la contrassegna come residente. Quest'azione evita un'operazione di page-in e risparmia memoria.

Per risparmiare spazio su disco, deve essere fatto uno sforzo per allocare meno spazio di swap possibile. Per iniziare, va allocato spazio di swap sufficiente per ospitare lo user stack e l'area dati. Successivamente, viene allocato lo spazio di swap in grandi blocchi quanto necessario. Questo approccio risente del problema che lo spazio di swap nel sistema può esaurirsi se l'area dati di un processo cresce; il processo deve, allora, essere sospeso o interrotto.

### ***Copy-on-write***

La semantica della *fork* richiede che il processo figlio ottenga una copia dello spazio di indirizzamento del padre. Queste semantiche possono essere implementate allocando aree di memoria distinte e uno spazio di swap per il processo figlio. Tuttavia, i processi figli frequentemente cancellano la copia dello spazio di indirizzamento del padre caricando qualche altro programma per l'esecuzione attraverso la chiamata *exec*. In ogni caso, un processo figlio potrebbe non voler modificare molti dei dati del padre. Di conseguenza la memoria e lo spazio di swap possono essere ottimizzati tramite la funzione *copy-on-write* (Paragrafo 12.6.1).

Il *copy-on-write* è implementato come segue: quando un processo viene creato con una *fork*, il reference count di tutte le pagine dati nello spazio di indirizzamento del padre è incrementato di 1 e tutte le pagine dati sono rese di sola lettura manipolando i bit nel campo dei privilegi di accesso delle loro entry della tabella delle pagine. Qualsiasi tentativo di modificare una pagina dati genera un protection fault. Il gestore della memoria virtuale trova che il contatore di riferimento della pagina è  $> 1$ , quindi ne deduce che non è un protection fault, ma un riferimento a una pagina *copy-on-write*. Quindi riduce il contatore, crea una copia della pagina per il processo figlio e assegna i privilegi di lettura e scrittura a questa copia settando i bit corretti nella sua entry della tabella delle pagine. Se il nuovo contatore di riferimento è  $= 1$ , abilita i privilegi di lettura e scrittura anche nella entry della tabella delle pagine che ha generato il page fault perché la entry non si riferisce più a una pagina condivisa.

### ***Uso efficiente della tabella delle pagine e dell'hardware di paginazione***

Se una pagina non è presente in memoria, il bit di *validità* della sua entry nella tabella delle pagine è "off." In questo scenario, i bit negli altri campi di questa entry, come il campo *Ref info* o il campo *frame #*, non contengono alcuna informazione utile. Di conseguenza, questi bit possono essere utilizzati per qualche altro scopo. Unix 4.3BSD usa questi bit per memorizzare l'indirizzo di un blocco del disco nel file system che contiene una pagina di testo.

L'architettura VAX 11 non prevede un bit di riferimento per raccogliere le informazioni dei riferimenti di pagina. La sua assenza è compensata usando il bit di *validità* in maniera nuova. Periodicamente, il bit di *validità* di una pagina è *off* anche se la pagina è in memoria. Il successivo riferimento alla pagina genera un page fault. Tuttavia, il gestore della memoria virtuale sa che questo non è un page fault "genuino", e quindi setta il bit di *validità* e ripristina il processo. In effetti, il bit di *validità* è usato come bit di riferimento.

### **Sostituzione delle pagine**

Il sistema consente a un processo di fissare una certa quantità delle proprie pagine in memoria per ridurre la sua percentuale di page fault e incrementare le sue prestazioni. Queste pagine non possono essere rimosse dalla memoria finché non vengono sbloccate dal processo. È interessante notare che, non ci sono I/O fixing di pagine in Unix poiché le operazioni di I/O avvengono tra un blocco del disco e un blocco nel buffer cache piuttosto che tra un blocco del disco e lo spazio di indirizzamento di un processo.

La sostituzione delle pagine in Unix è analoga allo schema di [Figura 12.19](#), incluso l'uso di un algoritmo di clock. Per facilitare le operazioni di page-in, il gestore della memoria virtuale di Unix tiene una lista di frame liberi e tenta di tenere sempre almeno il 5 percento dei frame totali in questa lista. A tale scopo viene creato un demone chiamato *demone del pageout* (che ha pid 2 nel sistema). È attivato ogni volta che il numero totale dei page frame liberi va al di sotto del 5 percento. Tenta di aggiungere pagine alla lista dei frame liberi e si pone automaticamente in stato di sleep quando la lista dei frame liberi contiene più del 5 percento di frame liberi. Alcune versioni di Unix usano due soglie - una alta e una bassa - invece di una sola soglia al 5 percento. Il demone va in stato di sleep quando si verifica che il numero delle pagine nella lista dei frame liberi supera la soglia alta. È attivato ogni volta che questo numero va al di sotto della soglia bassa. Questa soluzione evita frequenti attivazioni e disattivazioni del demone.

Il gestore della memoria virtuale divide le pagine che non sono fissate in memoria in pagine attive, ossia pagine che sono attivamente in uso da un processo, e pagine inattive, ossia pagine che non sono state riferite nel recente passato. Il gestore della memoria virtuale tiene due liste, la lista delle pagine attive e quella delle pagine inattive. Entrambe le liste sono trattate come code. Una pagina è aggiunta alla lista delle pagine attive quando diventa attiva, e alla lista delle inattive quando è in procinto di dover diventare inattiva. Quindi, la pagina attivata meno di recente è in cima alla lista delle pagine attive e la pagina inattiva più vecchia è in cima alla lista delle pagine inattive. Una pagina è spostata dalla lista delle inattive a quella delle attive quando viene riferita. Il demone del pageout tenta di mantenere un certo numero di pagine, calcolato come frazione del totale delle pagine residenti, nella lista delle inattive. Se raggiunge la fine della lista delle inattive aggiungendo page frame alla lista dei frame liberi, controlla se il numero totale di pagine nella lista delle pagine inattive è più piccolo del numero atteso. In tal caso, trasferisce un numero sufficiente di pagine dalla lista delle pagine attive alla lista delle pagine inattive.

Il demone del pageout si attiva quando il numero dei frame liberi va al di sotto della soglia bassa mentre il sistema gestisce un page fault. Libera i frame nel seguente ordine: frame che contengono pagine di processi inattivi, frame che contengono pagine inattive di processi attivi, e frame che contengono pagine attive di processi attivi. Il demone trova processi inattivi, se ce ne sono, e attiva lo swapper per scaricarli. Ritorna in stato di sleep se il numero dei frame liberi supera ora la soglia alta.

Se il numero di frame liberi dopo aver scaricato i processi inattivi è ancora al di sotto della soglia alta, il demone del pageout esamina la lista delle pagine inattive e decide se e quando aggiungere frame occupati dalle pagine inattive alla lista dei liberi. Un frame che contiene una pagina inattiva è aggiunto immediatamente alla lista dei liberi se la pagina non è riferita e non è dirty. Se la pagina è dirty e non ancora scaricata, il demone del pageout comincia un'operazione di page-out sulla pagina e procede per esaminare la prossima pagina inattiva. Se una pagina sta per essere scaricata, il demone semplicemente la scarta. Il bit *Modificato* di una pagina è resettato al completamento della sua operazione di page-out. Il page frame che contiene questa pagina sarà aggiunto alla lista dei frame liberi in un passo successivo se è ancora inattivo e se il demone trova che la sua operazione di page-out è stata completata. Il demone attiva lo swapper se non può aggiungere un numero sufficiente di page frame alla lista dei liberi. Lo swapper scarica uno o più processi attivi per liberare un numero sufficiente di frame.

Per ottimizzare il traffico di pagine, il gestore della memoria virtuale scrive pagine

dirty nello spazio di swap in gruppi detti cluster. Quando il demone trova una pagina dirty durante la sua scansione, esamina pagine adiacenti per controllare se anch'esse sono dirty. In tal caso, un cluster di pagine dirty viene scritto su disco in un'unica operazione di I/O. Un'altra ottimizzazione riguarda le operazioni di page-in ridondanti. Quando un frame  $f_i$  occupato da qualche pagina clean  $p_i$  viene aggiunto alla lista dei liberi, il bit di validità di  $p_i$  della entry nella tabella delle pagine è posto a 0. Tuttavia, la pagina non è immediatamente sovrascritta, caricando un'altra pagina nel frame. Questo accade dopo che la entry della pagina arriva in cima alla lista delle libere ed è allocata a qualche processo. Il successivo riferimento a  $p_i$  genererà un page fault poiché il bit di validità nella sua entry della tabella delle pagine è stato posto a 0. Se  $p_i$  è ancora in  $f_i$ , cioè se  $f_i$  è ancora nella lista dei liberi,  $f_i$  può essere semplicemente tolto dalla lista dei liberi e  $p_i$  può essere "ricollegato" allo spazio di indirizzamento logico del processo. Questo risparmia un'operazione di page-in e i conseguenti ritardi dovuti alla gestione dei page-fault.

### **Swapping**

Il gestore della memoria virtuale di Unix non utilizza un allocatore di memoria a working set a causa dell'elevato overhead che un allocatore di quel tipo comporta. Si concentra, invece, sul mantenere le pagine necessarie in memoria. Un processo è scaricato se tutte le sue pagine richieste non possono essere mantenute in memoria ed esistono nel sistema le condizioni che conducono al thrashing. Un processo inattivo, ossia un processo che è bloccato per molto tempo, può essere scaricato anche allo scopo di tenere un numero sufficiente di frame liberi. Quando tale situazione si verifica e diventa necessario uno swap-out, il demone del page-out attiva lo swapper, che è sempre il processo 0 nel sistema. Lo swapper cerca e scarica i processi inattivi. Se questo non libera memoria sufficiente, viene attivato ancora dal demone del page-out. Questa volta scarica il processo che è stato residente per il periodo di tempo più lungo. Quando ci sono processi scaricati nel sistema, lo swapper periodicamente controlla se esiste sufficiente memoria libera per ricaricare qualcuno di loro. A tale scopo viene usata la priorità di swap-in, che è funzione di quando il processo era stato scaricato, di quando l'ultimo era attivo, della sua dimensione e del suo valore *nice* - (Paragrafo 7.6.1 per i dettagli sul valore *nice*). Questa funzione assicura che nessun processo rimane scaricato indefinitamente. In Unix 4.3BSD un processo era ricaricato solo se poteva essere allocata tanta memoria quanta ne possedeva al momento del suo caricamento. In Unix 4.4BSD questo requisito è meno pressante; un processo è caricato se può essere allocata a esso memoria sufficiente a ospitare la sua struttura utente e lo stack kernel.

## **12.8.2 Memoria virtuale in Linux**

Linux utilizza una dimensione di pagina di 4 KB. Su architetture a 64 bit, usa una tabella delle pagine a tre livelli (Paragrafo 12.2.3). I tre livelli sono la page global directory, la page middle directory e la tabella delle pagine. Di conseguenza, un indirizzo logico consiste di quattro parti; tre di queste sono per i tre livelli e il quarto è il numero di byte all'interno di una pagina.

Linux usa un'interessante soluzione per eliminare le operazioni di page-in per le pagine che erano caricate precedentemente in memoria, ma erano contrassegnate per la sostituzione. Questo si realizza usando i seguenti stati per i frame: un frame *libero* è quello che non è stato allocato a un processo, mentre un frame *attivo* è quello che è in uso da un processo per il quale esso è stato allocato. Un frame *inattivo dirty* è stato modificato dal processo al quale era stato allocato ma non è più in uso dal processo. Una pagina *inattiva riciclata* è una pagina che era *inattiva dirty* ed è perciò scritta su disco. Una pagina *inattiva riciclata* diventa *inattiva clean* quando i suoi contenuti sono copiati su disco. Se il page fault di un processo per una pagina è in un frame contrassegnato *inattivo clean*, il frame è allocato un'altra volta al processo e la pagina è semplicemente segnata come presente in memoria. Se la pagina è in un frame segnato come *inattivo riciclato*, queste azioni vengono eseguite quando la sua operazione su disco termina. A meno del salvataggio delle operazioni su disco, questa soluzione evita anche l'accesso a una copia vecchia della pagina. Una pagina *inattiva clean* può anche essere, direttamente, allocata per un altro processo.

La sostituzione di pagina in Linux si basa sull'algoritmo di clock. Il kernel tenta di tenere sempre un numero sufficiente di frame liberi in modo che i page fault possano essere rapidamente gestiti usando uno dei frame liberi. Usa due liste chiamate *lista*

*attiva* e *lista inattiva*, e tiene la dimensione della lista attiva ai due-terzi della dimensione della lista inattiva. Quando il numero dei frame liberi va al di sotto della soglia più bassa, esegue un loop finché viene liberato qualche page frame. In questo loop esamina il frame alla fine della lista inattiva. Se il suo bit di riferimento è impostato a 1, resetta il bit e sposta il frame in cima alla lista; altrimenti, libera il frame. Quando deve essere mantenuto il bilanciamento tra le liste attiva e inattiva, l'algoritmo elabora qualche frame dalla fine della lista attiva allo stesso modo e, o li sposta in cima alla lista attiva, o li sposta in cima alla lista inattiva con il loro bit di riferimento a 1. Un frame è spostato sempre dalla lista inattiva alla lista attiva se è riferito da un processo.

Linux usa un allocatore buddy system per l'allocazione dei frame ai processi (Paragrafo 11.5.2). Questo metodo facilita l'esecuzione delle operazioni di I/O attraverso i classici bus DMA che usano gli indirizzi fisici, perché tali operazioni I/O richiedono che la memoria sia allocata in maniera contigua (Paragrafo 12.2.4).

Lo spazio di indirizzamento logico di un processo può consistere di diverse regioni di memoria virtuale; ogni regione può avere caratteristiche diverse ed è gestita utilizzando politiche diverse per il caricamento e la sostituzione delle pagine. Una pagina in una *regione di memoria zero-filled* è riempita di zeri al suo primo utilizzo. Una *regione file-backed* semplifica la mappatura in memoria di file. Le entry della tabella delle pagine delle sue pagine puntano ai buffer del disco usato dal file system. In questo modo, qualsiasi aggiornamento in una pagina di una regione si riflette immediatamente nel file ed è visibile agli utenti concorrenti del file. Una *regione di memoria privata* è gestita in maniera differente. Quando viene creato un nuovo processo tramite fork, il processo figlio ha una copia della tabella delle pagine del padre. In questo istante, le pagine di una regione di memoria privata assumono uno stato di copy-on-write. Quando un processo modifica una tale pagina, viene fatta per esso una copia privata della pagina.

### 12.8.3 Memoria virtuale in Solaris

Solaris supporta la dimensione multipla di pagina, in cui utilizza sia pagine normali che superpagine. Le superpagine sono usate automaticamente per processi con grandi spazi di indirizzamento; altri processi possono richiedere l'uso delle superpagine tramite la chiamata di sistema *memcntl*. Le superpagine non sono utilizzate per file mappati in memoria perché un piccolo cambiamento in una superpagina richiede che l'intera superpagina sia scritta su file; il che determina una considerevole penalizzazione delle prestazioni perché le superpagine dirty di un file mappato in memoria sono scritte frequentemente su disco per assicurare affidabilità del file (Paragrafo 12.7).

Una componente del gestore della memoria virtuale, chiamata *page scanner*, prova a tenere un numero sufficiente dei frame sulla *cache di pagina ciclica*, che è come la lista delle *inactive clean* di Linux, in modo che il gestore della memoria virtuale può allocare un frame dalla cache di pagina ciclica direttamente quando si verifica un page fault. Esso seleziona una pagina per la sostituzione dalla memoria, usando un algoritmo di clock a due lancette su base globale; la scrive su disco se è dirty, e aggiunge il suo frame alla cache di pagina ciclica. Il page scanner è implementato come due thread kernel analoghi a quelli mostrati in [Figura 12.19](#). Un thread identifica i page frame in aggiunta alla cache di pagina ciclica, mentre l'altro thread scrive le pagine dirty su disco da questi frame. Se la pagina per cui un processo ha causato un page fault si trova in un frame incluso nella cache di pagina ciclica, il gestore della memoria virtuale semplicemente rimuove il frame dalla cache di pagina ciclica e lo inserisce nella tabella delle pagine del processo. Questa soluzione risparmia un'operazione di page-in. Per ridurre il traffico di pagine, il page scanner non mette pagine condivise nella cache di pagina ciclica se un numero sufficientemente grande di processi le stanno condividendo.

*lotsfree* è un parametro del page scanner che indica quanti frame devono essere liberi a ogni istante. Il page scanner comincia l'analisi delle pagine usando l'algoritmo di clock a due lancette quando il numero dei frame liberi va al di sotto di *lotsfree*. Lo *scan rate*, che è il numero di pagine analizzate per secondo, viene modificato in base al numero di frame che sono attualmente liberi, è più piccolo quando questo numero è vicino al *lotsfree* ed è incrementato quando il numero va al di sotto di *lotsfree*. La differenza tra le due lancette dell'algoritmo di clock è calcolato all'istante di boot in base alla quantità di memoria del sistema. Questa differenza e la frequenza di scansione determinano l'elapsed time tra il resetting di un bit da parte di una lancetta dell'algoritmo di clock a due lancette e la valutazione da parte dell'altra lancetta dell'algoritmo. Un piccolo elapsed time implica che solo le pagine con accesso molto recente rimarranno in

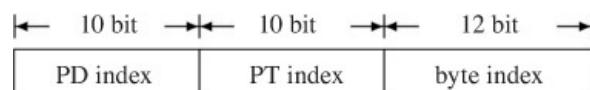
memoria, mentre un elapsed time più grande significa che solo le pagine alle quali non si è avuto accesso per molto tempo saranno rimosse dalla memoria. Per salvaguardare le prestazioni di sistema, il gestore della memoria virtuale limita la quantità di overhead di CPU che il page scanner può causare. Se il page scanner non riesce a gestire la richiesta di pagine libere usando l'algoritmo di clock, il gestore della memoria virtuale scarica i processi inattivi e libera tutti i frame occupati da essi.

Il gestore della memoria virtuale di Solaris si è evoluto nella sua forma attuale attraverso diversi aggiornamenti di progettazione. Prima di Solaris 6, il page scanner teneva una lista dei frame liberi che conteneva frame clean allocati a entrambi i processi utente e file. Il file system prendeva le pagine dalla lista dei liberi per sistemare i dati letti dai file. Durante i periodi di attività intensa su file, il file system effettivamente toglieva le pagine dagli spazi di indirizzamento dei processi utente, che influenzavano sulle prestazioni. Solaris 6 introdusse la proprietà detta *priority paging*, che assicurava che solo quei frame nella lista dei liberi che erano allocati a pagine di file sarebbero stati considerati per l'allocazione per i dati letti dai file. In questo modo, l'attività di elaborazione dei file non influenza l'esecuzione dei processi; tuttavia, i frame erano ancora allocati dalla lista dei liberi, la qual cosa causava alte frequenze di scansione e alti overhead del page scanner. Solaris 8 introdusse la cache di pagina ciclica descritta precedentemente e rese le pagine accessibili direttamente da parte del file system, in modo che l'attività di elaborazione del file non influenzasse le frequenze di scansione e l'overhead del page scanner.

#### 12.8.4 Memoria virtuale in Windows

Windows funziona su diverse architetture, di conseguenza supporta sia gli indirizzi logici a 32 bit che a 64 bit. La dimensione di pagina è 4 KB. Lo spazio di indirizzamento di un processo è 2 GB o 3 GB. Il resto dello spazio di indirizzamento logico è riservato al SO; il kernel è mappato in questa parte di spazio di indirizzamento di ogni processo. Sulla base dell'architettura, Windows usa tabella delle pagine a due, tre o quattro livelli e vari formati di entry di tabella delle pagine. La tabella delle pagine di un processo è essa stessa memorizzata nella parte riservata dello spazio di indirizzamento logico del processo.

Su un'architettura intel  $80 \times 86$ , Windows usa un'organizzazione con tabella delle pagine a due livelli simile a quella mostrata in [Figura 12.11](#). La tabella delle pagine di più alto livello è detta *page directory* (PD). La PD contiene 1024 entry di 4 byte ciascuna. Ogni entry nella PD punta alla *tabella delle pagine* (PT). Ogni tabella delle pagine contiene 1024 entry di 4 byte ognuna. Ogni indirizzo logico a 32-bit è diviso in tre componenti come mostrato di seguito:



Durante la traduzione degli indirizzi, il campo *PD index* viene utilizzato per localizzare una tabella delle pagine. Il campo *PT index* è usato per selezionare una entry a 32 bit della tabella delle pagine (PTE), che contiene un indirizzo a 20 bit del frame che contiene la pagina; il *byte index* è concatenato con il suo indirizzo per ottenere l'effettivo indirizzo fisico. Il gestore della memoria virtuale usa i rimanenti 12 bit in una entry della tabella delle pagine per indicare come il processo può accedere alla pagina - se read-only oppure read/write - e se il frame allocato a esso è *dirty*, ossia modificato, o *accessed*, ossia letto o modificato. Se la pagina non è in memoria, i 20 bit dell'indirizzo specificheranno l'offset nel file di paginazione, ossia lo spazio di swap. Se la pagina contiene codice, ne esisterà una copia in un file di codice, di conseguenza 28 bit nella entry della tabella delle pagine punteranno a una struttura dati del sistema che indica la sua posizione nel file di codice. Una pagina di questo tipo è direttamente caricata dal file di codice, quindi non è copiata nello spazio di swap.

Un frame può essere in uno di otto stati. Alcuni di questi stati sono:

- *valid*: la pagina è in uso;
- *free*: la pagina non è in uso;
- *zeroed*: la pagina è ripulita e disponibile per un uso immediato;
- *standby*: la pagina è stata rimossa dal working set del processo al quale era stata allocata, ma può essere "ricongiunta" al processo se fosse riferita nuovamente;

- *modified*: la pagina è dirty e deve ancora essere scritta;
- *bad*: non è possibile accedere alla pagina a causa di un problema hardware.

Un processo non può usare lo spazio di indirizzamento virtuale reso disponibile a esso direttamente; deve prima *riservarlo* per l'uso, e poi effettivamente *confermarlo* per allocare entità specifiche come file e oggetti. Quindi, solo alcune porzioni di spazio di indirizzamento logico di un processo possono essere riservate, e solo una parte dello spazio di indirizzamento logico riservato può essere attualmente in uso. Un accesso a una pagina che non è stata riservata e confermata genera una violazione di accesso. Quando un thread di un processo effettua una system call per richiedere memoria virtuale a una regione, il gestore della memoria virtuale genera un *virtual address descriptor* (VAD) che descrive l'intervallo degli indirizzi logici attribuiti a esso. Per minimizzare la dimensione della tabella delle pagine di un processo, il gestore della memoria virtuale la crea in modo incrementale, cioè l'entry della tabella delle pagine per una pagina richiesta è creata solo quando un accesso a essa genera un page fault. Per facilitare questa operazione, i VAD per le porzioni attribuite dello spazio di indirizzamento logico sono memorizzati in un albero AVL, che è un albero binario bilanciato.

Un oggetto *sezione* rappresenta una sezione di memoria che può essere condivisa. Può essere collegata a un file, nel qual caso fornisce file mappati in memoria, oppure alla memoria, nel qual caso fornisce memoria condivisa. Un processo mappa una *vista* di una sezione nel suo spazio di indirizzamento effettuando una chiamata di sistema con i parametri che indicano un offset nell'oggetto sezione, il numero di byte da mappare, e l'indirizzo logico nel suo spazio di indirizzamento dove l'oggetto è stato mappato. Quando il processo accede a una pagina nella vista per la prima volta, il gestore della memoria virtuale alloca un page frame e lo carica, a meno che esso non sia già presente in memoria perché un altro processo vi ha fatto accesso. Se la sezione di memoria ha l'attributo *based*, la memoria condivisa ha lo stesso indirizzo virtuale nello spazio di indirizzamento logico di ogni processo condiviso. Questo facilita la condivisione di codice tra processi (Paragrafo 12.6).

Per la condivisione delle pagine è adottata la proprietà copy-on-write (Paragrafo 12.6.1). È implementata settando il campo di protezione di una pagina a *read only*. Viene generata un'eccezione di protezione quando un processo tenta di modificare la pagina; il gestore della memoria, quindi, fa una copia privata della pagina a uso del processo.

Caricamento, accesso e sostituzione delle pagine condivise sono eseguite come segue: un *prototipo PTE* viene creato per ogni pagina condivisa in un'area di memoria riservata per i prototipi PTE. Ogni processo che usa la pagina condivisa ha un PTE per la pagina nella sua tabella delle pagine. Quando la pagina condivisa non esiste in memoria, perché non è ancora caricata o è stata rimossa dalla memoria, è marcata come invalid nel prototipo PTE e nei PTE nelle tabelle delle pagine di tutti i processi condivisi. Inoltre, i PTE nelle tabelle delle pagine dei processi sono impostate per puntare al prototipo PTE. Quando la pagina condivisa è riferita da uno dei processi condivisi, è caricata in memoria e il numero di frame dove è caricata è memorizzato in entrambi i prototipi PTE e il PTE del processo. Quando un altro processo fa riferimento a questa pagina, il suo PTE è aggiornato semplicemente copiando l'informazione del numero di frame dal prototipo PTE.

I translation look-aside buffer sono utilizzati per velocizzare la traduzione degli indirizzi. Nelle architetture a 32 bit, sono gestiti completamente dall'hardware MMU, mentre nelle architetture a 64 bit sono gestiti dal gestore della memoria virtuale. Quando un accesso di memoria da parte di un thread genera un page fault, il thread è bloccato finché termina l'operazione di page-in per la pagina. Diversi thread possono generare un page-fault per una pagina condivisa allo stesso istante. Questi page fault sono chiamati *collided page fault*. Il gestore della memoria virtuale assicura che tutti i thread coinvolti nel collided page fault si sbloccano quando termina l'operazione di page-in.

Per ridurre il numero di page fault attraverso la località dei riferimenti delle pagine, il gestore della memoria virtuale carica sempre qualche pagina precedente e seguente la pagina in cui si è verificato il page-fault in memoria. Durante il boot del sistema oppure iniziando un'applicazione, il prefetcher logico carica alcune pagine in memoria e monitora i page fault che si verificano in modo tale da poter caricare un insieme più efficace di pagine in memoria al prossimo boot del sistema o alla prossima esecuzione di un'applicazione.

Il kernel Windows usa la nozione di working set per controllare la quantità di memoria

allocata a un processo. Definisce una minima e una massima dimensione per il working set per ciascun processo; queste dimensioni sono determinate dalla configurazione della memoria del sistema, piuttosto che dalla dimensione o dalla natura di un processo. Per grandi configurazioni di memoria, le minime e le massime dimensioni del working set sono 50 e 345 pagine, rispettivamente. In un page fault, il kernel considera la quantità di memoria libera nel sistema, la dimensione del working set attuale del processo, e le sue minime e massime dimensioni del working set. Esso alloca un ulteriore frame al processo se la sua attuale allocazione è inferiore alla massima dimensione del working set ed esiste memoria libera; altrimenti, sostituisce una delle pagine del processo in memoria tramite un algoritmo di clock implementato usando l'*accessed* bit nella tabella delle pagine. Il *working set manager* è attivato periodicamente e quando i working set dei processi devono essere corretti. Se la quantità di memoria libera va al di sotto di una soglia a causa dell'allocazione di frame, esso esamina i working set le cui dimensioni superano la minima dimensione per il working set e rimuove dalla memoria quelle pagine che non sono state usate per molto tempo. Anche questo è fatto tramite un algoritmo di clock.

Il gestore della memoria virtuale mantiene un numero di liste di pagine: una lista delle pagine libere, una lista delle pagine inizializzate a zero, una lista delle pagine modificate e una lista delle pagine standby. Quando una pagina deve essere rimossa dalla memoria, oppure quando il processo a cui si riferisce è terminato, la pagina deve essere spostata nella lista delle pagine standby se era una pagina clean; altrimenti, deve essere spostata nella lista delle pagine modificate. (Ricordiamo che una pagina standby può essere semplicemente "ricongiunta" a un processo che la voglia utilizzare.) Il page writer scrive le pagine modificate e cambia il loro stato a standby. Usa due soglie – una soglia alta sul numero delle pagine modificate nel sistema e una soglia bassa sul numero delle pagine disponibili – per decidere quando le pagine devono essere scritte.

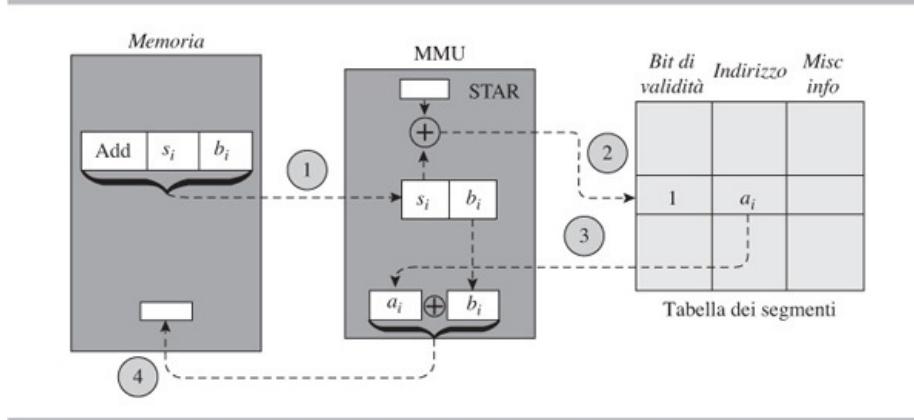
## 12.9 Memoria virtuale con uso di segmentazione

Un segmento è un'entità logica di un programma, come una funzione, una struttura dati o un oggetto; oppure è un modulo che consiste di alcuni o tutti questi elementi. Un programma è composto di segmenti. Durante l'esecuzione di un programma, il kernel tratta i segmenti come l'unità per l'allocazione di memoria. Questo determina l'allocazione di memoria non contigua per i processi, che fornisce un'efficiente uso della memoria riducendone la frammentazione. Essendo un'entità logica, un segmento è anche una conveniente unità di condivisione e protezione. Questa proprietà è utile nell'implementazione di grandi sistemi software che comprendono un insieme di moduli o oggetti.

Un indirizzo logico in un processo segmentato è visto come una coppia  $(s_i, b_i)$  in cui  $s_i$  e  $b_i$  sono rispettivamente l'id del segmento e l'id del byte. Ci sono differenze nel modo in cui  $s_i$  e  $b_i$  sono indicati in un indirizzo logico. Un metodo rappresenta ciascuno di loro numericamente. L'indirizzo logico consiste, quindi, di un numero per il segmento e un numero per il byte. Discuteremo il secondo metodo separatamente e successivamente in questo paragrafo. Lo spazio di indirizzamento logico di un processo è logicamente bidimensionale. Una dimensione è definita dall'insieme dei segmenti nel processo. Il numero di segmenti può variare, in base al numero massimo che può essere specificato dall'architettura dell'elaboratore o dal gestore della memoria virtuale. L'altra dimensione è definita dall'insieme di byte in un segmento. Il numero di byte in un segmento può variare, in base al massimo imposto dal numero di bit disponibili per rappresentare  $b_i$  in un indirizzo logico. La natura bidimensionale dello spazio di indirizzamento implica che l'ultimo byte di un segmento e il primo byte di un altro segmento non sono logicamente byte consecutivi. Infatti, se aggiungiamo 1 all'indirizzo dell'ultimo byte in un segmento, non punta al segmento successivo; è semplicemente un indirizzo non valido. Ci sono differenze significative rispetto alla paginazione. Ci sono anche somiglianze significative, che ora discuteremo nell'ambito della traduzione degli indirizzi.

La [Figura 12.26](#) mostra come viene effettuata la traduzione degli indirizzi nella memoria virtuale usando la segmentazione. Alcune analogie con la paginazione sono l'esistenza di una tabella dei segmenti o *segment table* (ST) per un processo, e un registro hardware speciale chiamato il *segment table address register* (STAR) che punta alla tabella dei segmenti di un processo. Per un indirizzo logico  $(s_i, b_i)$ , la traduzione degli indirizzi viene eseguita usando l'indirizzo di memoria trovato nella entry  $s_i$  della tabella dei segmenti e il numero di byte  $b_i$  nel segmento. Viene generato un missing

segment fault se il segmento  $s_i$  non esiste in memoria. Una differenza rispetto alla paginazione sta nel fatto che i segmenti non hanno una lunghezza standard. Di conseguenza, la traduzione degli indirizzi comporta l'aggiunta del numero di byte  $b_i$  all'indirizzo iniziale di  $s_i$ ; non può essere fatto usando la concatenazione di bit come avviene nella paginazione. La traduzione degli indirizzi può essere velocizzata usando un buffer per la traduzione degli indirizzi. Una entry nel buffer di traduzione degli indirizzi conterrà un identificativo del segmento e il suo indirizzo in memoria, copiato dalla sua entry nella tabella dei segmenti.



**Figura 12.26** Implementazione della memoria virtuale usando la segmentazione.

In uno spazio di indirizzamento logico ( $s_i, b_i$ ),  $s_i$  e  $b_i$  potrebbero essere specificati anche in forma simbolica, cioè con id. In questo caso, un indirizzo logico è della forma (*alpha*, *beta*) dove *alpha* è il nome di un segmento e *beta* è un id associato a un byte contenuto nel segmento *alpha*. La traduzione degli indirizzi di tali indirizzi logici viene fatta come segue. Compilando un segmento, il compilatore costruisce una tabella che mostra i byte id definiti nel segmento e i numeri di byte corrispondenti ai byte nel segmento. Questa tabella è disponibile al gestore della memoria virtuale durante la traduzione degli indirizzi. La chiameremo tabella di collegamento dei segmenti o *segment linking table* (SLT), e ci riferiamo a essa per *alpha* con la notazione  $SLT_{\alpha}$ . Durante la traduzione degli indirizzi, la MMU ottiene l'indirizzo di partenza di *alpha* dalla segment table, prende l'indirizzo di  $SLT_{\alpha}$  dal campo *Misc info* della entry di *alpha* e ottiene il numero di byte di *beta* dalla  $SLT_{\alpha}$ , e somma i due per ottenere l'effettivo indirizzo di memoria.

#### Esempio 12.10 Calcolo effettivo degli indirizzi nella segmentazione

La [Figura 12.27](#) illustra il calcolo dell'effettivo indirizzo per l'indirizzo logico (*alpha*, *beta*). La parte (a) della figura mostra il segmento *alpha*. *beta* e *gamma* sono due id associati a istruzioni o dati specifici in *alpha*. Questi id sono associati, rispettivamente, ai byte numerati 232 e 478 nel segmento. La tabella di collegamento dei segmenti SLT contiene alle entry per *beta* e *gamma* i valori 232 e 478, rispettivamente. La entry nella tabella dei segmenti di *alpha* indica che si trova nell'area di memoria con indirizzo di start 23480. Il numero di byte associato a *beta* è 232. Di conseguenza, l'indirizzo effettivo di (*alpha*, *beta*) sarà calcolato come  $23\ 480 + 232 = 23\ 712$ .

| (a) Segmento alpha |           | (b) Tabella dei segmenti |                 |           |            |           | (c) Tabella di collegamento dei segmenti |        |
|--------------------|-----------|--------------------------|-----------------|-----------|------------|-----------|--|--------|
| beta:...           | gamma:... | Nome                     | Bit di validità | Indirizzo | Dimensione | Misc info | Nome                                     | Offset |
|                    |           | alpha                    | 1               | 23480     | 764        |           | beta                                     | 232    |
|                    |           |                          |                 |           |            |           | gamma                                    | 478    |
|                    |           |                          |                 |           |            |           |  |        |

Tabella dei segmenti

Tabella di collegamento dei segmenti

**Figura 12.27** Uso degli id di segmento e delle word id simbolici.

Sia gli id numerici che quelli simbolici sono stati usati nella memoria virtuale segmentata. MULTICS è un sistema ben noto che usa gli identificativi simbolici.

### 12.9.1 Gestione della memoria

La gestione della memoria virtuale che utilizza la segmentazione ha alcune similitudini alla gestione della memoria con l'uso della paginazione. Un segmentation fault indica che un segmento richiesto non è presente in memoria. Viene eseguita un'operazione di segment-in per caricare il segmento. Se non c'è sufficiente memoria libera, alcune operazioni di segment-out potrebbero dover precedere il caricamento del segmento richiesto. Il gestore della memoria virtuale può usare un working set di segmenti per controllare l'allocazione di memoria per un processo. I segmenti potrebbero essere sostituiti in base alla NRU collezionando le informazioni dei riferimenti relativi ai segmenti in ciascuna entry del segmento.

Una differenza sostanziale rispetto alla memoria virtuale che usa la paginazione è che i segmenti non hanno dimensione fissa. La memoria liberata rimuovendo un segmento dalla memoria può non bastare per caricare un altro segmento. Di conseguenza, potrebbe essere necessario rimuovere molti segmenti prima di poter caricare un nuovo segmento. Le differenze nelle dimensioni dei segmenti possono provocare frammentazione esterna, che può essere bypassata o tramite tecniche di compattazione o tramite tecniche di riutilizzo di memoria come first-fit o best-fit. La compattazione è supportata dalla presenza della MMU, per cui, quando viene spostato un segmento in memoria viene modificato solo il campo indirizzo della entry della tabella dei segmenti. Tuttavia, il gestore della memoria virtuale dovrebbe assicurare che i segmenti che si stanno spostando non sono coinvolti in operazioni di I/O.

La natura bidimensionale dello spazio di indirizzamento logico permette a un segmento di incrementare o diminuire dinamicamente la sua dimensione. La crescita dinamica può essere gestita allocando un'area di memoria più grande per un segmento e liberando l'area di memoria precedentemente allocata a esso. Un segmento può crescere nella sua attuale locazione in memoria se esiste un'area libera aggiuntiva.

### 12.9.2 Condivisione e protezione

Due problematiche importanti nella condivisione e protezione dei segmenti sono:

- condivisione statica e dinamica dei segmenti;
- individuazione dell'uso di indirizzi non validi.

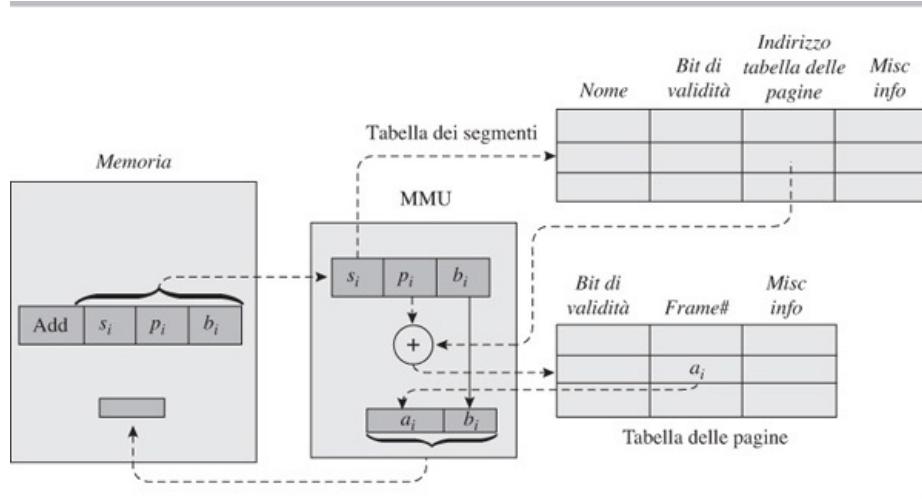
Un segmento è un'unità per la condivisione perché è un'entità logica in un processo. Può essere condiviso staticamente o dinamicamente usando gli schemi descritti precedentemente nel Paragrafo 12.6. Se gli id dei segmenti sono numerici, i segmenti devono occupare posizioni identiche negli spazi di indirizzamento logico dei processi che li condividono. Questo requisito è analogo a quello riguardante le pagine condivise nella memoria virtuale che usa la paginazione (Paragrafo 12.6 e [Figura 12.23](#)). Non si applica nel caso di id simbolici dei segmenti. I processi che condividono un segmento possono avere differenti privilegi di accesso ai programmi e ai dati. Il gestore della memoria virtuale mette i privilegi di accesso nel campo *Misc info* di una entry della tabella dei segmenti. Traducendo un indirizzo logico ( $s_i, b_i$ ), la MMU effettua due tipi di controlli di

protezione. Controlla se il tipo di accesso che si sta effettuando all'indirizzo logico è compatibile con i privilegi di accesso del processo per il segmento. Verifica, inoltre, se  $(s_i, b_i)$  è un indirizzo valido controllando se  $b_i <$  dimensione di  $s_i$ . Genera un interrupt di violazione della protezione di memoria se fallisce qualcuno di questi controlli.

### 12.9.3 Segmentazione con paginazione

La frammentazione esterna è presente nella memoria virtuale che utilizza la segmentazione in quanto le dimensioni dei segmenti sono diverse. Questo problema può essere risolto imponendo la paginazione su di uno schema di indirizzamento basato su segmenti. Un sistema che usa questo approccio conserva il vantaggio fondamentale della segmentazione - lo spazio di indirizzamento logico è bidimensionale, il che consente cambiamenti dinamici nella dimensione di un segmento - evitando la frammentazione esterna. Ogni segmento contiene un numero intero di pagine, e la gestione della memoria viene effettuata tramite la paginazione su richiesta. Questa soluzione può raggiungere un utilizzo più efficace della memoria poiché è necessario che siano in memoria sempre e solo le pagine richieste di un segmento. Tuttavia, la paginazione introduce frammentazione interna nell'ultima pagina di un segmento.

Un indirizzo logico in un sistema di questo tipo ha la forma  $(s_i, p_i, b_i)$ . Poiché ogni segmento consiste di un numero di pagine, viene creata una tabella delle pagine per ciascun segmento. La entry nella tabella dei segmenti di un segmento punta alla sua tabella delle pagine. La [Figura 12.28](#) illustra questa soluzione. Il campo *nome* della tabella dei segmenti è necessario solo se sono usati gli identificativi simbolici di segmento. La traduzione degli indirizzi ora implica un accesso alla tabella dei segmenti seguito da un accesso alla tabella delle pagine. Esso richiede due riferimenti in memoria se il segmento e le tabelle delle pagine sono tenute in memoria. Per velocizzare la traduzione degli indirizzi, dovrebbero essere impiegati i buffer di traduzione degli indirizzi per entrambi i riferimenti del segmento e della tabella delle pagine. Per questo scopo, può essere utilizzata una semplice estensione dello schema descritto precedentemente nel [Paragrafo 12.2.2](#). Altrimenti, può essere impiegato un solo buffer di traduzione degli indirizzi, ogni entry nel buffer che contiene una coppia  $(s_i, p_i)$  e il corrispondente numero di frame.



**Figura 12.28** Traduzione degli indirizzi nella segmentazione con paginazione.

La protezione della memoria può essere eseguita a livello dei segmenti attraverso lo schema descritto nel [Paragrafo 12.9.2](#). Le informazioni relative alla protezione per un segmento possono essere messe nella sua entry nella tabella dei segmenti, e possono essere copiate nella sua entry nel buffer di traduzione degli indirizzi. La validazione dell'accesso a livello di pagina non è necessaria.

## Riepilogo

La memoria virtuale è una parte della gerarchia di memoria composta da una memoria e da un disco. Durante l'esecuzione di un processo, alcune componenti del suo spazio di indirizzamento si trovano in memoria, mentre altre risiedono su un disco. Questa soluzione fa sì che la richiesta totale di memoria di un processo possa superare la dimensione della memoria del sistema. Permette, inoltre, che un maggior numero di processi risiedano in memoria contemporaneamente, perché ognuno di loro occupa meno memoria della propria dimensione. Le prestazioni di un processo dipendono dalla percentuale delle sue porzioni che devono essere caricate in memoria dal disco. In questo capitolo, abbiamo studiato le tecniche usate dal kernel per assicurare l'esecuzione efficiente di un processo e le buone prestazioni di sistema.

Due operazioni fondamentali nel funzionamento della memoria virtuale che utilizzano la paginazione sono la *traduzione degli indirizzi* e il *caricamento delle pagine su richiesta*. La *memory management unit* (MMU), che è un'unità hardware, e il *gestore della memoria virtuale*, che è una parte del kernel, implementano insieme queste due azioni. La memoria è divisa in parti chiamate *frame*, la cui dimensione è uguale alla dimensione delle pagine. Il gestore della memoria virtuale tiene una *tabella delle pagine* per ciascun processo per indicare quali delle sue pagine si trovano in quale frame di memoria. Quando un operando nell'istruzione corrente in un processo si trova in una delle pagine presenti in memoria, la MMU ottiene dalla tabella delle pagine il numero del frame dove si trova e lo usa per calcolare l'indirizzo effettivo di memoria dell'operando. Se la pagina non è in memoria, la MMU genera un interrupt chiamato *page fault*, e il gestore della memoria virtuale carica la pagina in memoria. Viene usato un *translation look-aside buffer* (TLB) per velocizzare la traduzione degli indirizzi; il TLB memorizza alcune entry delle tabelle delle pagine dei processi. In pratica, vengono usate la *tabella delle pagine invertita* e la *tabella delle pagine multilivello* perché richiedono meno memoria della tabella delle pagine convenzionale.

Il gestore della memoria virtuale deve prendere due decisioni chiave che influenzano le prestazioni di un processo: quale pagina rimuovere dalla memoria per liberare spazio per una nuova pagina richiesta da un processo, e quanta memoria allocare a un processo. Usa un *algoritmo di sostituzione delle pagine* per decidere quale pagina rimuovere dalla memoria. Il principio empirico della *località dei riferimenti* indica che una pagina a cui è stato fatto un accesso di recente è più probabile che sarà usata in futuro rispetto a una pagina a cui non è stato fatto accesso di recente. Di conseguenza, l'algoritmo di sostituzione di pagina *least recently used* (LRU) rimuove la pagina che è stata usata meno di recente. Esso possiede la *proprietà dello stack*, il che garantisce che la percentuale dei page fault non aumenterà se l'allocazione di memoria per un processo viene incrementata. Tuttavia, è oneroso reperire l'informazione che una pagina era meno referenziata. Di conseguenza, le MMU tipicamente prevedono un singolo bit per fornire informazioni riguardo i riferimenti di pagina, e in pratica, viene utilizzata una classe di algoritmi di sostituzione di pagina chiamati algoritmi *not recently used* (NRU). Gli *algoritmi clock* sono una sottoclasse ampiamente usata degli algoritmi NRU.

Il *working set* di un processo è l'insieme di pagine distinte usate o riferite più di recente. La sua dimensione fornisce un utile puntatore a quante pagine del processo dovrebbero essere in memoria per assicurare una buona prestazione del processo. Il gestore di memoria virtuale può usare la nozione di working set per evitare la situazione detta *thrashing* in cui la maggior parte dei processi nel sistema hanno quantità insufficienti di memoria allocate, quindi produce un'alta percentuale di page fault e viene fatto poco lavoro utile nel sistema.

Un sistema operativo usa tecniche speciali che supportano la memoria virtuale per velocizzare l'esecuzione dei processi. La tecnica *copy-on-write* evita di tenere copie identiche di pagine condivise in memoria, mentre la *mappatura in memoria* dei file fa in modo che il file sia trattato come una parte dello spazio di indirizzamento di un processo, velocizzando, così, gli accessi ai suoi dati.

## Domande

- 12.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
- In un sistema di elaborazione che supporta la memoria virtuale, il numero di bit in un indirizzo logico può superare il numero di bit in un indirizzo fisico.
  - Un'operazione di page-out è sempre necessaria nella sostituzione di pagina,

- indipendentemente dal fatto che la pagina da sostituire sia dirty o meno.
- c. La perdita di protezione si può verificare se una entry, che era stata creata nel translation look-aside buffer (TLB), durante l'esecuzione di un processo viene usata durante l'esecuzione di un altro processo.
  - d. L'organizzazione basata sulla tabella delle pagine invertita richiede più accessi alla memoria durante la traduzione degli indirizzi rispetto all'organizzazione convenzionale delle tabelle delle pagine.
  - e. La politica di sostituzione di pagina FIFO garantisce che, allocando più frame per un programma, la sua percentuale di page fault sarà ridotta.
  - f. Se l'hardware della memoria virtuale prevede un singolo bit di riferimento e sono settati i bit di riferimento nelle entry della tabella delle pagine di tutte le pagine residenti in memoria, l'algoritmo di sostituzione di pagina LRU degenera nella sostituzione di tipo FIFO.
  - g. Non si verificheranno page fault durante l'esecuzione di un processo se tutte le pagine incluse nel working set di un processo sono in memoria a ogni istante.
  - h. Un elevato traffico di pagine implica il verificarsi di thrashing.
  - i. Se una singola copia di un programma C è condivisa da due processi A e B, le pagine di C occuperanno le stesse posizioni nelle tabelle delle pagine dei processi A e B.
- 12.2. Selezionare l'alternativa corretta in ciascuna delle seguenti affermazioni.
- a. Se l'hardware della memoria virtuale prevede un singolo bit di riferimento in una entry della tabella delle pagine, il bit di riferimento nella entry della pagina  $p_i$  del processo  $P_k$  indica che:
    - i. la pagina  $p_i$  sarà probabilmente riferita in seguito;
    - ii. la pagina  $p_i$  sarà la prossima pagina a essere riferita durante l'esecuzione di  $P_k$ ;
    - iii. la pagina  $p_i$  è stata riferita da quando è stata caricata in memoria;
    - iv. la pagina  $p_i$  è la pagina più recentemente referenziata di  $P_k$ .
  - b. Durante l'esecuzione di un processo  $P_k$ , il translation look-aside buffer contiene:
    - i. alcune entry dalla tabella delle pagine di  $P_k$ ;
    - ii. le entry più recentemente riferite della tabella delle pagine di  $P_k$ ;
    - iii. le ultime entry della tabella delle pagine di  $P_k$ ;
    - iv. le entry meno recentemente riferite della tabella delle pagine di  $P_k$ .
  - c. La proprietà dello stack di un algoritmo di sostituzione di pagina implica che allocando più memoria per un processo:
    - i. si verificano meno page fault;
    - ii. si verificano più page fault;
    - iii. il numero di page fault è minore o uguale;
    - iv. nessuna delle (i)-(iii).
  - d. Se  $pfr_i$  e  $pfr_j$  sono le percentuali di page fault dei processi  $P_i$  e  $P_j$  quando il processo  $P_i$  ha il 5 percento delle sue pagine in memoria, il processo  $P_j$  ha il 10 percento delle sue pagine in memoria e la politica di sostituzione di pagina possiede la proprietà dello stack, allora:
    - i.  $pfr_i < pfr_j$
    - ii.  $pfr_i \leq pfr_j$
    - iii.  $pfr_i > pfr_j$
    - iv. non si può dire alcunché sugli ordini di grandezza di  $pfr_i$  e  $pfr_j$
  - e. Se  $pfr_i$  e  $pfr'_i$  sono le percentuali di page fault del processo  $P_i$  quando viene eseguito con il 5 percento delle sue pagine e con il 10 percento delle sue pagine in memoria, rispettivamente, e la politica di sostituzione di pagina possiede la proprietà dello stack, allora:
    - i.  $pfr_i < pfr'_i$
    - ii.  $pfr_i \geq pfr'_i$
    - iii.  $pfr_i > pfr'_i$
    - iv. nulla si può dire sugli ordini di grandezza di  $pfr_i$  e  $pfr'_i$
  - f. Il thrashing può essere superato se
    - i. È incrementato il grado di multiprogrammazione
    - ii. È incrementata la velocità di I/O

- iii. L'allocazione di memoria per un processo è controllata dalla dimensione del suo working set
- iv. Nessuna delle (i)-(iii)

## Problemi

- 12.1. Le tabelle delle pagine sono memorizzate in una memoria che ha un tempo di accesso di 100 nanosecondi. Il translation look-aside buffer (TLB) può contenere 64 entry della tabella delle pagine e ha un tempo di accesso di 10 nanosecondi. Durante l'esecuzione di un processo, si verifica che l'85 percento del tempo una entry di una tabella delle pagine richiesta si trova nel TLB e solo il 2 percento dei riferimenti genera dei page fault. Il tempo medio per la sostituzione di pagina è 2 ms. Calcolare il tempo effettivo di accesso in memoria.
- 12.2. Usando le velocità di accesso e gli hit ratio menzionati nel Problema 1, calcolare il tempo effettivo di accesso in memoria nelle organizzazioni delle tabelle delle pagine a due livelli, tre livelli e quattro livelli.
- 12.3. I tre approcci alla paginazione del kernel nella memoria virtuale sono:
  - a. rendere il kernel permanentemente residente in memoria;
  - b. paginare il kernel in maniera analoga alla paginazione dei processi utente;
  - c. rendere il kernel una parte importante dello spazio di indirizzamento logico di ciascun processo nel sistema e gestire le sue pagine come pagine condivise.
 Quale approccio si raccomanda? Fornire le motivazioni della scelta.
- 12.4. Le prestazioni dell'esecuzione di un processo nella memoria virtuale dipendono dalla località dei riferimenti mostrata durante la sua esecuzione. Sviluppare un insieme di linee guida che un programmatore può seguire per ottenere una buona prestazione di processo. Descrivere la motivazione di ciascuna linea guida.  
(*Suggerimento:* considerare gli array di riferimenti che si verificano nei loop annidati!)
- 12.5. Dare un esempio di stringa dei riferimenti alle pagine per un processo che produce più page fault quando viene utilizzata la politica di sostituzione di pagina LRU con  $alloc = 5$  rispetto a quando viene utilizzata la politica di sostituzione di pagina ottimale con  $alloc = 5$ .
- 12.6. Un processo effettua  $r$  riferimenti di pagina durante la sua esecuzione. La stringa dei riferimenti alle pagine del processo contiene  $d$  distinti numeri di pagina. La dimensione del processo è  $p$  pagine e ha  $f$  frame allocati tutti durante la sua esecuzione.
  - a. Qual è il minimo numero di page fault che si possono verificare durante la sua esecuzione?
  - b. Qual è il massimo numero di page fault che si possono verificare durante la sua esecuzione?
- 12.7. Provare la validità della seguente affermazione, se la politica di sostituzione di pagina usa un'allocazione fissa di memoria e la sostituzione di pagina locale: "Se un processo non modifica nessuna delle sue pagine, allora è ottimale sostituire la pagina il cui successivo riferimento è il più lontano nella stringa dei riferimenti alle pagine." Mostrare che questa politica non può condurre al minimo numero di operazioni di page-in e page-out se il processo modifica qualcuna delle sue pagine.
- 12.8. Cos'è l'anomalia di Belady? Mostrare che l'algoritmo di sostituzione di pagina che possiede la proprietà dello stack non può soddisfare l'anomalia di Belady.
- 12.9. Provare che la politica di sostituzione di pagina LRU possiede la proprietà dello stack.
- 12.10. La sostituzione di pagina ottimale può essere implementata sostituendo la pagina il cui successivo riferimento è il più lontano nella stringa dei riferimenti alle pagine. Questa politica possiede la proprietà dello stack? L'algoritmo di clock possiede la proprietà dello stack?
- 12.11. Per la stringa dei riferimenti alle pagine (12.6),
  - a. mostrare il working set in ogni istante di tempo se la dimensione della finestra del working set è (i) tre istruzioni, (ii) quattro istruzioni;
  - b. confrontare l'esecuzione e le prestazioni dell'allocatore working set con gli allocator FIFO e LRU.
- 12.12. Si consideri un allocatore working set usato per una stringa di riferimento della

- pagina con due valori di  $\Delta$ ,  $\Delta_1 < \Delta_2$ .  $pfr_1$  e  $pfr_2$  sono i tassi di page fault quando vengono usate  $\Delta_1$  e  $\Delta_2$ , rispettivamente. Vale  $pfr_1 \geq pfr_2$  se i working set sono ricalcolati (a) dopo ogni istruzione e (b) dopo ogni  $n$  istruzioni per qualche  $n$ ?
- 12.13. Descrivere le azioni di un gestore della memoria virtuale che usa un allocatore di memoria basato su working set quando decide di ridurre il grado di multiprogrammazione. Indicare chiaramente come usa e manipola le sue strutture dati per questo scopo.
- 12.14. Spiegare, con esempi, perché la dimensione del working set di un processo si può incrementare o decrementare durante la sua esecuzione.
- 12.15. Giustificare la seguente affermazione: "Il thrashing si può verificare quando viene utilizzato un allocatore di memoria basato su working set. Tuttavia, non può durare molto."
- 12.16. Un gestore della memoria virtuale usa la seguente politica di sostituzione di pagina: quando si nota una combinazione di elevata percentuale di page fault nel sistema e scarsa efficienza di CPU, ridurre l'allocazione per ciascun processo e caricare un ulteriore processo. Commentare l'efficacia di questa politica.
- 12.17. Spiegare perché l'algoritmo di clock a due lancette per la sostituzione di pagina è superiore a quello a una lancetta (Paragrafo 12.8.1).
- 12.18. Un gestore della memoria virtuale implementa un allocatore di memoria basato su working set e usa la condivisione dinamica delle pagine. Descrivere le azioni di ripulitura eseguite da esso nelle seguenti situazioni.
- Quando si verifica un page fault.
  - Quando una pagina condivisa viene tolta dal working set di uno dei processi che la condividono.
- 12.19. La quantità di memoria allocata per un processo in un sistema che usa memoria virtuale è tenuta costante e la dimensione di pagina è variata. (Questa azione varia il numero di pagine del processo in memoria.) Tracciare un grafico delle dimensioni di pagina rispetto alle attese percentuali di page fault.
- 12.20. Il grado di multiprogrammazione in un sistema che usa memoria virtuale varia cambiando l'allocazione della memoria per i processi. Tracciare un grafico del grado di multiprogrammazione rispetto all'efficienza di CPU. Spiegare la natura del grafico nella regione in cui il grado di multiprogrammazione è alto.
- 12.21. Ci riferiamo a "istruzioni nel passato" durante l'esecuzione di un processo come segue: l'istruzione eseguita più di recente si dice sia "1 istruzione nel passato" del processo, l'istruzione prima di essa si dice sia "2 istruzioni nel passato," ecc. Un allocatore di memoria si riferisce al riferimento di pagina nell'istruzione che è  $i$  istruzioni nel passato come il riferimento di pagina  $-i$ . Usa un parametro  $w$  e le seguenti regole per l'allocazione di memoria e la sostituzione di pagina.
- Non fa nulla se il successivo riferimento di pagina corrisponde al riferimento di pagina  $-w$ .
  - Altrimenti, se il successivo riferimento di pagina corrisponde al riferimento di pagina  $-i$  per qualche  $i < w$ , esegue quanto segue: se il riferimento di pagina  $-w$  non corrisponde al riferimento di pagina  $-j$  per qualche  $j < w$ , allora riduce l'allocazione di memoria per il processo di un page frame e rimuove la pagina meno recentemente usata, altrimenti non fa nulla.
  - Altrimenti, se il successivo riferimento di pagina genera un page fault e il riferimento di pagina  $-w$  non corrisponde con il riferimento di pagina nell'istruzione  $-j$  per qualche  $j < w$ , allora esegue una sostituzione di pagina usando la politica di sostituzione di pagina LRU.
  - Altrimenti, incrementa l'allocazione di memoria per il processo di un page frame e carica la pagina contenuta nel successivo riferimento di pagina.
- Mostrare che le azioni dell'allocatore di memoria sono equivalenti alle azioni dell'allocatore di memoria a working set con  $\Delta = w$ .
- 12.22. Confrontare le seguenti proposte di gestione della memoria nella memoria virtuale usando la segmentazione con paginazione.
- Usare la politica LRU all'interno di un processo.
  - Usare la politica LRU all'interno di un segmento.
- 12.23. Commentare la validità della seguente affermazione: "Nella memoria virtuale che usa la segmentazione con paginazione, il ruolo della segmentazione è limitato alla condivisione. Non gioca alcun ruolo nella gestione di memoria."
- 12.24. Un'operazione di I/O consiste nell'esecuzione di una sequenza di comandi di I/O.

Un'operazione di I/O *self-describing* è un'operazione di I/O di cui alcuni comandi di I/O vengono letti da un precedente comando di I/O della stessa operazione di I/O. Per esempio, consideriamo l'operazione di I/O:

1. read  $d, 6, aaa$
2. read  $d, count, bbb$

dove  $d$  è l'id del dispositivo di I/O. Il primo comando di I/O legge 6 byte nell'area di memoria con indirizzo  $aaa$ . Supponiamo che questa sia l'area dove sono memorizzati i campi che contengono  $count$  (2 byte) e  $bbb$  (4 byte) del secondo comando di I/O. Così, il primo comando di I/O modifica il secondo comando di I/O. Siano  $n$  e  $ccc$  i valori letti nei campi  $count$  e  $bbb$ , rispettivamente, dal primo comando di I/O. Dopo che l'I/O per il primo comando di I/O è completato, il secondo comando di I/O legge  $n$  byte nell'area di memoria con indirizzo  $ccc$ . I dati per questa operazione di I/O saranno:

$n, ccc,$    
 $n$  byte di dati

I metodi di esecuzione dell'I/O nella memoria virtuale descritti nel Paragrafo 12.2.4 possono gestire correttamente le operazioni di I/O self-describing? Giustificare in maniera chiara la risposta. In una forma semplificata dell'I/O self-describing, il primo comando di I/O legge solo 2 byte e li memorizza nel campo  $count$ . I metodi descritti nel Paragrafo 12.2.4 possono gestire correttamente tali operazioni di I/O?

- 12.25. Inizializzando un processo, il gestore della memoria virtuale copia il codice del processo, dal file in cui si trova nello spazio di swap riservato al processo. Dallo spazio di swap, le pagine di codice vengono caricate in memoria quando necessario. Spiegare i vantaggi di questa soluzione. Perché non caricare le pagine di codice direttamente dal file quando necessario? Alcune pagine di codice possono non essere utilizzate durante un'esecuzione, di conseguenza è ridondante copiarle nello spazio di swap. Per evitare copie ridondanti, alcuni gestori della memoria virtuale copiano una pagina di codice nello spazio di swap quando è usata per la prima volta. Discutiamo i vantaggi e gli svantaggi dell'ottimizzazione.
- 12.26. La prestazione della memoria virtuale è determinata dalla combinazione di tre fattori - la velocità di CPU, la dimensione della memoria e il valore massimo di throughput del dispositivo di paginazione. Possibili cause di scarsa o alta efficienza della CPU e del disco di paginazione possono essere:

|                      | <b>Alto utilizzo</b>                             | <b>Scarso utilizzo</b>                             |
|----------------------|--|--|
| CPU                  | I processi sono CPU-bound, oppure la CPU è lenta | Pochi processi sono CPU-bound o thrashing presente |
| Disco di paginazione | Thrashing presente o il disco è lento            | Memoria sovrardimensionata per ogni processo       |

La prestazione della memoria virtuale può migliorare se vengono effettuati uno o più dei seguenti cambiamenti: la CPU viene sostituita da una CPU più veloce, il disco di paginazione viene sostituito da un disco più veloce, la memoria viene incrementata, oppure è incrementato il grado di multiprogrammazione. In ognuna delle situazioni seguenti, quale dei seguenti cambiamenti suggeriresti per migliorare le prestazioni del sistema?

- a. Scarsa efficienza della CPU, scarsa efficienza del disco.
- b. Scarsa efficienza della CPU, alta efficienza del disco.
- c. Alta efficienza della CPU, scarsa efficienza del disco.
- d. Alta efficienza della CPU, alta efficienza del disco.

## Laboratorio: simulazione di un gestore della memoria virtuale

Un gestore della memoria virtuale usa la soluzione a due-thread mostrata in [Figura 12.19](#), dove il thread chiamato *gestore dei frame liberi* tenta di tenere un numero sufficiente di page frame liberi in tutti gli istanti e il thread chiamato *gestore di page I/O* esegue le operazioni di page-out sui frame dirty. Il gestore della memoria virtuale usa l'*algoritmo di clock a due lancette* discusso nell'Esempio 12.8 e illustrato in [Figura 12.20](#). Esegue la sostituzione delle pagine su base *globale*.

Deve essere simulato il funzionamento di questo gestore della memoria virtuale. La simulazione è controllata dai comandi in un file di input, in cui ogni comando ha il formato `<azione> <parametri>`. I dettagli delle azioni sono i seguenti:

| Nome Azione         | Parametri e descrizione   |
|---------------------|---|
| Dimensione_Memoria  | Numero di frame (interno)   |
| Soglia_Minima       | Minimo numero di frame liberi (interno)   |
| Soglia_Massima      | Massimo numero di frame liberi (interno)  |
| Distanza            | Distanza tra le lancette dei clock, in termini di numero di frame (interno)   |
| #processi           | Numero di processi (interno). Gli id di processo sono $P_0, P_1, \dots$   |
| Dimensione_Processo | L'id di processo, numero di pagine (entrambi sono interi)   |
| Read                | Id di processo, numero di pagina: il processo indicato legge la pagina indicata (entrambi sono interi)                    |
| Modifica            | Id di processo, numero di pagina: il processo indicato modifica la pagina indicata (entrambi sono interi)                 |
| Page_table          | Nessun parametro. Il simulatore mostra le tabelle delle pagine dei processi   |
| IO_list             | Nessun parametro. Il simulatore mostra la lista dei page frame sui quali devono essere eseguite le operazioni di page-out |
| Hit_ratio           | Il simulatore mostra l'hit ratio per i processi   |
| Azzera_contatori    | Il simulatore azzera i contatori usati per il calcolo degli hit ratio   |

Sviluppare un simulatore del gestore della memoria virtuale. Il simulatore deve tenere le tabelle delle pagine e gli spazi di swap dei processi. Deve, inoltre, tenere una lista di frame su cui dovranno essere eseguite le operazioni di page-out. Il gestore dei frame liberi mette i numeri dei frame in questa lista. Il gestore delle pagine di I/O esegue le operazioni di page-out sui frame nell'ordine opportuno; informa il gestore dei frame liberi quando l'operazione di page-out di un frame è stata completata.

## Problemi avanzati

- 12.1. Supponiamo che il sistema di paginazione utilizzato dal sistema operativo assegna un numero fissato di frame (blocchi di memoria) da 512 byte a ciascun processo. Prediamo in considerazione la seguente porzione di codice:

```

#define N 256
int a[2N], c[3][N];
compute(){
    unsigned i, j;
    for (i=0; i<3; i++)
        for (j=0; j<N/2; j++)
            c[i][j] = a[4*j] + c[i][j+N/2];
}

```

Si consideri che ogni int occupi 4 byte e che lo stack riempia una pagina inizialmente caricata in memoria.

**a** Determinare la stringa dei riferimenti.

**b** Considerando di avere a disposizione un numero di 3 frame dati per processo, mostrare la tabella delle pagine con algoritmo di sostituzione delle pagine del tipo CLOCK e LRU.

12.2. In un sistema con 6 frame e a partire dalla seguente tabella di pagine:

| Pagina | Tempo di caricamento | Tempo ultimo di riferimento | Bit modifica | Bit riferimento |
|--------|----------------------|-----------------------------|--------------|-----------------|
| 0      | 104                  | 260                         | 0            | 0               |
| 1      | —                    | —                           | 0            | 0               |
| 2      | 20                   | 270                         | 1            | 1               |
| 3      | —                    | —                           | 0            | 0               |
| 4      | —                    | —                           | 0            | 0               |
| 5      | 302                  | 303                         | 1            | 1               |
| 6      | —                    | —                           | 0            | 0               |
| 1      | 45                   | 305                         | 1            | 1               |
| 3      | 203                  | 260                         | 0            | 1               |
| 4      | 79                   | 250                         | 0            | 0               |

Determinare per la stringa di riferimento:

3 2 1 0 1 2 0 3 7 8 5 7 6 3 2 4 3 4

le pagine che vengono sostituite per gli algoritmi (a) FIFO (b) LRU e (c) CLOCK, supponendo che l'ultima pagina cambiata con questo algoritmo sia stata la quarta e le tabelle di pagine relative.

12.3. In un sistema che usa paginazione, l'accesso al TLB richiede 150 ns, mentre l'accesso alla memoria richiede 400 ns. a) Quando si verifica un page fault, si perdono 8ms per caricare la pagina che si sta cercando in memoria. Se il page fault rate è il 2%, il TLB hit è il 70%, indicare l'EAT ai dati. b) Su supponga ora che il sistema implementi la memoria virtuale e che, quando la ricerca nel TLB ha successo, la pagina riferita si trovi effettivamente in RAM, con il TLB che ha ancora un miss ratio del 10%. Si sa che il 4% degli indirizzi logici generati provoca un page fault, e nel 50% dei casi di page fault all'area di swap (o viceversa) è di 1 millisecondo. Indicare l'EAT ai dati.

## Note bibliografiche

Randell (1969) è autore di un primo scritto sulla motivazione dei sistemi a memoria virtuale. Ghanem (1975) ha discusso il partizionamento della memoria nei sistemi a memoria virtuale per la multiprogrammazione. Denning (1970) è autore di un articolo sulla memoria virtuale. Hatfield (1971) discute gli aspetti della prestazione dei programmi nei sistemi a memoria virtuale.

Belady (1966) ha descritto l'anomalia che porta il suo nome. Mattson et al. (1970) hanno descritto la proprietà dello stack degli algoritmi di sostituzione di pagina. Denning (1968a, 1968b) ha trattato il thrashing e il modello a working set. Denning (1980) ha discusso i working set. Smith (1978) è autore di una bibliografia sulla paginazione e argomenti collegati. Wilson et al. (1995) hanno trattato l'allocazione di memoria negli ambienti a memoria virtuale. Johnstone e Wilson (1998) si sono occupati del problema della frammentazione della memoria.

Chang e Mergen (1988) hanno descritto la tabella delle pagine invertita, mentre

Tanenbaum (2001) ha trattato le tabelle delle pagine a due livelli usate negli Intel 30386. Jacob e Mudge (1998) hanno confrontato le caratteristiche della memoria virtuale nelle architetture MIPS, Pentium e PowerPC. Swanson et al. (1998) e Navarro et al. (2002) hanno descritto le superpagine.

Car e Hennessy (1981) hanno descritto l'algoritmo di clock. Bach (1986) e Vahalia (1996) hanno descritto la memoria virtuale di Unix, Beck et al. (2002), Gorman (2004), Bovet e Cesati (2005), e Love (2005) hanno trattato la memoria virtuale di Linux. Mauro e McDougall (2006) hanno trattato la memoria virtuale in Solaris, mentre Russinovich e Solomon (2005) hanno trattato la memoria virtuale di Windows.

Organick (1972) ha descritto la memoria virtuale in MULTICS.

1. Aho, A.V., P.J. Denning, and J. D. Ullman (1971): "Principles of optimal page replacement," *Journal of ACM*, **18** (1), 80-93.
2. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, 3rd ed., Pearson Education, New York.
4. Belady, L.A. (1966): "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, **5** (2), 78-101.
5. Bensoussen, A., C.T. Clingen, and R.C. Daley (1972): "The MULTICS virtual memory - concepts and design," *Communications of the ACM*, **15** (5), 308-318.
6. Bryant, P. (1975): "Predicting working set sizes," *IBM Journal of R and D*, **19** (5), 221-229.
7. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, Calif.
8. Carr, W. R., and J. L. Hennessy (1981): "WSClock - a simple and effective algorithm for virtual memory management," *Proceedings of the ACM Symposium on Operating Systems Principles*, 87-95.
9. Chang, A., and M. Mergen (1988): "801 storage: architecture and programming," *ACM Transactions on Computer Systems*, **6**, 28-50.
10. Daley, R.C., and J.B. Dennis (1968): "Virtual memory, processes and sharing in MULTICS," *Communications of the ACM*, **11** (5), 305-322.
11. Denning, P.J. (1968a): "The working set model for program behavior," *Communications of the ACM*, **11** (5), 323-333.
12. Denning, P.J. (1968b): "Thrashing : Its causes and prevention," *Proceedings of AFIPS FJCC*, **33**, 915-922.
13. Denning, P.J. (1970): "Virtual Memory," *Computing Surveys*, **2** (3), 153-189.
14. Denning, P.J. (1980): "Working sets past and present," *IEEE Transactions on Software Engineering*, **6** (1), 64-84.
15. Ghanem, M.Z. (1975): "Study of memory partitioning for multiprogramming systems with virtual memory," *IBM Journal of R and D*, **19**, 451-457.
16. Gorman, M. (2004): *Understanding the Linux Virtual Memory Manager*, Prentice Hall, Englewood Cliffs, N.J.
17. Guertin, R.L. (1972): "Programming in a paging environment," *Datamation*, **18** (2), 48-55.
18. Hatfield, D.J., and J. Gerald (1971): "Program restructuring for virtual memory," *IBM Systems Journal*, **10** (3), 169-192.
19. Jacob, B., and T. Mudge (1998): "Virtual memory in contemporary microprocessors," *IEEE Micro Magazine*, **18**, 60-75.
20. Johnstone, M.S., and P.R. Wilson (1998): "The memory fragmentation problem: solved?," *Proceedings of the First International Symposium on Memory Management*, 26-36.
21. Love, R. (2005): *Linux Kernel Development*, 2nd Novell Press.
22. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice-Hall, Englewood Cliffs, N.J.
23. Mattson, R.L., J. Gecsei, D. R. Slutz, and I.L. Traiger (1970): "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, **9** (2), 78-117.
24. Navarro, J., S. Iyer, P. Druschel, and A. Cox (2002): "Practical, transparent operating system support for superpages," *ACM SIGOPS Operating Systems Review*, **36**, issue SI, 89-104.

25. Organick, E.I. (1972): *The MULTICS System*, MIT Press, Cambridge, Mass.
26. Randell, B. (1969): "A note on storage fragmentation and program segmentation," *Communications of the ACM*, **12** (7), 365-369.
27. Rosell, J.R., and J.P. Dupuy (1973): "The design, implementation and evaluation of a working set dispatcher," *Communications of the ACM*, **16**, 247-253.
28. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
29. Smith, A.J. (1978): "Bibliography on paging and related topics," *Operating Systems Review*, **12** (4), 39-56.
30. Swanson, M.,L. Stoller, and J. Carter (1998): "Increasing TLB reach using superpages backed by shadow memory," *Proceedings of the 25th International Symposium on Computer Architecture*, 204-213.
31. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
32. Vahalia, U. (1996): *Unix Internals - The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.
33. Wilson, P.R., M.S. Johnstone, M. Neely and D. Boles (1995): "Dynamic storage allocation: a survey and critical review," *Proceedings of the International Workshop on Memory Management*, 1-116.

---

## PARTE 4

# File system e gestione dell'I/O

---

Gli utenti di un computer richiedono convenienza ed efficienza nella creazione e nella manipolazione dei file e nella condivisione dei file con altri utenti del sistema. Inoltre, richiedono che il file system implementi caratteristiche di protezione, sicurezza e affidabilità in modo che i propri file non siano soggetti ad accessi illegali o interferenze da parte degli utenti, o siano danneggiati da malfunzionamenti del sistema. Da un altro punto di vista, un amministratore di sistema si aspetta che un file system assicuri un uso efficiente dei dispositivi di I/O e contribuisca alle alte prestazioni del sistema.

Il file system utilizza una gerarchia di livelli e organizzazioni per soddisfare questi differenti requisiti. Il *livello logico* è adottato per fornire le caratteristiche desiderate dagli utenti. A tale livello, un file è un'entità posseduta da qualche utente, condivisa da un gruppo di utenti e memorizzata in modo efficiente per un certo periodo di tempo. L'organizzazione logica implementa il livello logico. Consiste di differenti tipi file ed operazioni su file, *strutture di directory* e configurazioni/azioni utilizzate per la condivisione e la protezione dei file ed organizzazioni per il funzionamento efficiente del file system.

Il *livello fisico* è adottato per assicurare l'accesso veloce ai dati, buone prestazioni delle operazioni sui file e buone prestazioni dei dispositivi di I/O. A tale livello, un file è una collezione di dati, cui bisogna accedere velocemente, memorizzata su un dispositivo di I/O e che deve essere utilizzata in maniera efficiente. L'organizzazione fisica consiste di procedure che usano *buffer* e *cache* per implementare il livello fisico.

I livelli e le organizzazioni logiche e fisiche del file system sono discusse in capitoli separati. Il terzo capitolo di questa parte tratta la protezione e le misure di sicurezza adottate in un SO.

### Capitolo 13 - File system

Questo capitolo discute i file ed il file system dal punto di vista del programmatore. Descrive le fondamentali *organizzazioni dei file*, la *struttura delle directory*, le operazioni su file e directory e la *semantica della condivisione dei file*, che specifica il modo in cui i risultati delle modifiche ai file effettuate da processi concorrenti sono visibili ad ognuno di essi. Vengono, pertanto, affrontate le problematiche che compromettono l'affidabilità di un file system. Inoltre, sono descritti la fault tolerance implementata mediante *azioni atomiche* ed il ripristino mediante *backup*.

Questo capitolo affronta anche il ruolo del *file control block* come interfaccia tra le organizzazioni logica e fisica usate in un file system. Infine, viene affrontato il suo utilizzo nell'implementazione delle operazioni sui file e nella semantica della condivisione dei file.

#### Linee guida per la Parte 4



Schema che mostra l'ordine in cui i capitoli di questa parte dovrebbero essere affrontati in un corso.

### **Capitolo 14 - Implementazione delle operazioni su file**

Questo capitolo discute l'organizzazione fisica utilizzata nel file system. Inizia con una panoramica dei dispositivi di I/O e delle loro caratteristiche e discute le differenti *configurazioni RAID* che forniscono alta affidabilità, accesso veloce ed alti tassi di traferimento dati. Successivamente, vengono discusse le tecniche utilizzate per implementare l'I/O a livello di dispositivo, tra le quali l'uso dei *buffer* e delle *cache* per velocizzare le operazioni di I/O e l'uso delle politiche di *scheduling del disco* per migliorare il throughput dei dischi.

### **Capitolo 15 - Sicurezza e protezione**

Le misure di sicurezza e protezione assicurano che solo gli utenti autorizzati possano accedere ad un file. Questo capitolo discute diverse tipologie di minacce alla sicurezza e alla protezione di un sistema operativo, le misure utilizzate per impedire queste minacce ed il ruolo delle tecniche di *crittografia* nell'implementazione di queste misure.

---

# CAPITOLO 13

## File system

---

### Obiettivi di apprendimento

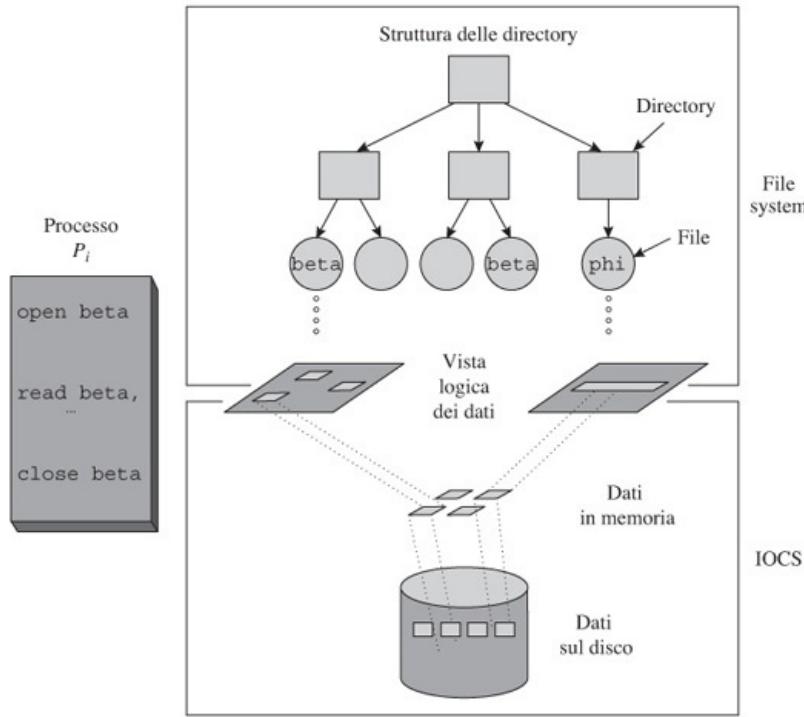
- File ed operazioni su file
- Organizzazione dei file e metodi di accesso
- Organizzazione delle directory
- Protezione dei file
- Allocazione dello spazio su disco
- Elaborazione sui file
- Affidabilità e prestazioni dei file system
- JFS e VFS
- File system dei sistemi operativi: Unix, Linux, Solaris e Windows

Gli utenti di un computer memorizzano programmi e dati in file per un uso conveniente ed efficace e per la condivisione con altri utenti del sistema. Gli utenti si attendono che il file system possa fornire caratteristiche di protezione e sicurezza, in modo tale che i loro file non siano soggetti ad accessi non autorizzati e non siano alterati da altri utenti o addirittura danneggiati da fault del sistema. Gli obiettivi principali che si pone un file system sono:

- accesso conveniente e veloce ai file;
- memorizzazione affidabile dei file;
- condivisione dei file con i collaboratori.

Le risorse utilizzate per memorizzare e accedere ai file sono i dispositivi di I/O. Il SO garantisce sia prestazioni efficienti delle attività di elaborazione dei file nei processi sia l'uso efficiente dei dispositivi di I/O.

I sistemi operativi gestiscono i file mediante due componenti chiamati *file system* e *sistema di controllo input-output* (*input-output control system* - IOCS), vedi [Figura 13.1](#), e ciò per separare il livello file dalle problematiche relative alla memorizzazione e all'accesso efficiente ai dati. Di conseguenza, un file system fornisce gli strumenti per la creazione e la manipolazione dei file, per garantire l'affidabilità dei file quando si verificano guasti come l'interruzione della corrente o malfunzionamenti dei dispositivi di I/O, e per specificare come i file devono essere condivisi tra gli utenti. Il IOCS fornisce accesso ai dati memorizzati nei dispositivi di I/O e buone prestazioni dei dispositivi di I/O. Per soddisfare le esigenze indicate precedentemente, il file system utilizza una gerarchia in cui si discerne tra vista logica, utile a soddisfare le esigenze degli utenti, e la corrispondente implementazione fisica, in cui sono presenti i file e sono stabilite le operazioni possibili su di essi, le strutture e le regole di condivisione tra gli utenti. La vista fisica è utilizzata per garantire velocità nell'accesso ai dati, buone prestazioni riguardo alle operazioni compiute sui file e buone prestazioni dei dispositivi di I/O. In tal modo, un file è un insieme di dati memorizzati su un dispositivo di I/O a cui si deve accedere velocemente ed il cui utilizzo deve risultare efficiente. Al fine di implementare la vista fisica è necessario utilizzare anche buffer e memorie cache.



**Figura 13.1** File system e IOCS.

Questo capitolo affronta la progettazione dei file system. Dopo aver discusso le basi dell'organizzazione dei file, della struttura delle directory e della gestione dello spazio sul disco, descriviamo le *semantiche della condivisione dei file* che governano la condivisione concorrente dei file e l'*affidabilità del file system*. L'implementazione delle operazioni sui file mediante il IOCS è discussa nel [Capitolo 14](#).

### 13.1 Elaborazione dei file: una panoramica

Il termine *elaborazione dei file* è utilizzato per descrivere la sequenza generale delle operazioni di apertura del file, lettura dei dati dal file o scrittura dei dati al suo interno e chiusura del file. La [Figura 13.1](#) mostra l'organizzazione attraverso la quale un SO implementa le attività di elaborazione dei file dei processi. Ogni *directory* contiene elementi che descrivono alcuni file. L'elemento di una directory corrispondente a un file indica il nome del suo proprietario, la posizione nel disco, il modo in cui i dati sono organizzati e quali utenti possono accedere a esso e in che maniera.

Nella parte sinistra della [Figura 13.1](#) è mostrato il codice di un processo  $P_i$ . Quando il file system apre un file per elaborarlo, localizza il file nella *struttura delle directory*, cioè in un'organizzazione di molte directory. In [Figura 13.1](#) ci sono due file chiamati *beta* memorizzati in directory differenti. Quando il processo  $P_i$  apre *beta*, il modo in cui richiama *beta*, la struttura delle directory e le identità dell'utente che ha avviato il processo  $P_i$ , insieme, determineranno a quale dei due file si vuole accedere.

Un file system fornisce diversi *tipi di file* (Paragrafo 13.2). Ogni tipo di file fornisce la propria vista astratta dei dati in un file - che chiamiamo *vista logica* dei dati. La [Figura 13.1](#) mostra che il file *beta* aperto dal processo  $P_i$  ha una vista logica orientata al *record*, mentre il file *phi* ha una vista logica orientata allo *stream di byte* in cui non esistono record distinti.

Il IOCS organizza i dati di un file in un dispositivo di I/O relativamente al tipo di file. Questa è denominata *vista fisica* dei dati del file. La mappatura tra la vista logica dei dati del file e la sua vista fisica è effettuata dal IOCS. Il sistema IOCS, inoltre, fornisce un'organizzazione che velocizza l'attività di elaborazione di un file - mantiene alcuni dati del file in aree di memoria organizzate come *buffer*, una *file cache* o una *disk cache*. Quando un processo esegue un'operazione di *read* per ottenere dei dati da un file, il sistema IOCS prende i dati da un *buffer* o da una *cache* se sono presenti. In questo modo,

il processo non deve aspettare finché i dati sono letti dal dispositivo di I/O sul quale il file è memorizzato. In modo analogo, quando un processo esegue un'operazione di write su di un file, il IOCS copia i dati scritti in un buffer o in una cache. Le effettive operazioni di I/O per leggere i dati da un dispositivo di I/O in un buffer o in una cache, o per scriverli da questa posizione in un dispositivo di I/O, sono eseguiti in background dal IOCS.

### 13.1.1 File system e IOCS

Un file system vede un file come un insieme di dati di cui è proprietario un utente, che può essere *condiviso* da un insieme di utenti autorizzati e che deve essere *memorizzato in modo affidabile* per un esteso periodo di tempo. Un file system concede agli utenti la libertà di dare un nome ai propri file per sancire la proprietà, così che un utente possa dare il nome desiderato a un file senza preoccuparsi se esso vada in conflitto con i nomi dei file di altri utenti; inoltre, offre la privacy fornendo la protezione dalle interferenze dovute ad altri utenti. Il IOCS, d'altra parte, vede un file come contenitore di dati cui è necessario *accedere velocemente* e che sono memorizzati su un dispositivo di I/O che deve essere *utilizzato in maniera efficiente*.

La [Tabella 13.1](#) elenca i mezzi forniti dal file system e dal IOCS. Il file system fornisce le strutture delle directory che consentono agli utenti di organizzare i propri dati in gruppi logici di file, per esempio, un gruppo di file per ogni attività professionale. Il file system fornisce protezione contro gli accessi non autorizzati ai file e assicura il corretto funzionamento quando più processi accedono e aggiornano un file in maniera concorrente. Inoltre, garantisce che i dati vengano memorizzati in maniera affidabile, ovvero i dati non siano persi quando si verifica un crash del sistema. Le caratteristiche del sistema IOCS sono quelle descritte in precedenza.

---

#### File system

- Strutture delle directory per il raggruppamento conveniente dei file
- Protezione dei file contro gli accessi illeciti
- Semantica condivisione dei file per gli accessi concorrenti a un file
- Memorizzazione affidabile dei file

#### Sistema di controllo input-output (IOCS)

- Funzionamento efficiente dei dispositivi di I/O
  - Accesso efficiente ai dati in un file
- 

**Tabella 13.1** Strumenti forniti dal file system e dal sistema di controllo input-output.

Il file system e il sistema IOCS costituiscono una gerarchia in cui ognuno adotta una politica differente e fornisce meccanismi per implementare tali comportamenti. Nel linguaggio del [Paragrafo 1.1](#), il IOCS e il file system forniscono differenti astrazioni che portano alla seguente divisione delle funzioni:

- il file system fornisce un'interfaccia attraverso la quale un processo può eseguire le operazioni di apertura, lettura/scrittura e chiusura dei file. I moduli che implementano le regole di comportamento gestiscono la politica relativa alla protezione e la condivisione dei file durante le operazioni di apertura e lettura/scrittura. I moduli che realizzano il meccanismo forniscono supporto all'implementazione delle operazioni di apertura e chiusura accedendo alle directory. Inoltre, passano al IOCS le richieste di lettura/scrittura per un file;
- i moduli del IOCS che realizzano tali regole di comportamento assicurano il funzionamento efficiente dei dispositivi di I/O e l'elaborazione efficiente dei file in ogni processo mediante i moduli che implementano il meccanismo. I moduli che realizzano il meccanismo nel IOCS, a loro volta, invocano il kernel mediante chiamate di sistema per avviare le operazioni di I/O.

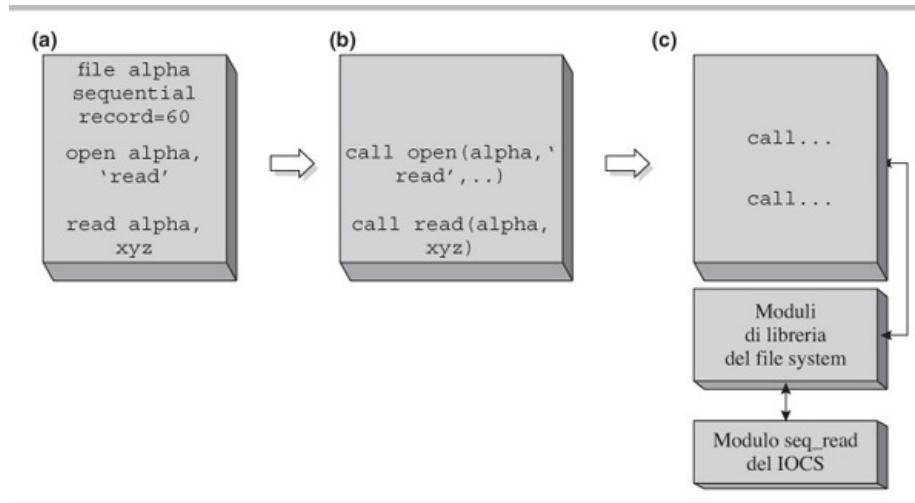
#### Dati e metadati

Un file system consiste di due tipi di dati - i dati contenuti nei file e i dati usati per accedere ai file. Chiamiamo i dati contenuti nei file *dati del file* o semplicemente *dati*. I dati usati per accedere ai file sono chiamati *dati di controllo* o *metadati*. Nella vista logica mostrata in [Figura 13.1](#), i dati contenuti nella struttura delle directory sono metadati. Come discusso successivamente in questo capitolo e nel [Capitolo 14](#), altri metadati giocano un ruolo nell'implementazione delle operazioni sui file.

### 13.1.2 Elaborazione dei file in un programma

A livello di linguaggio di programmazione, un file è un oggetto che possiede *attributi* che descrivono l'organizzazione dei suoi dati e il metodo di accesso ai dati. Un programma contiene un'istruzione di dichiarazione per un file, che specifica i valori dei suoi attributi e le istruzioni per aprirlo, per eseguire operazioni di lettura/scrittura, e chiuderlo (le chiamiamo istruzioni per l'elaborazione dei file). Durante l'esecuzione del programma, l'elaborazione dei file è di fatto implementata dai moduli di libreria del file system e del IOCS.

La [Figura 13.2](#) illustra come l'elaborazione dei file viene effettivamente implementata. Il programma di [Figura 13.2\(a\)](#) dichiara alpha come un file ad accesso sequenziale che contiene record di dimensione 60 byte (Paragrafo 13.2 per una descrizione dei record in un file). Inoltre, contiene le istruzioni per aprire alpha e leggere un record da esso. Il compilatore del linguaggio di programmazione elabora l'istruzione di dichiarazione del file nel programma e determina gli attributi del file. A questo punto sostituisce le istruzioni open, close, read e write con chiamate ai moduli di libreria open, close, read e write del file system e passa gli attributi del file come parametri alla chiamata open [[Figura 13.2\(b\)](#)]. I moduli del file system richiamano i moduli del IOCS per eseguire effettivamente le operazioni di I/O. Il linker linka i moduli di libreria del file system e i moduli del IOCS richiamati per produrre il programma mostrato in [Figura 13.2\(c\)](#) (Paragrafo 11.3.2 per una descrizione della funzione di linking). Un processo creato per l'esecuzione di questo programma, invoca i moduli di libreria del file system per eseguire le operazioni open e read sul file; questi moduli implementano le suddette operazioni utilizzando gli appropriati moduli di libreria del IOCS.



**Figura 13.2** Implementazione dell'attività di elaborazione di un file: (a) programma contenente le istruzioni di dichiarazione del file; (b) programma compilato che mostra le chiamate ai moduli del file system; (c) processo che invoca il file system e i moduli IOCS durante il funzionamento.

## 13.2 File e operazioni sui file

### Tipi di file

Un file system contiene e organizza diversi tipi di file, per esempio, i file dati, programmi eseguibili, moduli oggetto, informazioni testuali, documenti, fogli di calcolo, foto e file video. Ognuno di questi tipi di file ha il proprio formato per memorizzare i dati. Questi tipi di file possono essere raggruppati in due classi:

- file strutturati;
- file orientati allo stream di byte.

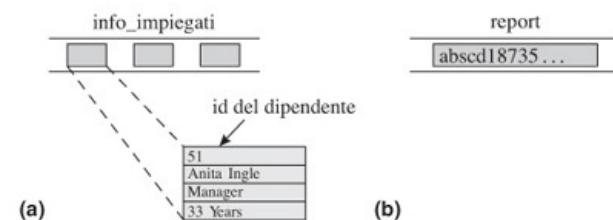
Un *file strutturato* è una collezione di record, dove un record è un'unità significativa per l'elaborazione dei dati. Un *record* è una collezione di campi e un *campo* contiene un singolo elemento di dati. Si assume che ogni record in un file contenga un campo *chiave*.

Il valore nel campo chiave di un record è unico nel file; ovvero non esistono due record che contengono la stessa chiave. Molti tipi di file menzionati in precedenza sono file strutturati. I tipi di file utilizzati dai software standard del sistema come i compilatori e i linker hanno una struttura determinata dal progettista del SO, mentre i tipi di file degli utenti dipendono dalle applicazioni o dai programmi che li creano.

Al contrario, un *file orientato allo stream di byte* è “piatto”. Non contiene né record né campi; viene visto come una sequenza di byte dai processi che lo usano. Il prossimo esempio illustra i file strutturati e i file orientati allo stream di byte.

### Esempio 13.1 - File strutturati ed orientati allo stream di byte

La [Figura 13.3\(a\)](#) mostra un file strutturato chiamato `info_impiegati`. Ogni record nel file contiene informazioni riguardanti un dipendente. Un record contiene quattro campi: id del dipendente, nome, nomina ed età. Il campo che contiene l'id del dipendente è il campo chiave. La [Figura 13.3\(b\)](#) mostra un file orientato allo stream di byte `report`.



**Figura 13.3** Vista logica di (a) un file strutturato `info_impiegati`; (b) un file orientato allo stream di byte `report`.

### Attributi di un file

Un attributo di un file è un'importante caratteristica di un file sia per gli utenti sia per il file system. Gli attributi comunemente usati per un file sono: tipo, organizzazione, dimensione, posizione sul disco, informazioni per il controllo degli accessi (che indicano il modo in cui utenti differenti possono accedere al file), nome del proprietario, tempo della creazione e tempo dell'ultimo utilizzo. Il file system memorizza gli attributi di un file nell'elemento della directory. Durante l'attività di elaborazione di un file, il file system usa gli attributi di un file per localizzarlo e assicurare che ogni operazione da effettuare su di esso sia consistente con gli attributi. Al termine dell'attività di elaborazione del file, il file system memorizza i valori modificati degli attributi nell'elemento della directory relativa al file.

### Operazioni su file

La [Tabella 13.2](#) descrive le operazioni eseguite sui file. Come menzionato precedentemente, le operazioni come apertura, chiusura, ridefinizione e cancellazione sono eseguite dai moduli del file system. L'accesso effettivo ai file, ovvero, lettura e scrittura dei record, viene implementato dai moduli del IOCS.

| Operazione                   | Descrizione  |
|------------------------------|--|
| Apertura di un file          | Il file system recupera l'elemento della directory corrispondente al file e controlla se l'utente il cui processo sta cercando di aprire il file ha i privilegi di accesso necessari per il file. Successivamente, esegue alcune azioni di gestione per avviare l'elaborazione del file. |
| Leggere o scrivere un record | Il file system considera l'organizzazione del file (Paragrafo 13.3) e implementa le operazioni di lettura/scrittura in maniera appropriata.  |
| Chiusura di un file          | L'informazione relativa alla dimensione del file nell'elemento della directory relativo al file viene aggiornata.  |

|                                    |  |
|------------------------------------|--|
| Copia di un file                   | Viene eseguita una copia del file, viene creato un nuovo elemento della directory per la copia e il suo nome, la sua dimensione, la posizione e le informazioni di protezione vengono memorizzate nella voce corrispondente. |
| Cancellazione del file             | L'elemento della directory relativo al file viene cancellato e l'area sul disco occupata viene liberata.   |
| Ridenominazione del file           | Il nuovo nome viene registrato nell'elemento della directory relativo al file.   |
| Specificare i privilegi di accesso | Le informazioni di protezione contenute nell'elemento della directory relativo al file vengono aggiornate.   |

**Tabella 13.2** Operazioni possibili su file.

### 13.3 Fondamenti dell'organizzazione dei file e metodi di accesso

Usiamo il termine “pattern di accesso al record” per descrivere l’ordine in cui un processo accede ai record in un file. I due metodi fondamentali di accesso ai record sono *l’accesso sequenziale*, secondo cui l’accesso ai record avviene nell’ordine in cui si trovano nel file (o nell’ordine inverso) e *l’accesso casuale*, secondo cui si può accedere ai record in qualsiasi ordine. Le azioni di elaborazione dei file in un processo saranno eseguite in maniera efficiente solo se il pattern di accesso ai record da parte del processo può essere implementato in maniera efficiente nel file system. Le caratteristiche di un dispositivo di I/O lo rendono adatto per uno specifico pattern di accesso ai record. Per esempio, un drive a nastro può accedere solo al record che si trova immediatamente prima o dopo la posizione corrente rispetto alla testina di lettura/scrittura; quindi, è adatto solo per l’accesso sequenziale ai record. Un disco può accedere direttamente, dato il suo indirizzo, a qualunque record; quindi, può implementare in maniera efficiente i pattern di accesso ai record sia sequenziale che casuale.

Un’organizzazione dei file è una combinazione di due caratteristiche: un modo di organizzare i record in un file e una procedura per accedervi. L’organizzazione di un file è progettata per sfruttare le caratteristiche di un dispositivo di I/O in modo da fornire un accesso efficiente ai record per uno specifico pattern di accesso ai record. Un file system supporta diverse organizzazioni dei file in modo che un processo possa adottare quello che meglio si adatta ai propri requisiti di elaborazione e al dispositivo di I/O in uso. Questo paragrafo descrive tre organizzazioni fondamentali dei file: l’organizzazione dei file sequenziale, diretta e indicizzata. Altre organizzazioni dei file utilizzate nella pratica sono varianti di queste fondamentali oppure sono organizzazioni adatte a uno specifico scopo che sfruttano dispositivi di I/O utilizzati meno comunemente.

Gli accessi ai file caratterizzati da una specifica organizzazione dei file sono implementati da un modulo del IOCS chiamato *metodo di accesso*. Un metodo di accesso è un modulo che implementa una politica del IOCS. Nella fase di compilazione di un programma, il compilatore ricava l’organizzazione dei file che governa un file a partire dall’istruzione di dichiarazione del file (o dalle regole per il caso di default, se il programma non contiene un’istruzione di dichiarazione del file) e identifica il metodo di accesso corretto per richiamare le operazioni sul file. Descriviamo le funzioni dei metodi di accesso dopo aver discusso le organizzazioni fondamentali dei file.

#### 13.3.1 Organizzazione sequenziale dei file

Nell’organizzazione sequenziale dei file, i record sono memorizzati in ordine crescente o decrescente in base al campo chiave; il pattern di accesso ai record di un’applicazione si comporta di conseguenza. Dunque, l’organizzazione sequenziale dei file supporta due tipi di operazioni: legge il record successivo (o precedente) e salta il record precedente (o successivo). Un file ad accesso sequenziale viene utilizzato in un’applicazione se i suoi dati possono essere pre-ordinati convenientemente in ordine crescente o decrescente. L’organizzazione dei file sequenziale viene utilizzata anche per i file orientati allo stream di byte.

### 13.3.2 Organizzazione diretta dei file

L'organizzazione diretta dei file fornisce convenienza ed efficienza per l'elaborazione quando si accede ai record in ordine casuale. Per accedere a un record, un comando di lettura/scrittura deve specificare il valore nel campo chiave; ci riferiamo a tali file come *file ad accesso diretto*. Un file ad accesso diretto viene implementato come segue: quando un processo fornisce il valore chiave di un record cui accedere, il modulo relativo al metodo di accesso per i file ad accesso diretto applica una trasformazione al valore chiave che genera l'indirizzo del record nel supporto di memorizzazione. Se il file è memorizzato su un disco rigido, la trasformazione genera un indirizzo (*num\_traccia*, *num\_record*). Le testine del disco vengono posizionate sulla traccia *num\_traccia* prima che venga eseguito un comando di lettura o scrittura sul record *num\_record*.

Si consideri un file contenente le informazioni dei dipendenti organizzato come file ad accesso diretto. Sia  $p$  il numero di record scritti su una traccia del disco. Assumendo che i numeri degli impiegati e i numeri di traccia e di record del file partano da 1, l'indirizzo del record per il dipendente numero  $n$  è (*numero di traccia* ( $t_n$ ), *numero di record* ( $r_n$ )) dove:

$$t_n = \left\lceil \frac{n}{p} \right\rceil \quad (13.1)$$

$$r_n = n - (t_n - 1) \times p \quad (13.2)$$

e  $\lceil \dots \rceil$  indica un valore intero arrotondato per eccesso.

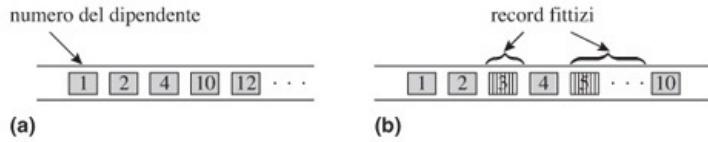
L'organizzazione diretta dei file fornisce efficienza nell'accesso quando i record sono elaborati in maniera casuale. Tuttavia, presenta tre svantaggi se confrontato con l'organizzazione sequenziale dei file.

- Il calcolo dell'indirizzo del record "consuma" tempo di CPU.
- I dischi possono memorizzare molti più dati lungo la traccia più esterna rispetto a quella più interna. Tuttavia, l'organizzazione diretta dei file memorizza una eguale quantità di dati lungo ogni traccia. Dunque una parte della capacità di memorizzazione viene sprecata.
- La formula per il calcolo dell'indirizzo (13.1) e (13.2) funziona correttamente solo se esiste un record per ogni possibile valore della chiave, per cui devono esistere record fintizi per le chiavi che non sono in uso. Questo requisito porta a uno scarso utilizzo del dispositivo di I/O.

Possiamo dedurne che l'elaborazione sequenziale dei record in un file ad accesso diretto è meno efficiente rispetto all'elaborazione dei record in un file ad accesso sequenziale. Un altro problema pratico è l'assunzione esplicita delle caratteristiche di un dispositivo di I/O nelle formule 13.1 e 13.2 per il calcolo dell'indirizzo, la qual cosa rende l'organizzazione dei file dipendente dal dispositivo. Infatti, riscrivere il file su un altro dispositivo con differenti caratteristiche, per esempio, con capacità per traccia differente, implicherebbe la modifica delle formule per il calcolo dell'indirizzo. Questo requisito limita la portabilità dei programmi.

#### Esempio 13.2 - File ad accesso sequenziale e diretto

La [Figura 13.4](#) mostra l'organizzazione dei record dei dipendenti, utilizzando l'organizzazione dei file sequenziale e diretta. I dipendenti con numero 3, 5-9 e 11 hanno lasciato l'organizzazione. Tuttavia, il file ad accesso diretto deve contenere un record per ognuno di questi dipendenti per soddisfare le formule 13.1 e 13.2 per il calcolo degli indirizzi. Ciò comporta la necessità di record fintizi nei file ad accesso diretto.



**Figura 13.4** I record nel (a) file sequenziale; (b) file ad accesso diretto.

### 13.3.3 Organizzazione dei file indicizzata

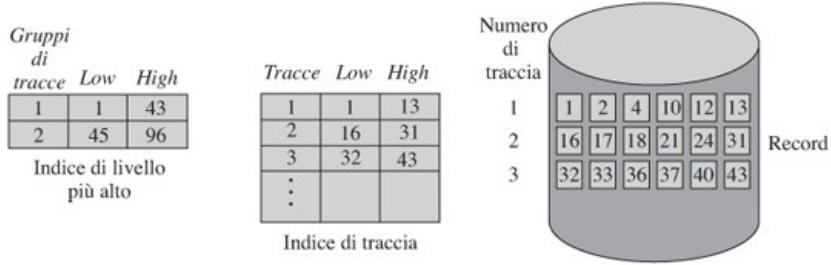
Un *indice* aiuta a determinare la posizione di un record a partire dal valore della sua chiave. In un'organizzazione dei file indicizzata di tipo puro, l'indice di un file contiene un elemento indice nella forma (valore della chiave, indirizzo del disco) per ogni valore della chiave esistente nel file. Per accedere a un record con chiave  $k$ , viene trovato l'elemento indice contenente  $k$  tramite la ricerca per indice, e si utilizza l'indirizzo del disco annotato nell'elemento trovato per accedere al record. Se un indice è inferiore al file, questa organizzazione fornisce un'elevata efficienza di accesso poiché una ricerca per indice è più efficiente rispetto a una ricerca nel file.

L'organizzazione dei file *sequenziale indicizzata* è un'organizzazione ibrida che combina gli elementi delle organizzazioni dei file indicizzata e sequenziale: cerca un indice per identificare una sezione del disco che può contenere il record ed effettua la ricerca sequenziale dei record in questa sezione del disco per trovare il record desiderato. La ricerca ha buon esito se il record è presente nel file; altrimenti avrà esito negativo. Questa organizzazione richiede un indice molto più piccolo rispetto all'indicizzazione pura poiché l'indice contiene elementi solo per alcuni valori della chiave. Inoltre, fornisce una migliore efficienza nell'accesso rispetto all'organizzazione sequenziale dei file garantendo, allo stesso tempo, un uso ugualmente efficiente del dispositivo di I/O.

Per un file di grandi dimensioni l'indice può contenere un gran numero di elementi, per cui il tempo richiesto per la ricerca mediante l'indice può risultare molto lungo. Un indice di livello più alto può essere utilizzato per ridurre il tempo di ricerca. Un elemento nell'indice di livello più alto punta a una sezione dell'indice. Questa sezione dell'indice viene esaminata per trovare la sezione del disco che può contenere un record cercato e in questa sezione del disco si esegue una ricerca sequenziale per il record cercato. Il prossimo esempio illustra questa organizzazione.

#### Esempio 13.3 - Organizzazione dei file sequenziale indicizzata

La [Figura 13.5](#) illustra un file di informazioni riguardanti alcuni dipendenti, organizzato come un file sequenziale indicizzato. Vengono costruiti due indici per facilitare una ricerca veloce. L'indice di traccia indica il più piccolo e il più grande valore posizionati su ogni traccia (vedi i campi chiamati *low* e *high* in [Figura 13.5](#)). L'indice di livello più alto contiene gli elementi per gruppi di tracce contenenti 3 tracce ognuno. Per localizzare il record con una chiave  $k$ , si cerca prima l'indice di più alto livello per localizzare il gruppo di tracce che possono contenere il record cercato. L'indice di traccia per le tracce del gruppo è ora esaminato per trovare la traccia che può contenere il record cercato e si cerca sequenzialmente, nella traccia selezionata, il record con chiave  $k$ . La ricerca termina senza successo se il record non viene trovato nella traccia.



**Figura 13.5** Indice di traccia e indice di livello più alto in un file a indicizzazione sequenziale.

### 13.3.4 Metodi di accesso

Un *metodo di accesso* è un modulo del IOCS che implementa gli accessi a una classe di file che utilizza una specifica organizzazione dei file. La procedura da utilizzare per accedere ai record in un file, o con un ricerca sequenziale o mediante calcolo dell'indirizzo, è determinata dall'organizzazione del file. Il modulo relativo al metodo di accesso utilizza questa procedura per accedere ai record. È possibile anche utilizzare alcune tecniche avanzate nella programmazione del I/O per rendere più efficiente l'accesso al file. Due di queste tecniche sono il *buffering* e il *blocking* dei record.

#### **Buffering dei record**

Il metodo di accesso legge i record di un file di input prima che siano effettivamente richiesti da un processo e li memorizza temporaneamente in aree di memoria chiamate *buffer* finché non vengono richiesti dal processo. Lo scopo del buffering è di ridurre o eliminare l'attesa per il completamento di un'operazione di I/O; il processo deve attendere solo quando il record desiderato non è già presente in un buffer. Le azioni inverse sono eseguite per un file di output. Quando il processo esegue un'operazione di scrittura, i dati da scrivere nel file sono copiati in un buffer e il processo continua la sua esecuzione. I dati sono scritti nel dispositivo di I/O in un momento successivo e il buffer viene rilasciato per essere riutilizzato. Il processo deve attendere solo se non è disponibile un buffer al momento dell'operazione di scrittura.

#### **Blocking dei record**

Il metodo di accesso legge o scrive sempre grandi blocchi di dati, che contengono diversi record di file, da o verso un dispositivo di I/O. Questa caratteristica riduce il numero totale di operazioni di I/O richieste per elaborare un file, migliorando di conseguenza l'efficienza nell'elaborazione di un file da parte di un processo. Il blocking, inoltre, migliora l'utilizzo di un dispositivo di I/O e il throughput di un dispositivo.

Parleremo delle tecniche di buffering e blocking dei record nel [Capitolo 14](#).

## 13.4 Directory

Una directory contiene le informazioni relative a un gruppo di file. Ogni elemento in una directory contiene gli attributi di un file, come il suo tipo, l'organizzazione, la dimensione e le modalità in cui i vari utenti del sistema vi possono accedere. La [Figura 13.6](#) mostra i campi di un tipico elemento di una directory. I campi *Open count* e *Lock* sono usati quando diversi processi aprono un file in maniera concorrente. Finché questo contatore è diverso da zero, il file system mantiene in memoria alcuni metadati riguardanti il file per velocizzare gli accessi ai dati contenuti nel file. Il campo *Lock* viene utilizzato quando un processo richiede l'accesso esclusivo al file. Il campo *Flag* è utilizzato per differenziare i diversi tipi di elementi di una directory. Usiamo il valore "D" in questo campo per indicare che un file è una directory, "L" per indicare che è un link e "M" per indicare che si tratta di un file system. I paragrafi successivi di questo capitolo descriveranno questi usi. Il campo *Misc info* contiene informazioni quali: il proprietario del file, il momento della creazione e l'ultima modifica.

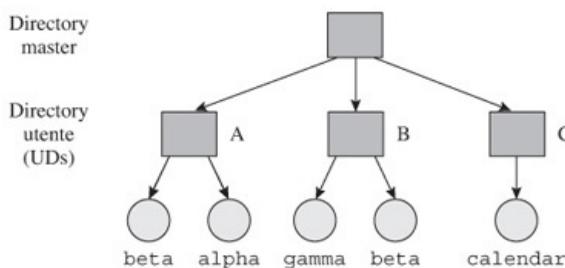
| Nome del file                 | Tipo e dimensione   | Informazioni sulla posizione | Informazioni sulla protezione | Open count | Lock | Flag | Misc info |
|-------------------------------|---|------------------------------|-------------------------------|------------|------|------|-----------|
|                               |   |                              |                               |            |      |      |           |
| <b>Campo</b>                  | <b>Descrizione</b>  |                              |                               |            |      |      |           |
| Nome del file                 | Nome del file. Se questo campo ha dimensione fissa, i nomi lunghi oltre una certa lunghezza saranno troncati.   |                              |                               |            |      |      |           |
| Tipo e dimensione             | Il tipo e la dimensione del file. In molti file system, il tipo di file è implicito nella sua estensione; per esempio, un file con estensione .c è un file che contiene un programma C e un file con estensione .obj è un file oggetto, che spesso è un file strutturato. |                              |                               |            |      |      |           |
| Informazioni sulla posizione  | Informazioni sulla posizione del file sul disco. Queste informazioni sono tipicamente sotto forma di una tabella o di una lista concatenata contenente gli indirizzi dei blocchi sul disco allocati al file.  |                              |                               |            |      |      |           |
| Informazioni sulla protezione | Le informazioni relative agli utenti cui è concesso l'accesso a questo file e in che modo.  |                              |                               |            |      |      |           |
| Open count                    | Numero di processi che attualmente accedono al file.  |                              |                               |            |      |      |           |
| Lock                          | Indica se un processo sta accedendo al file in maniera esclusiva.   |                              |                               |            |      |      |           |
| Flag                          | Informazioni sulla natura del file, quali se il file è una directory, un link o un file system montato.   |                              |                               |            |      |      |           |
| Misc info                     | Informazioni varie come l'id del proprietario, la data e l'ora della creazione, l'ultimo accesso e l'ultima modifica.   |                              |                               |            |      |      |           |

**Figura 13.6** Campi di un tipico elemento di una directory.

Un file system contiene i file appartenenti a diversi utenti. Quindi deve garantire agli utenti due importanti prerogative:

- *libertà nella scelta del nome*: possibilità per un utente di dare un qualunque nome a un file, senza essere vincolato ai nomi dei file scelti dagli altri utenti;
- *condivisione dei file*: possibilità per un utente di accedere ai file creati da altri utenti e possibilità di concedere ad altri utenti il permesso di accedere ai propri file.

Il file system crea diverse directory e le organizza in una *struttura* al fine di fornire libertà nella scelta dei nomi e la condivisione dei file. Includiamo un diagramma schematico per illustrare le strutture di directory, utilizzando la convenzione che una directory è rappresentata da un rettangolo, mentre un file è rappresentato da un cerchio. La [Figura 13.7](#) mostra una semplice struttura di directory contenente due tipi di directory. Una *directory utente* (UD) contiene elementi che descrivono i file appartenenti a un utente. La *directory master* contiene informazioni relativamente alle UD di tutti gli utenti registrati del sistema; ogni elemento nella directory master è una coppia ordinata che consiste di un id utente e di un puntatore a una UD. Nel file system mostrato, gli utenti A e B hanno creato ognuno un proprio file chiamato beta. Questi file corrispondono a elementi contenuti nelle rispettive UD. Descriviamo la struttura della directory mostrata in [Figura 13.7](#) come una struttura di directory a *due livelli*.



**Figura 13.7** Struttura di una directory composta da directory master e utente.

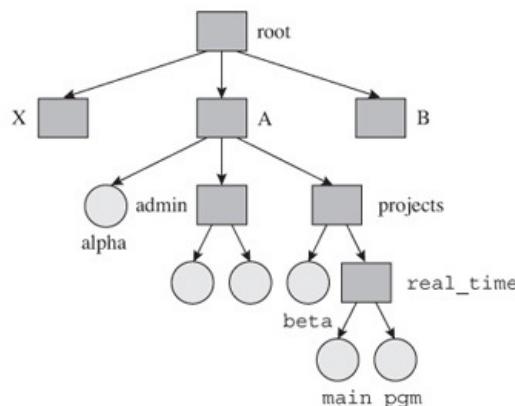
L'uso di UD separate permette la libertà nell'assegnazione di un nome. Quando un

processo creato da un utente A esegue l'istruzione `open (beta, ...)`, il file system effettua una ricerca nella directory master per localizzare la UD di A e cerca `beta` al suo interno. Se invece la chiamata `open (beta, ...)` fosse stata eseguita da qualche processo creato da B, il file system avrebbe dovuto cercare `beta` nella UD di B. Una tale organizzazione garantisce l'accesso al file corretto anche se nel sistema esistono molti file con lo stesso nome. Tuttavia l'utilizzo delle UD ha uno svantaggio: impedisce agli utenti di condividere i loro file con altri utenti. Una sintassi speciale può dover essere fornita per consentire a un utente di utilizzare il file di un altro utente. Per esempio, un processo creato da un utente C può eseguire l'istruzione `open (A → beta, ...)` per aprire il file `beta` di A. Il file system può implementarlo semplicemente utilizzando la UD di A, piuttosto che la UD di C, per localizzare il `beta`. Per implementare la protezione dei file, il file system deve determinare se all'utente C è consentita l'apertura del file `beta` di A. A tale scopo, controlla il campo *Protection info* dell'elemento della directory corrispondente al file `beta`. I dettagli della protezione dei file sono discussi nel Paragrafo 13.6.

### 13.4.1 Alberi di directory

Il file system MULTICS del 1969 conteneva caratteristiche che consentivano all'utente di creare una nuova directory, dargli un nome di sua scelta e creare i file e altre directory al suo interno fino a un qualunque livello. La struttura di directory risultante è un *albero* che chiamiamo *albero delle directory*. Dopo MULTICS, molti file system hanno utilizzato gli alberi di directory.

Un utente può creare un file per contenere dati o può creare una directory. Quando una distinzione tra i due è importante, chiameremo questi file rispettivamente *file dati* e *file directory*, o semplicemente *directory*. Il file system fornisce una directory chiamata *root* che contiene la *directory home* per ogni utente, ovvero una directory che solitamente ha lo stesso nome del nome utente. Un utente organizza le proprie informazioni creando file directory e file dati nella sua directory home, creando file directory e file dati in una directory e così via. Assumiamo che il file system inserisca una "D" nel campo *Flag* dell'elemento di directory relativo a un file se il file in oggetto è una directory. La [Figura 13.8](#) mostra l'albero di directory del file system. La radice di questo albero è la directory *root*, che contiene una directory *home* per ogni utente con lo stesso nome dell'utente. L'utente A ha creato un file chiamato `alpha` e directory chiamate `admin` e `projects`. La directory `projects` contiene una directory `real_time`, che contiene un file `main_pgm`. Dunque l'utente A ha un albero di directory proprio; la sua radice è la sua directory *home*.



**Figura 13.8** Alberi di directory del file system e dell'utente A.

A ogni istante, si dice che un utente si trova "in" una specifica directory, che si chiama *directory corrente*. Quando l'utente vuole aprire un file, il file viene cercato in questa directory. Quando l'utente effettua il login, il SO consente all'utente di operare solo nella sua directory home; la directory home è dunque la directory corrente dell'utente. Un utente può cambiare la sua directory corrente in ogni momento mediante un comando "change directory".

Il nome dato a uno specifico file può non essere unico nel file system, per cui un utente o un processo utilizza un *path* per identificarlo in maniera univoca. Un *path name* è una

sequenza di una o più componenti del percorso separate da uno slash (/), in cui ogni componente è un riferimento a una directory e l'ultima componente è il nome del file.

I path per localizzare un file a partire dalla directory corrente sono detti *path relativi*. I path relativi sono spesso corti e convenienti da usare; tuttavia, possono essere fonte di confusione poiché un file può avere diversi path relativi quando vi si accede a partire da diverse directory correnti. Per esempio, in [Figura 13.8](#), il file alpha ha il path relativo alpha quando vi si accede dalla directory corrente A, mentre ha il path relativo nella forma ../alpha e ../../alpha quando vi si accede, rispettivamente, dalle directory projects e real\_time. Per facilitare l'uso dei path relativi, ogni directory conserva l'informazione sulla sua directory genitrice nella struttura delle directory.

Il *path assoluto* di un file parte dalla directory root dell'albero delle directory del file system. I file con lo stesso nome creati in directory differenti differiscono nei rispettivi path assoluti. Useremo la convenzione che la prima componente di un path assoluto è un simbolo nullo e la home directory di un utente A è specificata come ~A. Dunque, in [Figura 13.8](#), il path assoluto del file alpha è /A/alpha. Un path alternativo è ~A/alpha.

### 13.4.2 Grafi di directory

In un albero di directory, ogni file eccetto la directory root ha esattamente una directory genitrice. Questa struttura di directory fornisce la separazione totale dei file di utenti differenti e la completa libertà nella scelta dei nomi. Tuttavia, rende la condivisione dei file piuttosto ingombrante. Un utente che desidera accedere ai file di un altro utente deve usare un percorso che contiene due o più directory. Per esempio, in [Figura 13.8](#), l'utente B può accedere al file beta utilizzando il path . . /A/projects/beta oppure ~A/projects/beta.

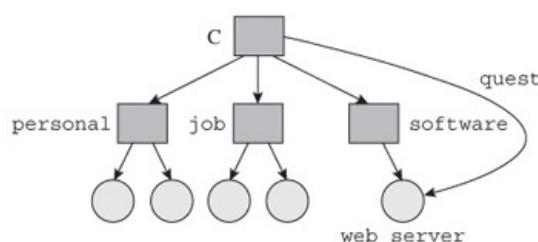
L'uso della struttura ad albero porta a una fondamentale asimmetria nel modo in cui differenti utenti possono accedere a un file condiviso. Il file sarà localizzato in qualche directory appartenente a uno degli utenti, che può accedere a esso con un path più breve rispetto a quello degli altri utenti. Questo problema può essere risolto organizzando le directory in una *struttura a grafo aciclico*. In questa struttura, un file può avere molte directory genitrici, per cui un file condiviso può essere puntato dalle directory di tutti gli utenti che hanno accesso a esso. Le strutture a grafo aciclico sono implementate mediante link.

#### Link

Un *link* è una connessione diretta tra due file esistenti nella struttura delle directory. Può essere scritto come una tripla (*<da\_nome\_file>*, *<a\_nome\_file>*, *<nome\_link>*), dove *<da\_nome\_file>* è una directory e *<a\_nome\_file>* può essere una directory o un file. Una volta creato un link, si può accedere a *<a\_nome\_file>* come se fosse un file chiamato *<nome\_link>* nella directory *<da\_nome\_file>*. Il fatto che *<nome\_link>* è un link nella directory *<da\_nome\_file>* è indicato inserendo il valore "L" nel suo campo *Flag*. L'Esempio 13.4 illustra come impostare un link.

#### Esempio 13.4 - Link in una struttura di directory

La [Figura 13.9](#) mostra la struttura di directory dopo che l'utente C ha creato un link usando il comando (~C, ~C/software/web\_server, quest). Il nome del link è quest. Il link viene creato nella directory ~C e punta al file ~C/software/web\_server. Questo link consente di accedere a ~C/software/web\_server mediante il nome ~C/quest.



**Figura 13.9** Un link nella struttura delle directory.

Un comando di "unlink" cancella un link. L'implementazione dei comandi di link e unlink

coinvolgono la manipolazione delle directory che contengono i file  $<da\_nome\_file>$  e  $<a\_nome\_file>$ . Si possono verificare deadlock quando vengono utilizzati i comandi di link e unlink se diversi processi eseguono questi programmi simultaneamente. Il file system può usare alcune semplici regole per assicurare l'assenza di deadlock (Paragrafo 8.8.1).

### 13.4.3 Operazioni sulle directory

La ricerca di informazioni o file è l'operazione più frequente sulle directory. Altre operazioni sulle directory sono operazioni di manutenzione come la creazione o la cancellazione di file, l'aggiornamento degli elementi relativi ai file quando un processo esegue un'operazione di chiusura, l'elencazione del contenuto di una directory e la cancellazione di una directory.

L'operazione di cancellazione specifica un percorso del file da cancellare, che risulta complesso quando la struttura delle directory è un grafo poiché un file può avere molti genitori. Un file è cancellato solo se ha un singolo genitore; altrimenti viene semplicemente reso inaccessibile dalla sua directory genitrice nel path specificato nel comando di cancellazione. Per semplificare l'operazione di cancellazione, il file system mantiene un contatore di link per ogni file. Il contatore è impostato a 1 quando il file è creato, incrementato di 1 quando un link punta a esso e decrementato di 1 quando viene eseguito un comando di cancellazione. Il file può essere cancellato solo quando il contatore dei link diventa 0.

Questa semplice strategia non è adeguata se la struttura delle directory contiene cicli. Un ciclo si crea quando un link è impostato da una directory a una delle sue directory progenitrici, per esempio, se un link è impostato dalla directory `real_time` in [Figura 13.8](#) alla directory `projects`. Ora il contatore di link di `projects` vale 2, per cui la sua cancellazione usando il path `A/projects` porterebbe solo alla cancellazione dell'elemento relativo a `projects` in `A`. Tuttavia, non c'è ragione di mantenere la directory `projects` e i file raggiungibili da essa, poiché `projects` non sarebbe accessibile dalla directory `home` di nessun utente! Questo problema può essere risolto o utilizzando una tecnica per rilevare i cicli che non sono raggiungibili da nessuna directory `home`, soluzione che può essere costosa, o prevenendo il verificarsi di cicli nella struttura delle directory, soluzione egualmente costosa.

### 13.4.4 Organizzazione delle directory

Una directory dovrebbe essere costituita da una lista monodimensionale in modo da poter effettuare ricerche lineari per trovare il file richiesto. Tuttavia, questa organizzazione è inefficiente se la directory contiene un elevato numero di elementi. Una tabella hash e un albero B+ sono utilizzati per ottenere una maggiore efficienza nelle ricerche.

#### **Directory come hash table**

Una hash table che utilizza l'organizzazione *hash con concatenazione* è stata illustrata nel Paragrafo 12.2.3 nell'ambito delle tabelle delle pagine invertite. Una directory può essere mantenuta utilizzando una hash table più semplice chiamata *hash con open addressing* che richiede una singola tabella. Quando deve essere creato un nuovo file in una directory, una funzione di hash  $h$  viene applicata a una stringa di bit ottenuta dal nome del file, che produce un numero di elemento  $e$ . Se l'elemento  $e$ -simo nella directory è già occupato da un altro file, l'elemento dato da  $(e + 1)mod(n)$ , dove  $n$  è la dimensione della tabella di hash, viene controllato e così via finché non viene trovato un elemento vuoto e i dettagli del nuovo file sono inseriti al suo interno. Quando deve essere aperto un file, viene eseguita una ricerca simile per localizzare l'elemento corrispondente nella directory. Le configurazioni con hash table che non richiedono più di due confronti per localizzare un nome di file sono pratiche, per cui le ricerche in una directory implementata con hash table possono essere effettuate in maniera efficiente. Tuttavia, l'uso di questa organizzazione ha alcuni svantaggi – è inefficiente per modificare la dimensione di una directory o per cancellarne un elemento.

#### **Directory come albero B+**

Un albero B+ è un albero di ricerca a  $m$  uscite dove  $m \leq 2 \times d$  e  $d$  un intero chiamato *ordine* dell'albero. Un albero B+ è un albero bilanciato; ovvero la lunghezza del percorso dalla root a ogni nodo foglia è la stessa. Questa proprietà ha un'utile implicazione per la

ricerca nelle directory: impiega approssimativamente la stessa quantità di tempo per trovare le informazioni relative a ogni file esistente nella directory.

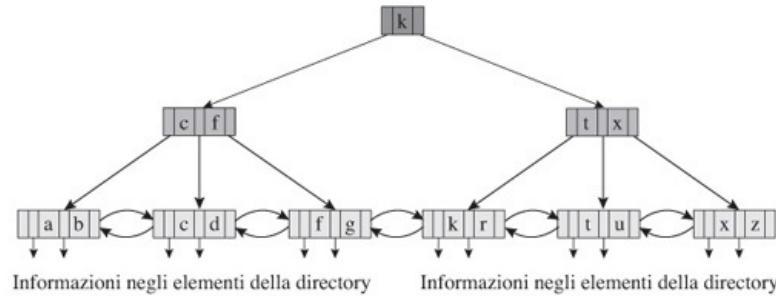
Una directory B+ è organizzata come segue: le informazioni sui file sono memorizzate solo nei nodi foglia dell'albero. I nodi interni vengono usati semplicemente per dirigere la ricerca verso la parte appropriata dell'albero. I nodi interni dell'albero contengono gli *elementi indice*, dove ogni elemento indice è una coppia ordinata composta da un puntatore a un altro nodo nell'albero e un nome di file. L'ultimo elemento indice in un nodo non contiene un nome di file; contiene solo un puntatore a un altro nodo nell'albero. I nodi foglia dell'albero contengono solo *elementi di informazione* dei file; ogni elemento è una coppia ordinata composta da un puntatore all'informazione associata a un nome di file e lo stesso nome del file.

Il nodo radice contiene un numero di elementi compreso tra 2 e  $2 \times d$ , estremi inclusi, dove  $d$  è il grado dell'albero. Un nodo non radice contiene un numero di elementi compreso tra  $d$  e  $2 \times d$ , estremi inclusi. Per facilitare la ricerca di un nome di file, gli elementi in un nodo - siano essi elementi indice o elementi informazione - sono ordinati in ordine lessicografico. Dunque, un nome di file in un elemento è "maggiore" del nome del file nell'elemento precedente nel nodo ed è "minore" del nome del file nell'elemento successivo nel nodo. Un nodo foglia contiene due puntatori aggiuntivi. Questi puntatori puntano, rispettivamente, a nodi dell'albero che sono alla sua sinistra e alla sua destra nell'albero, se ve ne sono. Questi puntatori sono usati per facilitare l'inserimento e la cancellazione degli elementi. Il loro uso non verrà approfondito in questa sede.

Per localizzare un file in una directory, viene eseguita una ricerca nella directory B+, a partire dal suo nodo radice. Il nome del file è confrontato con il nome del file contenuto nel primo elemento indice nel nodo. Se è lessicograficamente "minore" del nome del file contenuto nell'elemento indice, il puntatore nell'elemento indice viene utilizzato per localizzare un altro nodo dell'albero, dove continua la ricerca; altrimenti la ricerca continua con il prossimo elemento indice nel nodo, se esiste, e così via. Se l'elemento indice successivo è l'ultimo elemento indice nel nodo, la ricerca continua semplicemente con il nodo dell'albero puntato dal puntatore nell'elemento indice (si noti che l'ultimo elemento indice in un nodo non contiene un nome di file). Questa procedura viene seguita finché non si incontra un nodo foglia dell'albero. A questo punto, gli elementi informazione nel nodo foglia vengono esaminati mediante un'opportuna tecnica di ricerca come per esempio quella lineare o quella binaria. Se viene trovato un elemento di informazione corrispondente al nome del file che si sta cercando, si utilizza il puntatore nell'elemento di informazione per localizzare l'informazione associata al nome del file; altrimenti il nome del file non esiste nella directory.

### Esempio 13.5 - Directory come albero B+

La [Figura 13.10](#) mostra una directory organizzata come albero B+ di grado 2. Una freccia rivolta verso il basso in un nodo foglia rappresenta un puntatore all'informazione associata con un nome di file. Per cercare un file  $c$ , confrontiamo  $c$  con  $k$ , il nome di file contenuto nel primo elemento indice nella root. Poiché il nome di file  $c$  è "inferiore" a  $k$ , usiamo il puntatore in questo elemento indice per localizzare il nodo dell'albero dove deve proseguire la ricerca. Questo è il nodo che contiene gli elementi indice per il nome di file  $c$  e  $f$ . Poiché  $c$  non è inferiore al nome di file contenuto nel primo elemento indice, possiamo confrontarlo con il nome del file contenuto nel successivo elemento indice nel nodo, ovvero, con  $f$ . È inferiore, dunque usiamo il puntatore in questo elemento indice. Questo puntatore punta a un nodo foglia. Dunque cerchiamo  $c$  negli elementi informazione contenuti in questo nodo. Troviamo una corrispondenza nel primo elemento di informazione, per cui usiamo il puntatore in questo elemento per localizzare le informazioni contenute nella directory relativa al file  $c$ .



**Figura 13.10** Un directory organizzato come albero B+.

I vantaggi di un albero B+ sono la capacità di eseguire ricerche veloci e l'efficienza dei metodi per il bilanciamento dell'albero quando vengono eseguite inserzioni e cancellazioni. Windows NTFS utilizza gli alberi B+ per le directory.

### 13.5 Montaggio dei file system

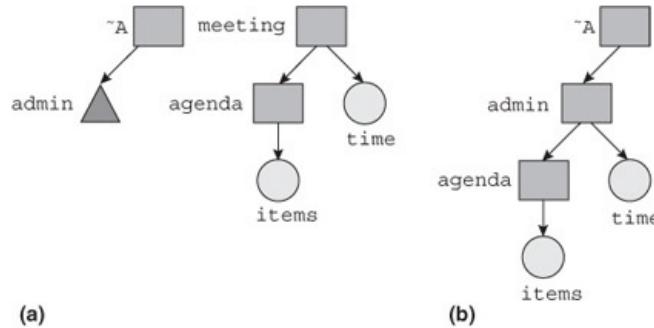
Possono convivere più file system in un sistema operativo. Ogni file system è creato su un disco logico, ovvero su una partizione del disco. I file contenuti in un file system possono essere utilizzati solo quando il file system è *montato*. L'operazione di montaggio (comunemente detto *mount*) "connette" il file system alla struttura delle directory del sistema. Un'operazione di smontaggio (comunemente detto *umount*) "disconnette" un file system. Le operazioni di *mount* e *umount* sono eseguite dall'amministratore di sistema. Queste operazioni forniscono un elemento di protezione ai file in un file system.

Il *mount* crea un effetto analogo a quello di un *link*. La differenza è che il *mount* non altera permanentemente la struttura delle directory. Il suo effetto dura solo finché il file system non è "smontato" o finché il sistema non viene riavviato. Il *mount* del file system è utile quando ci sono più file system nel SO (Paragrafo 13.14.1), o quando un utente di un sistema distribuito desidera accedere ai file presenti in un sistema remoto (Capitolo 20).

Un *mount point* è una directory in cui un file system può essere montato. Un'operazione di *mount* viene eseguita lanciando il comando *mount (<nome\_FS>, <nome\_mount\_point>)*, dove *<nome\_FS>* e *<nome\_mount\_point>*, che sono path, rappresentano, rispettivamente, la root del file system da montare e il mount point. Quando viene eseguita l'operazione di *mount*, la root del file system montato prende il nome *<nome\_mount\_point>*. In questo modo, si può accedere a ogni file con path relativo *ap<sub>i</sub>* nella directory *<nome\_FS>* utilizzando il path *<nome\_mount\_point> / ap<sub>i</sub>*. Se un file system è montato in una directory che già contiene dei file, questi file diventano invisibili all'utente finché non si smonta il file system. Il prossimo esempio illustra l'effetto dell'esecuzione del comando *mount*.

#### Esempio 13.6 - Mount di un file system

In [Figura 13.11\(a\)](#), *~A/admin* è un mount point in una struttura di directory e *meeting* è la directory radice di un altro file system. La [Figura 13.11\(b\)](#) mostra l'effetto del comando *mount (meeting, ~A/admin)*. Si può ora accedere al file *items* come *~A/admin/agenda/items*.



**Figura 13.11** Struttura delle directory (a) prima di un comando mount; (b) dopo un comando mount.

L'effetto di un'operazione di mount è annullato dal corrispondente comando *umount* (*<nome\_FS>*, *<noem\_mount\_point>*). L'operazione di *umount* va a buon fine solo se non ci sono file aperti del file system montato. Per verificare facilmente questa condizione, il file system mantiene un contatore nella root del file system montato per indicare quanti dei suoi file sono stati aperti.

## 13.6 Protezione dei file

Un utente vorrebbe condividere un file con i collaboratori, ma non con altri utenti. Definiamo questa esigenza *condivisione controllata* dei file. Per implementarla, il proprietario di un file specifica quali utenti possono accedere al file e in quale maniera. Il file system memorizza questa informazione nel campo *Protection info* dell'elemento della directory relativo al file (Figura 13.6) e lo utilizza per controllare l'accesso al file.

Nel Capitolo 15 saranno discussi diversi metodi per strutturare l'informazione di protezione dei file. In questo paragrafo, assumiamo che l'informazione relativa alla protezione del file sia memorizzata nella forma di una *access control list* (ACL). Ogni elemento della access control list è una coppia di controllo di accesso nella forma (*<nome\_utente>*, *<lista\_dei\_privilegi\_di\_accesso>*). Quando un processo eseguito da qualche utente X cerca di eseguire un'operazione *<open>* su di un file alpha, il file system cerca la coppia con *<nome\_utente>= X* nella access control list di alpha e controlla se *<open>* è consistente con la lista *<lista\_dei\_privilegi\_di\_accesso>*. Se non lo è, il tentativo di accesso ad alpha fallisce. Per esempio, un tentativo di scrittura da parte di X fallisce se l'elemento relativo ad X nella access control list è (X, *read*), o se la lista non contiene elementi per X.

La dimensione della ACL di un file dipende dal numero di utenti e dal numero di privilegi di accesso definiti nel sistema. Per ridurre la dimensione delle informazioni di protezione, gli utenti possono essere classificati in maniera conveniente in modo da specificare una access control list per ogni classe di utenti piuttosto che per ogni utente. In questo modo una ACL ha solo tante coppie quante sono le classi di utenti. Per esempio, Unix specifica i privilegi di accesso per tre classi di utenti - il proprietario del file, gli utenti nello stesso gruppo del proprietario e tutti gli altri utenti del sistema.

In molti file system, i privilegi di accesso sono di tre tipi: *read* (lettura), *write* (scrittura) ed *execute* (esecuzione). Un privilegio *write* consente di modificare i dati presenti in un file e inoltre di aggiungere nuovi dati: si potrebbe ulteriormente distinguere tra questi due privilegi definendo un nuovo privilegio di accesso chiamato *append*; tuttavia, farebbe aumentare la dimensione dell'informazione di protezione. Il privilegio *execute* consente a un utente di eseguire il programma contenuto in un file. I privilegi di accesso hanno diversi significati per le directory. Il privilegio *read* per una directory implica che è possibile visualizzare il contenuto della directory, mentre il privilegio *write* per una directory implica che è possibile creare nuovi file nella directory. Il privilegio *execute* per una directory permette un accesso tramite la directory, ovvero consente di accedere a un file contenuto nella directory. Un utente può usare il privilegio di esecuzione delle directory per rendere visibile una parte della sua struttura delle directory ad altri utenti.

## 13.7 Allocazione dello spazio sul disco

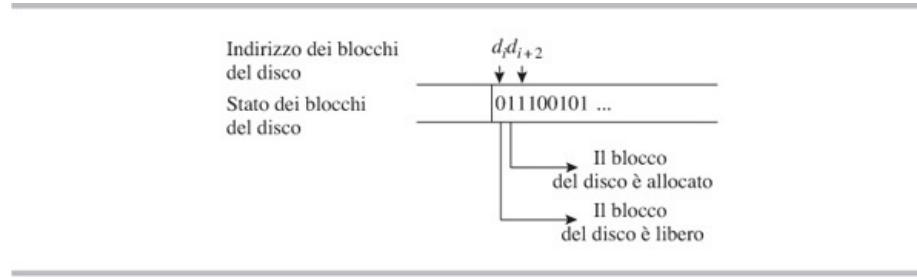
Come menzionato nel Paragrafo 13.5, un disco può contenere molti file system, ognuno nella sua partizione del disco. Il file system ha informazioni sulla partizione in cui è presente un file, ma il sistema IOCS no. Dunque l'allocazione dello spazio sul disco è eseguita dal file system.

I primi file system adottavano il modello di allocazione contigua della memoria (Paragrafo 11.6) allocando una singola area di memoria contigua a un file al momento della creazione. Questo modello era semplice da implementare e forniva efficienza nell'accesso ai dati riducendo il movimento della testina del disco durante l'accesso sequenziale ai dati di un file. Tuttavia, l'allocazione contigua dello spazio su disco portava alla frammentazione esterna. È interessante notare che questo modello soffriva anche di frammentazione interna poiché il file system era progettato per allocare spazio su disco extra per consentire al file di crescere. La contiguità dello spazio su disco conduceva a complicate organizzazioni per evitare l'uso di blocchi del disco danneggiati: il file system identificava i blocchi danneggiati durante la formattazione del disco e annotava i loro indirizzi. Successivamente allocava dei blocchi sostitutivi e costruiva una tabella con gli indirizzi dei blocchi danneggiati e i loro sostituti. Durante un'operazione di lettura/scrittura, IOCS controllava se i blocchi cui si accedeva erano blocchi danneggiati. In caso affermativo, recuperava l'indirizzo del blocco sostitutivo e vi accedeva.

I moderni file system adottano l'allocazione della memoria non contigua (Paragrafo 11.7) per l'allocazione dello spazio sul disco. In questo approccio, una parte di spazio sul disco è allocata su richiesta, ovvero, quando viene creato un file oppure quando la sua dimensione aumenta a seguito di un'operazione di aggiornamento. Il file system deve risolvere tre problemi per implementare questo approccio:

- *gestione dello spazio libero sul disco*: tenere traccia dello spazio libero sul disco e allocarlo quando un file richiede un nuovo blocco del disco;
- *evitare movimenti eccessivi della testina del disco*: garantire che i dati in un file non siano sparsi in diverse parti del disco, poiché ciò causerebbe un movimento eccessivo delle testine del disco durante l'elaborazione del file;
- *accesso ai dati del file*: mantenere informazioni sullo spazio sul disco allocato a un file e utilizzarlo per trovare i blocchi del disco che contengono i dati richiesti.

Il file system può mantenere una *free list* dello spazio sul disco e allocarlo quando un file richiede un nuovo blocco. Alternativamente, può usare una tabella chiamata *mappa dello stato del disco* (DSM, Disk State Map) per indicare lo stato dei blocchi del disco. La DSM ha un elemento per ogni blocco del disco, che indica se il blocco del disco è libero o è già stato allocato a un file. Questa informazione può essere mantenuta in un singolo bit per cui una DSM è anche chiamata *bitmap*. La [Figura 13.12](#) illustra una DSM. Un 1 in un elemento indica che il blocco del disco corrispondente è allocato. La DSM viene consultata ogni volta che un nuovo blocco del disco deve essere allocato a un file.

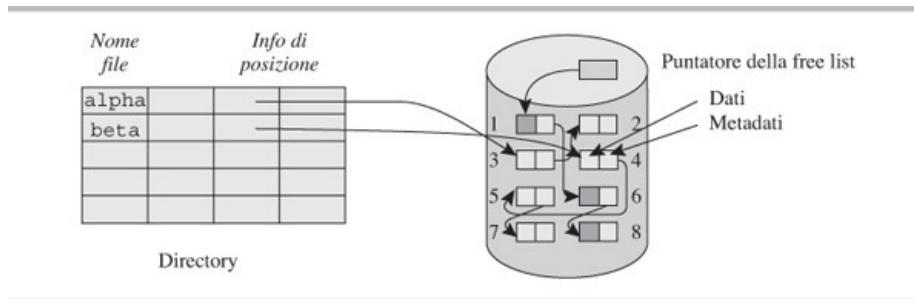


**Figura 13.12** Disk status map (DSM).

Per evitare di disperdere i dati di un file su differenti parti del disco, il file system confina l'allocazione dello spazio sul disco per un file o a blocchi del disco consecutivi, che formano una *estensione*, anche chiamata *cluster*, o a cilindri consecutivi in un disco, che formano un *gruppo di cilindri* (Paragrafo 14.3.2). L'uso di una status map, piuttosto che di una free list, ha il vantaggio di consentire al file system di prelevare prontamente i blocchi disco da un cluster o da un gruppo di cilindri. Discuteremo due approcci fondamentali all'allocazione dello spazio su disco non contiguo. Differiscono nel modo in cui mantengono l'informazione sullo spazio su disco allocato a un file.

### 13.7.1 Allocazione concatenata

Un file è rappresentato da una lista concatenata di blocchi del disco. Ogni blocco del disco ha due campi al suo interno: *Dati* e *Metadati*. Il campo *Dati* contiene i dati scritti nel file, mentre il campo *Metadati* è il campo di tipo link, che contiene l'indirizzo del prossimo blocco del disco allocato al file. La [Figura 13.13](#) illustra l'allocazione concatenata. Il campo *Info di posizione* dell'elemento della directory del file *alpha* punta al primo blocco sul disco del file. Agli altri blocchi si accede seguendo i puntatori nella lista dei blocchi del disco. L'ultimo blocco del disco contiene un'informazione null nel campo metadati. In questo modo, il file *alpha* si compone dei blocchi 3 e 2, mentre il file *beta* si compone dei blocchi 4, 5 e 7. Lo spazio libero sul disco è rappresentato da una *free list* in cui ogni blocco libero contiene un puntatore al successivo blocco libero. Quando un blocco è richiesto per memorizzare nuovi dati aggiunti al file, un blocco viene estratto dalla free list e aggiunto alla lista dei blocchi del file. Per cancellare un file, la lista di blocchi del file viene semplicemente aggiunta alla free list.



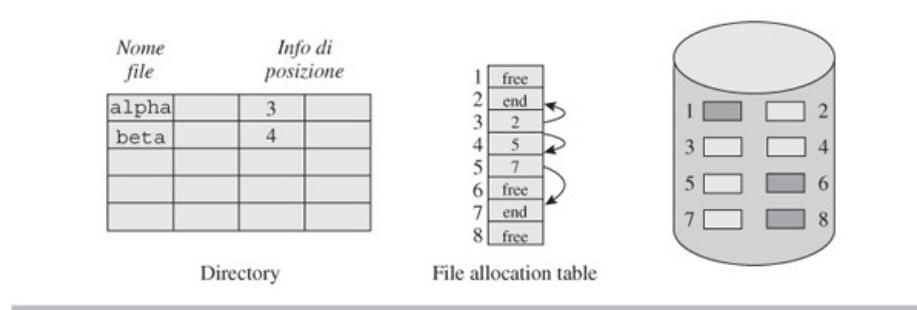
**Figura 13.13** Allocazione concatenata dello spazio su disco.

L'allocazione concatenata è semplice da implementare e causa di un basso overhead di allocazione/deallocazione. Inoltre, supporta i file sequenziali in modo abbastanza efficiente. Tuttavia, l'accesso ai file con organizzazione non sequenziale non è efficiente. Anche l'affidabilità è scarsa poiché il danneggiamento del campo dei metadati in un blocco del disco può comportare la perdita dei dati dell'intero file. In modo analogo, il funzionamento del file system può essere influenzato se un puntatore nella free list è danneggiato. Discuteremo queste problematiche di affidabilità nel Paragrafo 13.11.

#### File Allocation Table (FAT)

MS-DOS utilizza una variante dell'allocazione concatenata che memorizza i metadati separatamente dai dati del file. Una *file allocation table* (FAT) di un disco è un array che ha un elemento per ogni blocco del disco. Per un blocco allocato a un file, il corrispondente elemento della FAT contiene l'indirizzo del blocco successivo. In questo modo il blocco e il suo elemento nella FAT insieme formano una coppia che contiene la stessa informazione contenuta nel blocco nel classico schema dell'allocazione concatenata.

L'elemento di una directory relativo a un file contiene l'indirizzo del primo blocco sul disco. L'elemento della FAT corrispondente a questo blocco contiene l'indirizzo del secondo blocco e così via. L'elemento della FAT corrispondente all'ultimo blocco contiene un codice speciale per indicare che il file termina con questo blocco. La [Figura 13.14](#) illustra la FAT per il disco di [Figura 13.13](#). Il file *alpha* si compone dei blocchi 3 e 2. Dunque l'elemento della directory di *alpha* contiene 3. L'elemento della FAT relativo al blocco 3 contiene 2 e l'elemento della FAT relativo al blocco 2 indica che il file termina con quel blocco. Il file *beta* si compone dei blocchi 4, 5 e 7. La FAT può anche essere usata per memorizzare l'informazione relativa allo spazio libero. La lista dei blocchi liberi può essere memorizzata come se fosse un file e l'indirizzo del primo blocco libero può essere contenuto in un puntatore alla free list. In alternativa, del codice speciale può essere memorizzato nell'elemento della FAT corrispondente a un blocco libero, per esempio il codice "free" in [Figura 13.14](#).

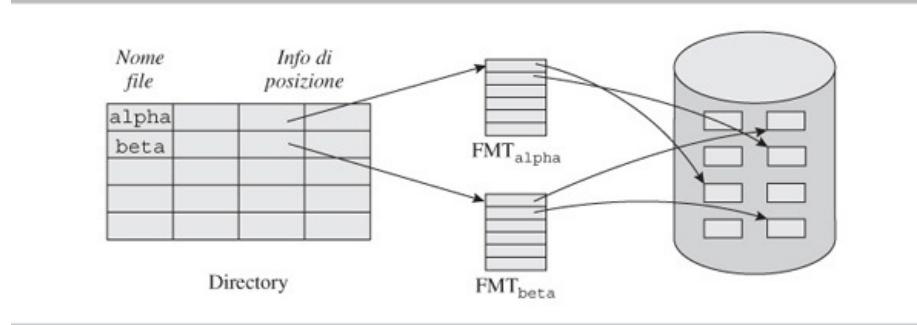


**Figura 13.14** File Allocation Table (FAT).

L'uso della FAT piuttosto che la classica allocazione concatenata comporta una penalizzazione nelle prestazioni, poiché è necessario accedere alla FAT per ottenere l'indirizzo del blocco successivo. Per risolvere questo problema, la FAT è mantenuta in memoria durante l'elaborazione del file. L'uso della FAT fornisce maggiore affidabilità rispetto alla classica allocazione concatenata poiché il danneggiamento di un blocco contenente i dati del file comporta danni limitati. Tuttavia, il danneggiamento di blocco usato per memorizzare la FAT risulta disastroso.

### 13.7.2 Allocazione indicizzata

Nell'allocazione indicizzata, un indice chiamato *file map table* (FMT) viene mantenuto per annotare gli indirizzi dei blocchi del disco allocati a un file. Nella sua forma più semplice, un FMT può essere un array contenente gli indirizzi dei blocchi del disco. Ogni blocco contiene un singolo campo: il campo dati. Il campo *Info di posizione* di un elemento di una directory relativo a un file punta alla FMT per il file (Figura 13.15). Nella discussione che segue usiamo la notazione  $fmt_{\text{alpha}}$  per la FMT del file alpha. Se la dimensione del file alpha cresce, il DSM viene esaminato per localizzare un blocco libero e l'indirizzo del blocco è aggiunto a  $fmt_{\text{alpha}}$ . La deallocazione è eseguita quando alpha è cancellato. Tutti i blocchi del disco puntati da  $fmt_{\text{alpha}}$  sono marcati come liberi prima di  $fmt_{\text{alpha}}$  e l'elemento di directory relativo ad alpha è cancellato.

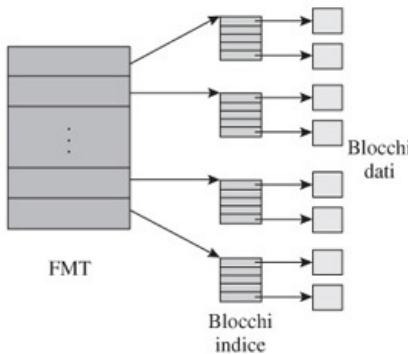


**Figura 13.15** Allocazione indicizzata dello spazio su disco.

Il problema dell'affidabilità è meno sentito nell'allocazione indicizzata rispetto all'allocazione concatenata poiché il danneggiamento di un elemento della FMT comporta solo un danno limitato. Confrontato con l'allocazione concatenata, l'accesso a un file ad accesso sequenziale è meno efficiente poiché bisogna accedere alla FMT di un file per ottenere l'indirizzo del blocco sul disco successivo. Tuttavia, l'accesso ai record in un file ad accesso diretto è più efficiente poiché l'indirizzo del blocco che contiene un record specifico può essere ottenuto direttamente dalla FMT. Per esempio, se il calcolo dell'indirizzo analogo alle Equazioni 13.1 e 13.2 mostra che un record richiesto esiste nel blocco  $i$ -esimo di un file, il suo indirizzo può essere ottenuto dall'elemento  $i$ -esimo della FMT.

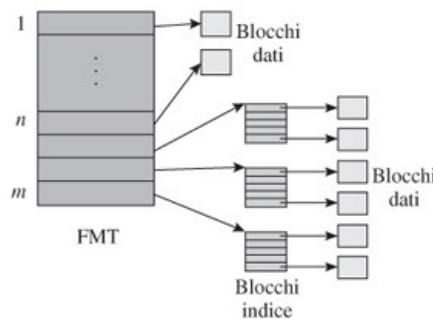
Per file piccoli, la FMT può essere memorizzata nell'elemento della directory relativo al file; risulta conveniente ed efficiente. Per un file medio o grande, la FMT non potrà essere contenuta nell'elemento della directory. Un'allocazione indicizzata a due livelli mostrata nella Figura 13.16 può essere usata per queste FMT. In questa organizzazione,

ogni elemento della FMT contiene l'indirizzo di un *blocco indice*. Un blocco indice non contiene dati; contiene elementi che contengono gli indirizzi dei blocchi dati. Per accedere ai blocchi dati, prima accediamo a un elemento della FMT e otteniamo l'indirizzo di un blocco indice. Successivamente, accediamo a un elemento del blocco indice per ottenere l'indirizzo di un blocco dati. Questa organizzazione ricorda la tabella delle pagine multilivello (Paragrafo 12.2.3). Il blocco indice ricorda le pagine di una tabella delle pagine per il file e la FMT ricorda la tabella delle pagine di più alto livello. Tale FMT è compatta; dunque anche le FMT di file grandi possono essere contenute in un elemento della directory. Tuttavia, l'accesso ai blocchi dati è più lento poiché sono coinvolti due livelli di indirizzamento.



**Figura 13.16** Organizzazione di una FMT a due livelli.

Alcuni file system implementano un'organizzazione ibrida della FMT che include alcune delle caratteristiche dell'allocazione classica e dell'allocazione indicizzata multilivello. La [Figura 13.17](#) mostra questa organizzazione. I primi elementi nella FMT, per esempio  $n$  elementi, puntano a blocchi dati come nell'allocazione indicizzata. Gli altri elementi puntano a blocchi indice. Il vantaggio di questa organizzazione è che piccoli file contenenti un numero di blocchi pari o inferiore ad  $n$  continuano a essere accessibili in maniera efficiente, poiché la FMT non utilizza i blocchi indice. I file di medie o grandi dimensioni soffrono di un parziale degrado delle prestazioni di accesso a causa dei livelli di indirizzamento. Il file system di Unix utilizza una variante dell'organizzazione ibrida della FMT.



**Figura 13.17** Organizzazione ibrida di una FMT.

## Esercizio

### Allocazione dello spazio sul disco

Un file è dotato di una struttura a record di 512 byte ed è allocato su un disco caratterizzato da blocchi anch'essi di 512 byte.

Si considerino le tre strategie di allocazione del file - contigua, concatenata e indicizzata - e si supponga che le informazioni che riguardano il file, in ognuno dei tre casi, siano già in memoria centrale. L'ultimo record letto dal file è il record 3. Il prossimo record da leggere è il record 8.

Per ognuno dei tre casi, si calcoli quanti accessi a disco sono necessari per la lettura del record 8 e si motivi la risposta.

### **Soluzione**

Notiamo innanzitutto, che, data la dimensione di record e blocco sopra specificate, ogni record del file occupa esattamente un blocco del disco. Quindi:

- **allocazione contigua:** 1 accesso alla memoria di massa permetterà di reperire direttamente il record 8;
- **allocazione concatenata:** 5 accessi alla memoria, in quanto i record che separano il terzo dall'ottavo devono essere letti sequenzialmente;
- **allocazione indicizzata:** 2 accessi alla memoria, uno al blocco indice, se non presente in memoria centrale, per reperire il puntatore al blocco dati richiesto, l'altro al blocco dati.

### **13.7.3 Il problema delle prestazioni**

L'utilizzo dei blocchi come unità di allocazione dello spazio sul disco presenta due problematiche: la dimensione dei metadati, ovvero i dati di controllo del file system e l'efficienza nell'accesso ai dati del file. Entrambe le problematiche possono essere affrontate utilizzando un'unità di allocazione più grande. Dunque i moderni file system tendono a usare un'estensione, anche detta *cluster*, come unità di allocazione dello spazio sul disco. Un'estensione è un insieme di blocchi consecutivi. L'uso di grandi estensioni fornisce una migliore efficienza nell'accesso; tuttavia, causa una maggiore frammentazione interna. Per ottenere entrambi i vantaggi, i file system preferiscono utilizzare dimensioni di estensione variabili. I metadati contengono la dimensione di un'estensione insieme con il suo indirizzo.

### **13.8 Interfaccia tra file system e IOCS**

Il file system utilizza IOCS per eseguire le operazioni di I/O che il sistema IOCS implementa attraverso le chiamate al kernel. L'interfaccia tra il file system e IOCS consiste di tre strutture dati - la *file map table* (FMT), il *file control block* (FCB) e la *active file table* (AFT) - e di funzioni che eseguono le operazioni di I/O. L'uso di queste strutture dati evita l'elaborazione ripetuta degli attributi dei file da parte del file system e fornisce un metodo conveniente per tracciare lo stato delle attività di elaborazione dei file in esecuzione.

Come discusso precedentemente nel Paragrafo 13.7.2, il file system alloca lo spazio sul disco a un file e memorizza le informazioni sullo spazio allocato sul disco nella *file map table* (FMT). La FMT tipicamente viene mantenuta in memoria durante l'elaborazione di un file.

Un *file control block* (FCB) contiene tutte le informazioni riguardanti un'attività di elaborazione di un file. Queste informazioni possono essere classificate in tre categorie mostrate nella [Tabella 13.3](#). Le informazioni nella categoria relativa all'organizzazione del file è semplicemente estratta tramite l'istruzione di dichiarazione del file contenuta in un programma, o è ricavata dal compilatore, per esempio informazioni come la dimensione di un record e il numero di buffer vengono estratte dalla dichiarazione di un file, mentre il nome del metodo di accesso è ricavato dal tipo e dall'organizzazione del file.

| <b>Categoria</b>             | <b>Campi</b>   |
|------------------------------|--|
| Organizzazione del file      | Nome del file<br>Tipo di file, organizzazione e metodo di accesso<br>Tipo di dispositivo e indirizzo<br>Dimensione di un record<br>Dimensione di un blocco<br>Numero di buffer<br>Nome del metodo di accesso |
| Informazioni sulla directory | Informazioni sull'elemento della directory relativo al file  |

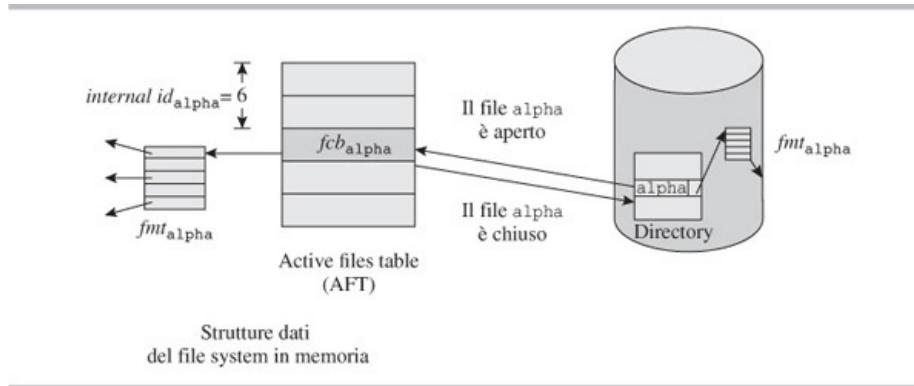
|   |  |
|---|--|
| Indirizzo del FCB della directory genitrice                       |  |
| Indirizzo della file map table (FMT) (o la stessa file map table) |  |
| Informazioni relative alla protezione                             |  |
| Stato corrente dell'elaborazione                                  | Indirizzo del prossimo record da elaborare<br>Indirizzi dei buffer |

**Tabella 13.3** Campi nel File Control Block (FCB).

Il compilatore utilizza queste informazioni come parametri della chiamata open. Quando la chiamata viene eseguita durante la dichiarazione del file, il file system inserisce questa informazione nel FCB. Le informazioni relative alla directory vengono copiate nell'FCB mediante azioni combinate del file system e del IOCS quando viene creato un nuovo file. Le informazioni riguardanti lo stato corrente dell'elaborazione vengono scritte nel FCB dal IOCS. Queste informazioni sono aggiornate continuamente durante l'elaborazione di un file.

La *active file table* (AFT) mantiene gli FCB di tutti i file aperti. La AFT risiede nello spazio di indirizzamento del kernel in modo che i processi utente non possano alterarla. Quando viene aperto un file, il file system memorizza il suo FCB in un nuovo elemento della AFT. Lo scostamento di questo elemento nella AFT è chiamato l'*id interno* del file. L'id interno viene passato nuovamente al processo, che lo usa come parametro in tutte le chiamate successive.

La [Figura 13.18](#) mostra l'organizzazione quando viene aperto un file alpha. Il file system copia  $fmt_{alpha}$  in memoria; crea  $fcb_{alpha}$ , che è un FCB per alpha, nella AFT; inizializza i suoi campi in modo appropriato; e passa l'offset nella AFT, che in questo caso è 6, al processo come  $id\_interno_{alpha}$ .



**Figura 13.18** Interfaccia tra file system e IOCS - AFT, FCB e FMT.

Il file system supporta le seguenti operazioni:

- `open (<nome_file>, <modalità_di_elaborazione>, <attributi_del_file>);`
- `close (<id_interno_del_file>);`
- `read/write (<id_interno_del_file>, <record_info>, <indirizzo_area_I/O>).`

*<nome\_file>* è il path assoluto o relativo del file da aprire. *<modalità\_di\_elaborazione>* indica quale tipo di operazioni saranno eseguite sul file - i valori "inserimento," "creazione," e "aggiunta" hanno un chiaro significato, mentre "aggiornamento" indica che il processo intende aggiornare i dati esistenti. *<attributi\_del\_file>* è una lista di attributi del file, come l'organizzazione del file, la dimensione del record e le informazioni sulla protezione. È rilevante solo quando viene creato un nuovo file: gli attributi dalla lista vengono copiati in questo momento nell'elemento della directory relativo al file. *<record\_info>* indica l'identità del record da leggere o scrivere se il file è elaborato in modo non sequenziale. *<indirizzo\_area\_I/O>* indica l'indirizzo dell'area di memoria da cui devono essere letti i dati del record, o l'area di memoria che contiene i dati da scrivere nel record.

L'interfaccia IOCS supporta le seguenti operazioni:

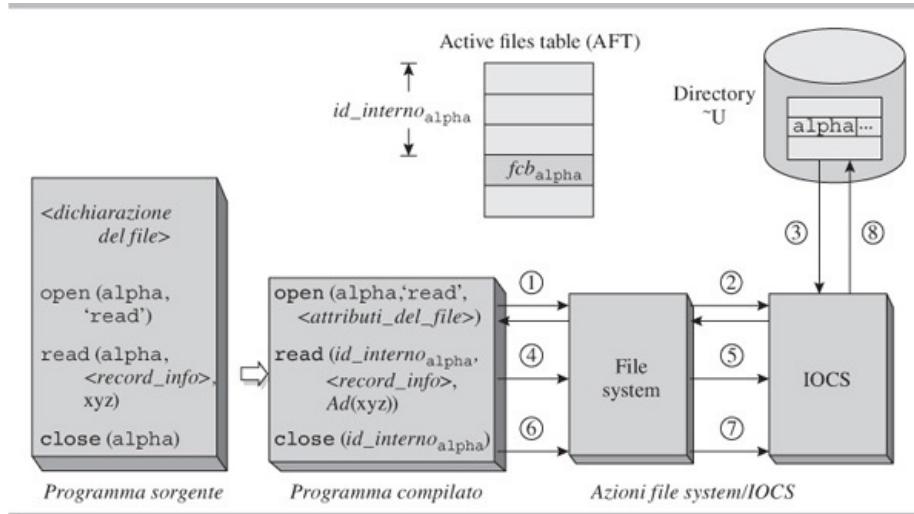
- `iocs-open (<id_interno_del_file>, <indirizzo_elemento_directory>);`

- `iocs-close (<id_interno_del_file>, <indirizzo_elemento_directory>);`
- `iocs-read/write (<id_interno_del_file>, <record_info>, <indirizzo_area_I/O>).`

Ognuna di queste operazioni è una generica operazione per i vari tipi di organizzazione dei file supportati dal file system. Si compone di due parti: esegue alcune azioni che sono comuni a tutte le organizzazioni dei file e invoca un modulo del metodo di accesso indicato nella FCB del file per eseguire azioni specifiche richieste per l'organizzazione del file utilizzata.

Le operazioni `iocs-open` e `iocs-close` sono operazioni specializzate di lettura e scrittura che copiano le informazioni nella FCB dall'elemento della directory o dalla FCB nell'elemento della directory. Le operazioni `iocs-read/write` accedono alla FCB per ottenere le informazioni relative allo stato corrente dell'attività di elaborazione, come l'indirizzo del prossimo record da elaborare. Quando un'operazione di scrittura richiede più spazio sul disco, `iocs-write` invoca una funzione del file system per eseguire l'allocazione dello spazio sul disco (Paragrafo 13.7).

La [Figura 13.19](#) rappresenta un diagramma schematico dell'elaborazione di un file esistente `alpha` in un processo eseguito da un utente `U`. Il compilatore sostituisce le istruzioni `open`, `read` e `close` nel codice sorgente, rispettivamente, con chiamate alle operazioni `open`, `read` e `close` del file system. Di seguito sono riportati gli step fondamentali per l'elaborazione dei file che coinvolgono il file system e il IOCS, mostrati con frecce numerate in [Figura 13.19](#).



**Figura 13.19** Panoramica dell'elaborazione dei file.

1. Il processo esegue le chiamate `open (alpha, 'read', <attributi_del_file>)`. La chiamata restituisce `id_interno_alpha` se la modalità di elaborazione "read" è consistente con le informazioni di protezione del file. Il processo salva `id_interno_alpha` per utilizzarlo durante l'esecuzione delle operazioni sul file `alpha`.
2. Il file system crea un nuovo FCB nella tabella dei file attivi. Ricava il percorso per `alpha` come descritto successivamente nel Paragrafo 13.9.1, localizza l'elemento della directory relativo ad `alpha` e memorizza l'informazione a esso relativa nel nuovo FCB per utilizzarlo durante la chiusura del file. In questo modo, il nuovo FCB diventa `fcb_alpha`. Il nuovo file system a questo punto esegue una chiamata `iocs-open` con `id_interno_alpha` e l'indirizzo dell'elemento della directory relativo ad `alpha` come parametri.
3. Il IOCS accede all'elemento della directory relativo ad `alpha` e copia la dimensione del file e l'indirizzo della FMT, o la stessa FMT, dall'elemento della directory in `fcb_alpha`.
4. Quando il processo vuole leggere un record di `alpha` nell'area `xyz`, richiama l'operazione `read` del file system con `id_interno_alpha`, `<record_info>` e `Ad(xyz)` come parametri.
5. Le informazioni sulla localizzazione di `alpha` sono ora disponibili in `fcb_alpha`. Dunque l'operazione `read/write` richiama semplicemente le operazioni `iocs-read/write`.
6. Il processo richiama l'operazione `close` con `id_interno_alpha` come parametro.

7. Il file system effettua una chiamata `iocs-close` con  $id\_interno_{\alpha}$ .
8. Il IOCS ottiene le informazioni sull'elemento della directory relativo ad  $\alpha$  da  $fcb_{\alpha}$  e copia la dimensione del file e l'indirizzo della FMT, o la FMT stessa, da  $fcb_{\alpha}$  nell'elemento relativo ad  $\alpha$ .

## 13.9 Elaborazione dei file

In questo paragrafo discutiamo l'elaborazione dei file strutturati, in cui le operazioni di lettura/scrittura vengono eseguite su di un record.

### 13.9.1 Azioni del file system per la `open`

Lo scopo della chiamata `open (<path_name>, <modalità_di_elaborazione>, <attributi_file>)`, dove  $<path\_name>$  è un path assoluto o relativo per un file  $<nome\_file>$ , è di impostare l'elaborazione del file. Come descritto nel Paragrafo 13.8, `open` esegue le seguenti azioni.

1. Termina il processo se  $<modalità_di_elaborazione>$  non è consistente con le informazioni di protezione del file. Altrimenti, crea un FCB per il file  $<nome\_file>$  nella AFT e inserisce le informazioni rilevanti all'interno dei suoi file. Se  $<nome\_file>$  è un nuovo file, scrive anche  $<attributi_file>$  all'interno dell'elemento della directory relativo al file.
2. Passa l'id interno del file  $<nome\_file>$  nuovamente al processo per utilizzarlo nelle azioni di elaborazione del file.
3. Se il file  $<nome\_file>$  deve essere creato o incrementato, esegue delle azioni per aggiornare l'elemento della directory relativo al file quando il processo esegue la chiamata `close`.

La procedura chiamata *risoluzione del path* attraversa tutte le componenti di un path e controlla la validità di ogni componente. Utilizza due puntatori allo scopo chiamati *puntatore FCB del file* e *puntatore FCB della directory*. Punta il *puntatore al FCB del file* al FCB del file corrispondente alla componente corrente nel path e il *puntatore al FCB della directory* al FCB della sua directory genitrice. Al termine della risoluzione del path, il *puntatore FCB del file* viene usato per determinare l'*id interno* del file. La risoluzione del path consiste dei seguenti passi.

1. Se viene utilizzato un path assoluto, localizza il FCB della directory radice del file system nella AFT; altrimenti, localizza il FCB della directory corrente. (Questo passo suppone che gli FCB di queste directory siano già stati creati nella AFT. In caso contrario, dovrebbero essere creati in questo passo.) Imposta il *puntatore FCB della directory* in modo da puntare a questo FCB.
2.
  - a. Cerca la prossima componente del path nella directory rappresentata dal *puntatore al FCB della directory*. Indica un errore se la componente non esiste o se il processo proprietario non ha i privilegi per accedervi.
  - b. Crea un FCB per i file descritti dalla componente del path. Memorizza questo FCB in un elemento libero della AFT. Copia il *puntatore FCB della directory* in questo FCB.
  - c. Imposta il *puntatore FCB della directory* in modo che punti a questo FCB.
  - d. Se questa non è l'ultima componente del percorso, inizializza il FCB appena creato utilizzando le informazioni contenute nell'elemento della directory relativo al file. Imposta il *puntatore FCB della directory = puntatore FCB del file* e ripete il passo 2.
3.
  - a. Se il file già esiste, copia la dimensione del file e il puntatore alla FMT dall'elemento della directory relativo al file all'interno del FCB puntato dal *puntatore FCB del file*.
  - b. Se il file non esiste, crea la FMT del file e memorizza il suo indirizzo nel FCB. (Questa azione può coinvolgere l'allocazione di un blocco del disco per la FMT.)
4. Imposta l'*id interno* del file all'offset del *puntatore FCB del file* nella AFT. Copia il *puntatore FCB della directory* nel FCB del file. Restituisce l'*id interno* al processo.

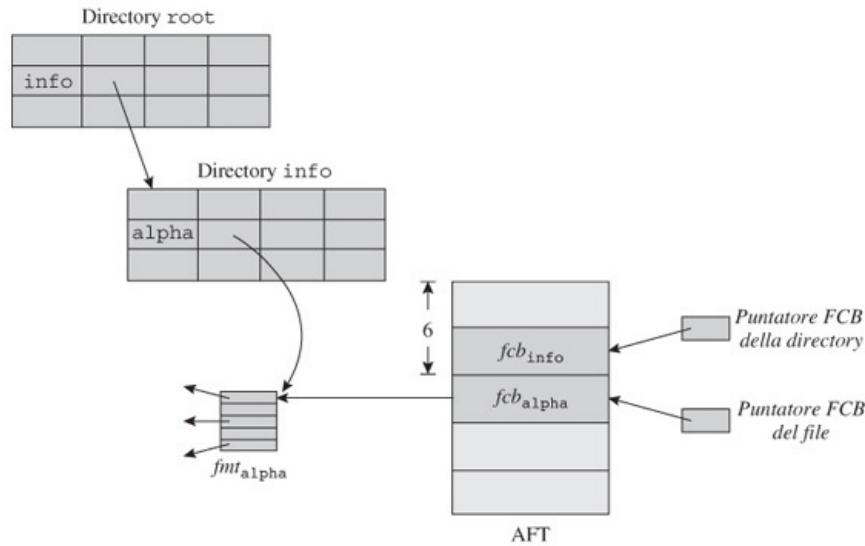
Oltre alle azioni appena descritte, il file system può eseguire altre azioni nell'interesse dell'efficienza. Per esempio, durante l'apertura di un file esistente può copiare una parte o l'intera FMT del file in memoria [Passo 3(a)]. Questa azione assicura l'accesso

efficiente ai dati nel file. Inoltre, solo i FCB puntati dal *puntatore FCB della directory* e *puntatore FCB del file* sono necessari durante l'elaborazione del file, per cui gli altri FCB creati durante la risoluzione del path possono essere distrutti.

L'esempio seguente illustra le strutture dati costruite dal file system quando viene aperto un file.

#### Esempio 13.7 - Implementazione dell'operazione `open`

La [Figura 13.20](#) mostra il risultato delle azioni del file system dopo aver eseguito la chiamata



**Figura 13.20** Azioni del file system al momento della `open`.

```
open (/info/alpha, ...);
```

Il percorso usato nella chiamata `open` è un path assoluto. Il file system cerca il nome *info* nella directory root e crea un FCB per *info* nella AFT. A questo punto cerca il nome *alpha* in *info* e crea un FCB per *alpha* nella AFT. Il *puntatore FCB della directory* punta a *fcb<sub>info</sub>* e il *puntatore FCB del file* punta a *fcb<sub>alpha</sub>*. Poiché *alpha* è un file esistente, il suo puntatore FMT viene copiato in *fcb<sub>alpha</sub>* dall'elemento della directory relativo al file *alpha*. La chiamata restituisce l'id interno di *alpha*, che è 6.

Il comando `mount (<nome_FS>, <nome_mount_point>)` monta *<nome\_FS>* nel mount point (Paragrafo 13.4). Un modo semplice per implementare il montaggio è di cambiare temporaneamente l'elemento della directory relativo a *<nome\_mount\_point>* nella sua directory genitrice in modo da puntare all'elemento della directory relativo a *<nome\_FS>*.

Quando un mount point viene attraversato durante la risoluzione di un percorso, il file system deve passare dalla struttura della directory del mount point alla struttura della directory del file system montato, o viceversa. Per facilitare questa procedura, nell'eseguire un comando `mount`, il file system inserisce il valore "M" nel campo *Flag* dell'elemento della directory relativo a *<nome\_FS>* e mantiene una *mount table* per memorizzare le coppie nella forma (*<nome\_FS>*, *<nome\_mount\_point>*). Per esempio, quando la chiamata `mount (meeting, ~A/admin)` del Paragrafo 13.4 viene eseguita, il file system aggiunge la coppia (*meeting*, *~A/admin*) alla *mount table*. Durante la risoluzione del path, viene consultata questa tabella quando si incontra un mount point durante l'attraversamento della struttura delle directory dal genitore al figlio (per l'operatore slash (/) nel percorso) o dal figlio al genitore (per l'operatore "..."). Il file system deve anche garantire che l'allocazione dello spazio su disco effettuata durante l'elaborazione di un file montato avvenga nel file system montato piuttosto che nel file system ospite.

#### 13.9.2 Azioni del file system durante un'operazione su file

Dopo l'apertura di un file *<nome\_file>*, un processo eseguito da un utente U lancia delle operazioni di lettura o scrittura sul file. Ognuna di queste operazioni viene tradotta in una chiamata:

*<open> (id interno, id record, <indirizzo IO\_area>);*

dove *id interno* è l'id interno di *<nome\_file>* restituito dalla chiamata open e *id record* è assente se l'operazione viene eseguita su file ad accesso sequenziale poiché l'operazione è necessariamente eseguita sul *successivo* record nel file. Il file system esegue le seguenti azioni per eseguire questa chiamata.

1. Localizza il FCB di *<nome\_file>* nella AFT utilizzando l'*id interno*.
2. Cerca nella access control list di *<nome\_file>* la coppia (U, ...). Restituisce un errore se le informazioni di protezione trovate nel FCB del file non consentono all'utente U di eseguire *<open>* sul file.
3. Effettua una chiamata *iocs-read* o *iocs-write* con parametri *id interno*, *id record* e *<indirizzo IO area>*. Per i file ad accesso non sequenziale, l'operazione viene eseguita sul record indicato. Per i file ad accesso sequenziale, l'operazione viene eseguita sul record il cui indirizzo è nel campo FCB "indirizzo del prossimo record da elaborare" e il contenuto di questo campo viene aggiornato per puntare al prossimo record nel file.

Nel passo 3, il IOCS e il metodo di accesso invocato ottengono la FMT del file dal suo FCB e lo usano per convertire l'*id del record* in una coppia nella forma (*id del blocco*, *byte di offset*). Se termina lo spazio a disposizione durante un'operazione di scrittura, richiamano un modulo del file system, che alloca un nuovo blocco e aggiungono l'indirizzo alla FMT.

#### **Esempio 13.8 - Implementazione delle operazioni read/write**

Successivamente alla chiamata open dell'Esempio 13.7, un chiamata read (*alpha*, 25, ...) eseguita dal processo, dove 25 è l'*id del record*, porterebbe alla chiamata *iocs-read* (6, 25, ...). Se i blocchi del disco hanno dimensione 1000 byte ognuno e un record ha dimensione 100 byte, il IOCS convertirà *id record* nel blocco numero 3 e record numero 5 nel blocco, che implica un offset di 400 byte. A questo punto l'indirizzo del terzo blocco allocato ad *alpha* è ottenuto dalla sua FMT e questo blocco viene letto per ottenere il record desiderato.

### **13.9.3 Azioni del file system per la close**

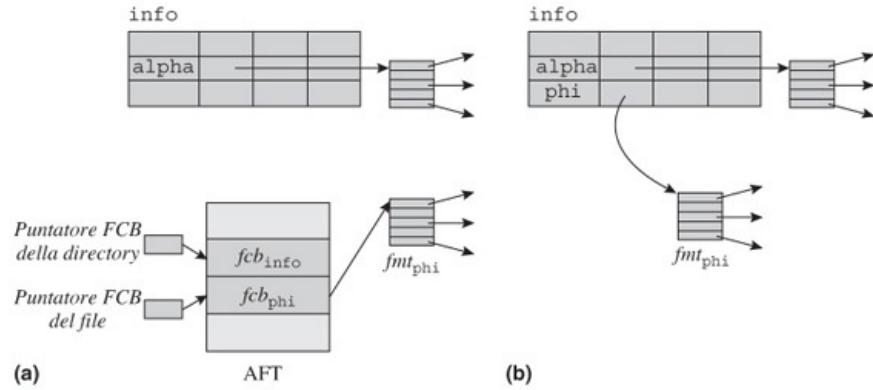
Il file system esegue le seguenti azioni quando un processo esegue l'istruzione close (*id interno*, ...).

1. Se il file è stato appena creato o aggiornato.
  - a. Se è un file appena creato, crea un elemento relativo al file nella directory puntata dal *puntatore FCB della directory*. Se il formato dell'elemento della directory contiene un campo dove può essere memorizzata completamente la FMT, copia la FMT in questo campo; altrimenti, prima scrive la FMT in un blocco del disco e poi copia l'indirizzo di questo blocco nell'elemento della directory relativo al file.
  - b. Se il file è stato aggiornato, l'elemento della directory relativo al file viene aggiornato usando il *puntatore al FCB della directory*.
  - c. Se necessario, ripete i passi 1b e 1c per aggiornare altre directory nel path del file dopo aver impostato il *puntatore FCB del file* = *puntatore FCB della directory* e *puntatore FCB della directory* = indirizzo del FCB della directory genitrice trovato nel FCB del file. Se i loro FCB fossero cancellati dopo la open, i file della directory dovrebbero essere aperti e aggiornati.
2. Il FCB del file e i FCB delle sue directory genitrice e progenitrice vengono cancellati dalla AFT.

#### **Esempio 13.9 - Implementazione dell'operazione close**

La Figura 13.21 illustra le azioni del file system prima e dopo l'esecuzione del comando close/phi per un file appena creato phi che era stato aperto usando il path /info/phi. Viene creato un elemento della directory info relativo al file phi e un puntatore a *fmt<sub>phi</sub>* viene inserito nel campo *location info* di questo elemento. L'aggiunta di questo elemento a info incrementa la dimensione di info; dunque un blocco del disco

addizionale può dover essere allocato a `info`. Questo coinvolgerà l'aggiornamento della FMT di `info` e la dimensione di `info` nel relativo elemento della directory root [passi 1b e 1c delle azioni per la `close`].



**Figura 13.21** Strutture dati del file system (a) prima e (b) dopo la `close`.

### 13.10 Semantiche della condivisione dei file

Come discusso nel Paragrafo 13.6, il proprietario di un file può autorizzare alcuni utenti ad accedere al file. I processi creati da utenti autorizzati possono leggere, scrivere o eseguire il file in accordo ai privilegi di accesso concessi. Il file system fornisce due metodi di condivisione dei file in modo che i processi possano scegliere quello che consente loro di collaborare in modo efficace:

| Modalità                                | Descrizione  |
|---|--|
| File non modificabile                   | Il file da condividere non può essere modificato da nessun processo.   |
| File modificabile a singola immagine    | Tutti i processi che condividono un file "vedono" la stessa immagine del file, ovvero, hanno una vista identica dei dati del file. In questo modo, le modifiche fatte da un processo sono immediatamente visibili agli altri processi che utilizzano il file.  |
| File modificabile con immagini multiple | I processi che condividono un file possono "vedere" immagini differenti del file. In questo modo, le modifiche fatte da un processo possono non essere visibili a qualche processo concorrente. Il file system può mantenere molte immagini di un file, o può riunirle in qualche modo per creare una singola immagine quando i processi chiudono il file. |

**Tabella 13.4** Modalità di condivisione concorrente dei file.

- *condivisione sequenziale*: i processi accedono a un file condiviso uno dopo l'altro. In questo modo, le modifiche al file fatte da un processo, se ve ne sono, sono visibili ai processi che accedono al file successivamente;
- *condivisione concorrente*: due o più processi accedono al file nello stesso intervallo di tempo.

Le *semantiche della condivisione dei file* sono un insieme di regole che determinano il modo in cui i risultati delle manipolazioni del file eseguite da processi concorrenti sono visibili a tutti.

La condivisione sequenziale di un file può essere implementata mediante il campo `lock` dell'elemento della directory relativo al file (Figura 13.6). Se il campo `lock` ha valore "reset", un'operazione `open` avrebbe successo e cambierebbe il valore a "set"; altrimenti, l'operazione `open` fallirebbe e dovrebbe essere ripetuta. Un'operazione `close` può

cambiare il valore del *lock* a “reset”.

Per facilitare la condivisione concorrente di un file, il file system deve garantire che le attività di elaborazione del file da parte dei processi non interferiscano tra loro. Di conseguenza, crea un FCB separato per ogni processo seguendo la procedura del Paragrafo 13.9.1 ogni volta che un file viene aperto. Diversi FCB possono così essere creati per la condivisione concorrente di *alpha*. Usiamo la notazione  $fcb_{\alpha}^{P_2}$  per il FCB di *alpha* creato per il processo  $P_1$ . La [Tabella 13.4](#) riassume tre modi di condivisione concorrente dei file disponibili nei file system.

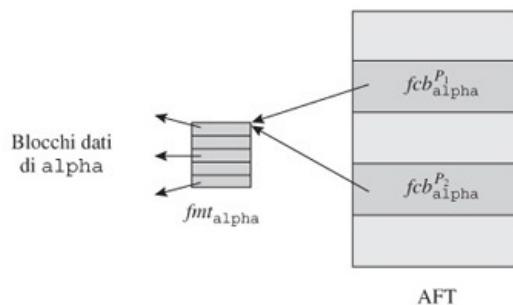
### Condivisione di file non modificabili

Quando il file *alpha* viene creato *come file non modificabile*, nessuno dei processi che lo condivide può modificarlo. Dunque i processi che condividono il file *alpha* sono indipendenti l’uno dall’altro. La creazione di un  $fcb_{\alpha}$  per ogni processo è sufficiente a implementare questa forma di condivisione.

### Condivisione di file modificabili con immagine singola

Una singola copia del file è condivisa dai processi che vi accedono. Dunque le modifiche fatte da un processo sono immediatamente visibili agli altri processi. Per implementare questa forma di condivisione, è essenziale che una singola copia della FMT venga usata da tutti i processi che condividono il file. Dunque è meglio mantenere un puntatore alla FMT, piuttosto che la stessa FMT, in un FCB.

La [Figura 13.22](#) mostra la condivisione concorrente del file *alpha* utilizzando questa configurazione. I FCB  $fcb_{\alpha}^{P_1}$  e  $fcb_{\alpha}^{P_2}$  vengono creati quando *alpha* è aperto dai processi  $P_1$  e  $P_2$ . Entrambi i FCB puntano alla stessa copia di  $fmt_{\alpha}$ . Ogni FCB contiene l’indirizzo del prossimo record cui un processo avrà accesso. Se l’insieme dei record elaborati da  $P_1$  e  $P_2$  si sovrappongono, le rispettive modifiche saranno visibili a entrambi. Si possono verificare delle race condition in queste situazioni per cui le modifiche fatte dai processi potrebbero essere perse. Un tipico file system non fornisce nessun mezzo per affrontare questo problema; a questo scopo i processi devono sviluppare le proprie convenzioni di sincronizzazione. Il file system Unix supporta i file modificabili con immagine singola; affronteremo la *semantica della condivisione dei file in Unix* nel Paragrafo 13.14.1.



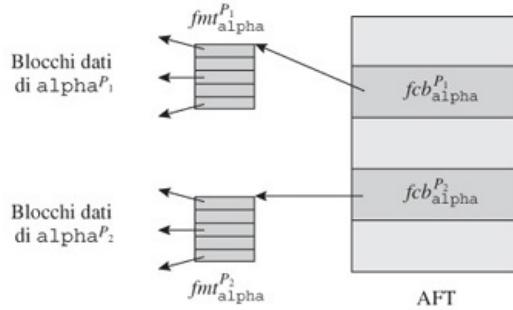
**Figura 13.22** Condivisione concorrente di un file modificabile con immagine singola da parte dei processi  $P_1$  e  $P_2$ .

### Condivisione di file con immagine multipla

Quando un file *alpha* modificabile con immagine multipla viene condiviso da diversi processi, ogni processo che modifica il file crea una nuova versione di *alpha* distinta dalle versioni create dagli altri processi concorrenti. In questo schema, ci deve essere un  $fmt_{\alpha}$  distinto per ogni FCB e ogni FMT deve puntare a una copia esclusiva del file. Questa condizione è implementata al meglio effettuando una copia di *alpha* (e della FMT) per ogni processo che accede a esso in maniera concorrente.

La [Figura 13.23](#) illustra la configurazione per implementare i file modificabili con immagine multipla. I processi  $P_1$  e  $P_2$  sono coinvolti nella modifica di *alpha*.  $\alpha^{P_1}$  rappresenta la copia di *alpha* fatta per il processo  $P_1$ . L’elaborazione eseguita da  $P_1$

utilizza  $fcb_{\alpha}^{P_1}$  e  $fmt_{\alpha}^{P_1}$  per accedere ad  $\alpha^{P_1}$ , mentre l'elaborazione eseguita da  $P_2$  utilizza  $fcb_{\alpha}^{P_2}$  e  $fmt_{\alpha}^{P_2}$  per accedere ad  $\alpha^{P_2}$ .  $\alpha^{P_1}$  e  $\alpha^{P_2}$  sono due versioni di  $\alpha$ . Per arrivare a un unico schema di implementazione, la semantica di condivisione dei file deve specificare come i processi accedono ad  $\alpha$  in lettura, ovvero a quale versione di  $\alpha$  hanno accesso.



**Figura 13.23** Condivisione concorrente di un file modificabile con immagine multipla da parte dei processi  $P_1$  e  $P_2$ .

La condivisione di file modificabili con immagine multipla presenta delle caratteristiche speciali che possono non essere valide o applicabili in molte applicazioni. Dunque possono essere usate solo in applicazioni in cui ha un significato la presenza di versioni multiple dovuta agli aggiornamenti concorrenti. Discutiamo un tipo di semantica per i file modificabili con immagine multipla, chiamata *semantica della sessione*, nel Paragrafo 19.3. Sfortunatamente, le semantiche della condivisione di file modificabili con immagine multipla sono difficili da comprendere e da implementare. Dunque il loro utilizzo non è molto comune.

## 13.11 Affidabilità del file system

L'affidabilità del file system è il grado di funzionamento corretto di un file system, anche quando si verificano malfunzionamenti come la corruzione dei dati nei blocchi del disco e fault del sistema dovuti a interruzioni di corrente. I due aspetti principali dell'affidabilità del file system sono:

- garantire la correttezza della creazione, cancellazione e aggiornamento dei file;
- prevenire la perdita dei dati contenuti nei file.

Il primo riguarda la consistenza e la correttezza dei metadati, ovvero i dati di controllo del file system, mentre il secondo riguarda la consistenza e la correttezza dei dati memorizzati nei file.

La letteratura relativa all'affidabilità distingue tra i termini *fault* (o guasto) e *failure* (o insuccesso). Un fault è un difetto in qualche parte del sistema. Un failure è un comportamento erroneo, o che differisce dal comportamento atteso. L'occorrenza di un fault causa un failure. In questo senso la corruzione di un blocco del disco dovuta a una testina del disco danneggiata o a una mancanza di corrente è un guasto, mentre l'impossibilità del file system di leggere un blocco guasto è un failure. Il [Capitolo 19](#) discute questi termini in maniera formale.

### 13.11.1 Perdita di consistenza del file system

La *consistenza del file system* implica la correttezza dei metadati e il corretto funzionamento del file system. La perdita di consistenza si verifica se i metadati del file system vengono persi o danneggiati. È interessante vedere come ciò si possa verificare. Si consideri il funzionamento di un processo che aggiorna un file  $\alpha$ . Per assicurare un funzionamento efficiente, il file system mantiene alcuni metadati in memoria. In questo modo, il  $fcb_{\alpha}$  (presente nella active file table), parte della  $fmt_{\alpha}$  e parte della mappa di stato del disco o della free list si trovano in memoria. Alcuni di questi metadati, come

la  $fmt_{\alpha\beta\gamma}$ , vengono scritti su un disco quando  $\alpha\beta\gamma$  viene chiuso. Inoltre, il file system periodicamente può copiare la mappa di stato del disco o la free list sul disco. Ciò nonostante, i metadati vengono modificati costantemente, per cui le copie sul disco dei metadati generalmente non contengono informazioni aggiornate durante il funzionamento del sistema. Quando si verifica una mancanza di corrente elettrica, i metadati mantenuti in memoria verranno persi e quando si verifica un malfunzionamento del disco i metadati memorizzati sul disco verranno persi. Queste situazioni possono condurre a uno o più dei seguenti problemi.

1. Alcuni dati del file  $\alpha\beta\gamma$  possono essere persi.
2. Parte del file  $\alpha\beta\gamma$  può risultare inaccessibile.
3. I contenuti di due file vengono confusi tra loro.

È facile visualizzare una situazione del primo tipo. Per esempio, si supponga che si verifichi un guasto dopo che un nuovo blocco del disco è stato aggiunto al file  $\alpha\beta\gamma$ . La copia sul disco della  $fmt_{\alpha\beta\gamma}$  non conterrà l'id di questo blocco per cui i dati nel blocco appena aggiunto andranno persi dopo il verificarsi del malfunzionamento. Il secondo e il terzo tipo di situazione possono verificarsi nei file system che non adottano alcuna tecnica di affidabilità. Illustreremo queste situazioni in un file system che usa l'allocazione concatenata dello spazio sul disco e adotta l'Algoritmo 13.1 per aggiungere un nuovo blocco a un file. Il terzo tipo di situazione può anche verificarsi in un file system che utilizza l'allocazione indicizzata dello spazio su disco.

**Algoritmo 13.1 Aggiungi il blocco  $d_j$  tra i blocchi  $d_1$  e  $d_2$**

**Input:**

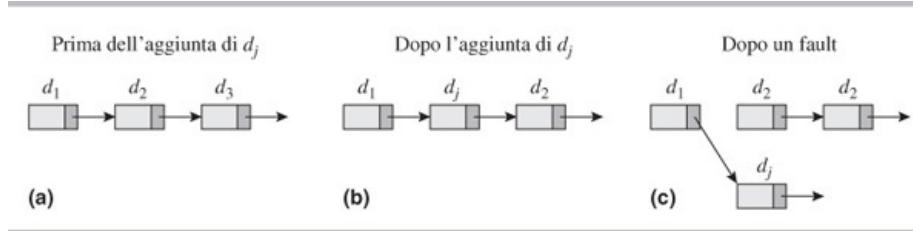
```

 $d_1, d_2, d_j$  : record
    next : ...; { id del prossimo blocco }
    data : ...;
end

1.  $d_j.next := d_1.next$ ;
2.  $d_1.next := \text{indirizzo}(d_j)$ ;
3. write  $d_1$  sul disco;
4. write  $d_j$  sul disco.

```

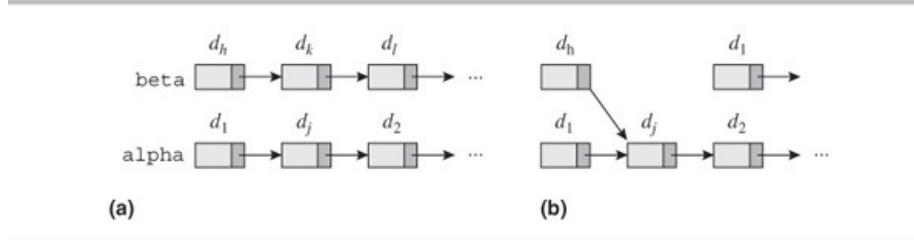
L'Algoritmo 13.1 aggiunge un nuovo blocco  $d_j$  tra i blocchi  $d_1$  e  $d_2$  del file. La [Figura 13.24](#) illustra come le parti del file  $\alpha\beta\gamma$  possano diventare inaccessibili in seguito a un malfunzionamento. Le [Figure 13.24 \(a\)](#) e [\(b\)](#) mostrano il file prima e dopo una normale esecuzione dell'algoritmo. La [Figura 13.24\(c\)](#) mostra il file nel caso in cui si verifichi un malfunzionamento tra i passi 3 e 4 dell'Algoritmo 13.1. I nuovi contenuti devono essere scritti nel blocco  $d_1$ , non nel blocco  $d_j$ . Dunque  $d_1.next$  punta a  $d_j$ , mentre  $d_j$  non contiene metadati corretti nel suo campo *next*. I blocchi del disco  $d_2, d_3, \dots$  non sarebbero più accessibili come parti del file.



**Figura 13.24** Inconsistenze nei metadati dovuti a malfunzionamenti: (a)-(b) prima e dopo l'aggiunta di  $d_j$  durante il normale funzionamento; (c) dopo un malfunzionamento.

I contenuti dei due file possono essere confusi se il file system scrive i metadati sul

disco solo al momento della chiusura del file e non dopo ogni operazione su file. Si consideri il seguente caso: un processo  $P_1$  cancella un blocco del disco  $d_k$  da un file beta.  $d_k$  sarà restituito alla free list (o sarà segnato come libero nella mappa di stato del disco). Ora il processo  $P_2$  aggiunge un nuovo record al file alpha. Il file system alloca un nuovo blocco  $d_j$  e lo aggiunge prima del blocco  $d_2$  nel file alpha [Figura 13.25(a)]. A questo punto, si consideri la situazione in cui  $d_j = d_k$  e nel sistema si verificano i seguenti eventi.



**Figura 13.25** I file alpha e beta. (a) dopo l'aggiunta di  $d_j$  durante il normale funzionamento; (b) se  $d_j = d_k$ , alpha viene chiuso e si verifica una mancanza di corrente.

1. Il file *alpha* viene chiuso.
2. Il file system aggiorna la copia del disco del file *alpha*. Questo implica l'aggiunta del blocco  $d_j$  ad *alpha*.
3. Si verifica una mancanza di corrente elettrica.

Si noti che il file *beta* non era stato chiuso prima della mancanza di corrente elettrica, per cui il disco contiene una vecchia copia di *beta* che contiene il blocco  $d_k$  e la nuova copia di *alpha* che contiene il blocco  $d_j$ . Poiché  $d_j = d_k$ , *alpha* e *beta* ora condividono il blocco  $d_j$  e tutti gli altri blocchi da esso accessibili [Figura 13.25(b)]. Tutti i blocchi del file *beta* che erano accessibili tramite  $d_k$ , ovvero il blocco  $d_1$  e gli altri blocchi da esso accessibili ora sono inaccessibili. In effetti, alcuni dati sono in comune ai file *alpha* e *beta*, mentre alcuni dati di *beta* sono stati persi.

### 13.11.2 Approcci all'affidabilità del file system

Seguendo i due approcci descritti nella Tabella 13.5, i sistemi operativi garantiscono che i file utente siano memorizzati in maniera affidabile durante un periodo di tempo. Il *recupero* è un approccio classico attuato quando si riscontra un malfunzionamento. L'azione effettuata è quella di ripristinare i dati e i metadati del file system relativi a un precedente stato consistente. Il file system a questo punto riprende l'esecuzione a partire da quest'ultimo stato. In questo modo, i comportamenti anomali si verificano, ma il funzionamento del sistema è rettificato quando viene rilevato il malfunzionamento. La *fault tolerance*, d'altra parte, garantisce il corretto funzionamento del file system in tutte le situazioni, ovvero assicura che i malfunzionamenti non portino a situazioni critiche. Questa caratteristica è ottenuta mediante l'uso di tecniche speciali.

Per comprendere la differenza tra i due approcci, si consideri l'esempio di un blocco del disco che non sia più leggibile. L'impossibilità del file system di leggere il blocco porta a una mancanza di informazioni non più recuperabili. Nel caso dell'approccio con recupero, qualora si verificasse un guasto, i dati nel blocco sarebbero ripristinati a un valore precedente. Nel caso della fault tolerance, ogni dato sarebbe memorizzato in due blocchi - un blocco primario e un blocco alternativo. Se si verificasse un guasto durante la lettura del blocco primario, il file system leggerebbe automaticamente il blocco alternativo. Naturalmente, la fault tolerance non è assoluta. Il sistema può gestire solo i guasti per i quali è stato progettato. Per esempio, quando un dato è memorizzato in due blocchi, il sistema può gestire un guasto nel blocco primario, ma non i guasti nel blocco primario e in quello alternativo.

#### Tecniche di ripristino

Lo *stato* del file system in un istante di tempo  $t_i$  è l'insieme di tutti i dati e metadati nel file system nell'istante  $t_i$ . Un *backup* del file system è una copia dello stato del file system. Per supportare il ripristino, il file system produce periodicamente dei backup

durante il suo funzionamento in modo da poterli utilizzare in caso di guasti per effettuare il ripristino. Sia  $t_{lb}$  il momento in cui è stato prodotto l'ultimo backup.

| Approccio       | Descrizione   |
|-----------------|---|
| Recupero        | Ripristinare i dati e i metadati del file system a un precedente stato consistente.   |
| Fault tolerance | Prevenzione contro la perdita di consistenza dei dati e dei metadati in seguito a guasti, in modo che il funzionamento del sistema sia corretto in ogni situazione, ovvero, non si verificano perdite o corruzioni di dati. |

**Tabella 13.5** Approcci all'affidabilità del file system.

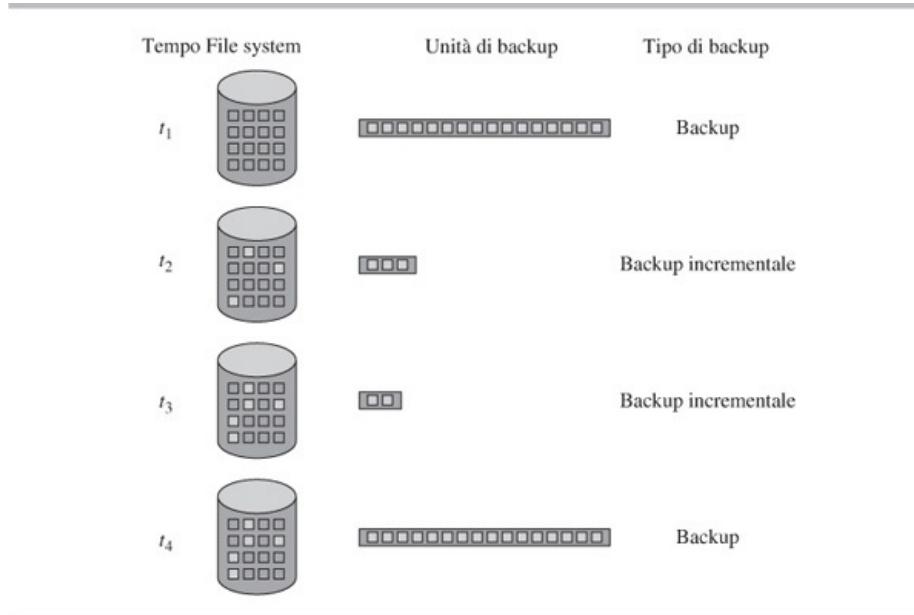
In caso di guasto, per esempio, al tempo  $t_f$ , il file system viene ripristinato allo stato registrato nell'ultimo backup, ma non viene persa la possibilità di continuare a utilizzare i file, anche se non aggiornati. Le modifiche ai file effettuate tra  $t_{lb}$  e  $t_f$  andranno perse; le operazioni che hanno generato queste modifiche devono essere ripetute dopo il ripristino. Il ripristino mediante backup in questo modo coinvolge due tipi di overhead - l'overhead per la creazione dei backup e l'overhead per la ripetizione dei comandi.

L'overhead dovuto alla rielaborazione nel caso di recupero può essere ridotto mediante una combinazione di backup e backup incrementali del file system. Un *backup incrementale* contiene solo le copie di quei file o blocchi del disco che risultano modificati dopo la creazione dell'ultimo backup o backup incrementale. Il file system crea backup per lunghi periodi di tempo, per esempio un giorno, pochi giorni o una settimana. I backup incrementali vengono creati per intervalli più brevi e sono scartati quando viene creato il backup successivo. Per esempio, un backup incrementale può essere creato quando un processo chiude un file dopo averlo aggiornato; il backup incrementale conterebbe solo una copia di quel file. L'uso dei backup incrementali fa aumentare l'overhead dell'attività di backup. Anche l'overhead di spazio è alto poiché i backup e i backup incrementali coesistono e alcuni file si trovano in più di un backup incrementale. Tuttavia, l'overhead di rielaborazione è basso per le seguenti ragioni: dopo un crash, il sistema può essere ripristinato dall'ultimo backup e successivamente vengono rielaborati i backup incrementali nell'ordine in cui sono stati creati. Questo consente di recuperare tutti i file le cui modifiche sono state completate prima della creazione dell'ultimo backup incrementale. Solo le attività di elaborazione dei file che erano ancora in esecuzione al momento del guasto dovranno essere ripetute.

Per ridurre l'overhead di recupero, il file system potrebbe essere ripristinato elaborando tutti i backup incrementali e l'ultimo backup nell'ordine inverso, avendo cura di non ripristinare un file che è già stato ripristinato da un backup incrementale successivo. Questo approccio ridurrebbe l'overhead ripristinando ogni file esattamente una volta. Tuttavia, sarebbe efficace solo se i metadati del file system fossero consistenti al momento del guasto.

#### Esempio 13.10 - Ripristino in un file system

La [Figura 13.26](#) illustra un sistema in cui i backup sono stati presi al tempo  $t_1$  e  $t_4$  e i backup incrementali sono stati presi al tempo  $t_1$  e  $t_2$  e due blocchi del disco sono stati modificati tra il tempo  $t_2$  e  $t_3$ . Se si verifica un guasto dopo  $t_4$ , il sistema viene ripristinato allo stato registrato nel backup preso a  $t_4$ . Tuttavia, se si verificasse un guasto tra  $t_3$  e  $t_4$ , il sistema sarebbe ripristinato usando il backup preso a  $t_1$  e i backup incrementali presi a  $t_2$  e  $t_3$ .



**Figura 13.26** Backup e backup incrementale in un file system.

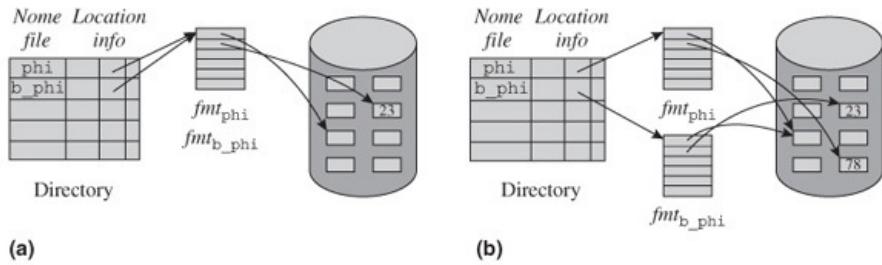
### **Creazione di backup**

La problematica principale nella creazione dei backup è la consistenza dei metadati registrati in un backup. Si consideri il seguente scenario durante il funzionamento di un sistema.

1. La struttura dati che mantiene la free list viene scritta nel backup.
2. Un record viene aggiunto al file `phi`, cosa che richiede l'allocazione a `phi` di un blocco dalla free list.
3. Il file `phi` a questo punto viene scritto nel backup.

Le registrazioni della free list e del file `phi` nel backup sarebbero mutuamente inconsistenti. Questo potrebbe portare a una confusione dei dati nei file, come discusso nel Paragrafo 13.11.1. Problemi simili si verificherebbero anche se queste tre azioni fossero eseguite in ordine inverso. Le inconsistentez dei metadati potrebbero essere prevenute congelando tutte le attività nel file system durante la creazione di un backup; tuttavia, questo metodo è intrusivo e causerebbe ritardi nei processi. Un'alternativa è creare un backup durante il normale funzionamento del sistema, ma di usare alcune semplificazioni come non scrivere la free list nel backup. Quando lo stato del file system viene ripristinato da questo backup, il file system potrebbe fare la scansione completa del disco e costruire una nuova free list. Tuttavia, in questo schema i file sarebbero registrati nel backup in tempi differenti, per cui soffrirebbero di perdita dei dati in misura differente se il file system fosse ripristinato usando il backup. Un'altra problematica riguarda il backup di un file elaborato quando il backup è iniziato - o il suo backup dovrebbe essere ritardato fino al termine dell'elaborazione, o l'utente non saprebbe precisamente in che misura sia andata persa l'elaborazione nel caso di ripristino del file system con il backup. Un backup incrementale creato quando il file viene chiuso non affronta nessuno di questi problemi di consistenza poiché solo i file modificati sono scritti nel backup, per cui i metadati del file system come le free list non sarebbero scritti al suo interno.

Qual è l'overhead per la creazione di un backup? Quando lo spazio sul disco era costoso, i backup tipicamente erano creati su dispositivi di I/O lenti quali le unità a nastro; tuttavia, lo spazio sul disco oggi è molto meno costoso, per cui è possibile creare i backup sui dischi. Quando viene utilizzata l'allocazione indicizzata dello spazio sul disco, è possibile creare un backup sul disco di un file in modo economico mediante una tecnica simile alla tecnica *copy-on-write* della memoria virtuale. La [Figura 13.27](#) illustra questa tecnica.



**Figura 13.27** Creazione di un backup: (a) dopo il backup del file *phi*; (b) quando *phi* viene modificato.

Quando si deve effettuare il backup del file *phi*, il file system crea una copia dell'elemento della directory di *phi* e dà il nome al nuovo file in maniera appropriata, per esempio *b\_phi*. A questo punto, il puntatore FMT di *phi* e *b\_phi* sono identici [Figura 13.27(a)], per cui il file *b\_phi* è una copia di *phi* come desiderato. Se il contenuto del secondo blocco del disco allocato a *phi* cambia da 23 a 78 in seguito a una modifica del file, il file system esegue le seguenti azioni [Figura 13.27(b)].

1. Se i puntatori FMT di *phi* e *b\_phi* sono identici, effettua una copia della FMT e fa puntare l'elemento della directory di *b\_phi* alla copia.
2. Alloca un nuovo blocco al file *phi*.
3. Cambia il puntatore appropriato in *fmt\_phi* per puntare al nuovo blocco.
4. Scrive il nuovo contenuto nel nuovo blocco.

In questo modo, sarebbero copiati solo la FMT e il blocco del disco il cui contenuto è aggiornato dopo la creazione del backup. Questa organizzazione preserva sia spazio su disco che tempo.

### Tecniche che implementano la fault tolerance

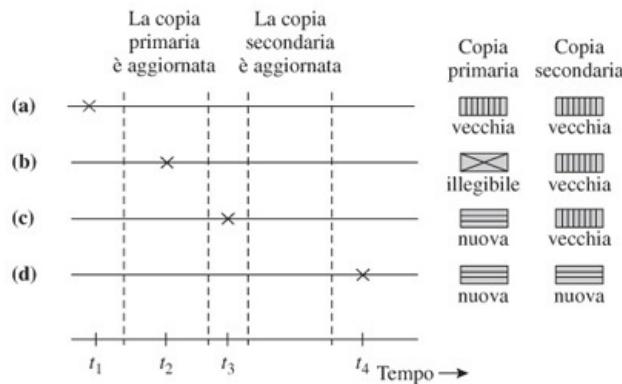
L'affidabilità del file system può essere migliorata prendendo due precauzioni - prevenendo la perdita dei dati o metadati dovuta a un malfunzionamento del dispositivo di I/O e prevenendo l'inconsistenza dei metadati dovuta a guasti. Queste precauzioni sono implementate usando le tecniche di fault tolerance, rispettivamente, di memorizzazione stabile e azioni atomiche.

### Memorizzazione stabile

Lamson (1981) propose la tecnica della memorizzazione ridondante dei dati per garantire l'affidabilità. Viene chiamata *memorizzazione stabile* poiché può consentire un guasto nella registrazione di un dato. Vengono mantenute due copie di un record, chiamate la sua copia *primaria* e *secondaria*. Un'operazione di write aggiorna entrambe le copie - la copia primaria viene aggiornata per prima, poi viene aggiornata la copia secondaria. Un'operazione di lettura accede al blocco del disco contenente la copia primaria. Se non è leggibile, si utilizza la copia secondaria. Dal momento che si presume che si possano verificare solo singoli guasti, uno dei blocchi sicuramente conterrà dati leggibili.

La Figura 13.28 illustra il funzionamento della tecnica di memorizzazione stabile se si verifica un guasto, rispettivamente, al tempo  $t_1$ ,  $t_2$ ,  $t_3$  o  $t_4$  mentre un processo  $P_i$  sta eseguendo un'operazione di aggiornamento su alcuni dati  $D$ . Le Parti (a)-(d) mostrano il grafico dei tempi e i valori nelle copie primaria e secondaria di  $D$  quando si verifica il guasto. Nella Parte (a), un guasto si verifica al tempo  $t_1$ , ovvero prima che la copia primaria venga aggiornata. Dunque la copia primaria, contenente il vecchio valore del dato, è accessibile dopo il guasto. Nella Parte (b), un guasto si verifica mentre la copia primaria è in aggiornamento, per cui la copia primaria diventa illeggibile. Il vecchio valore del dato è accessibile dalla copia secondaria. Nella Parte (c), un guasto si verifica dopo l'aggiornamento della copia primaria ma prima che la copia secondaria venga aggiornata. I nuovi dati sono accessibili nella copia primaria dopo che si è verificato il guasto. Nella Parte (d), un guasto si verifica dopo che entrambe le copie sono state

aggiornate. Dunque entrambe le copie sono accessibili.



**Figura 13.28** Fault tolerance mediante la tecnica della memorizzazione stabile.

La tecnica della memorizzazione stabile può essere applicata a interi file. (Lampson chiamò questa tecnica disk mirroring; è differente dal *disk mirroring* di cui parleremo nel Paragrafo 14.3). Tuttavia, la memorizzazione stabile incorre in un grande overhead di spazio e tempo, che lo rende non adatto per un uso generale in un file system, per cui i processi lo usano in modo selettivo per proteggere alcuni dati. Inoltre, mentre la memorizzazione stabile garantisce che una copia dei dati sopravviva a un singolo guasto, non può indicare se questo valore è vecchio o nuovo [Parti (a), (d) di [Figura 13.28](#)]. Dunque l'utente non sa se se rieseguire l'operazione di modifica  $P_i$  quando viene ripristinato il funzionamento del sistema. Un'azione atomica risolve questo problema.

### Azioni atomiche

Un'azione può andare a modificare molte strutture dati, per esempio, si consideri l'Algoritmo 13.1 del Paragrafo 13.11.1. Queste strutture dati possono diventare inconsistenti se un guasto interrompe l'esecuzione dell'azione. Un'*azione atomica* è un metodo per evitare gli effetti di un eventuale guasto.

#### Definizione 13.1 Azione atomica

Un'azione che consiste di un insieme di sottoazioni e la cui esecuzione ha una delle due seguenti proprietà:

1. si realizzano gli effetti di tutte le sue sottoazioni, oppure
2. non si realizza l'effetto di alcuna sottoazione.

In questo modo, un'operazione atomica possiede una proprietà *tutto o niente*. Questa proprietà evita l'inconsistenza dei dati quando si verificano i guasti. La consistenza dei metadati di un file system può essere preservata aggiornando tutte le strutture dati del file system mediante azioni atomiche. I database usano un concetto chiamato *transazione atomica* o *transazione di database* che garantisce anche altre proprietà quali la serializzabilità; la nostra trattazione è limitata alle azioni atomiche relative alla realizzazione dell'affidabilità di un file system.

Le sottoazioni in un'azione atomica sono racchiuse tra le istruzioni **begin atomic action** ed **end atomic action**. L'esecuzione dell'azione atomica inizia quando viene eseguita l'istruzione **begin atomic action**. L'azione può terminare in due modi - o fallire o andare a buon fine. Fallisce se perde interesse a continuare la propria esecuzione ed esegue un'istruzione **abort**, o se si verifica un guasto prima dell'esecuzione dell'istruzione **end atomic action**. Se fallisce, lo stato di ogni file o dei metadati usati dovrebbe essere lasciato come si trovava prima dell'esecuzione dell'istruzione **begin atomic action**.

Un'operazione atomica ha esito positivo quando esegue l'istruzione **end atomic action**. In questo momento l'azione si definisce *commit* (compiuta). Tutte le modifiche effettuate con il *commit* non saranno perse in caso di guasto.

La [Figura 13.29](#) mostra l'Algoritmo 13.1 del Paragrafo 13.11.1 codificato come azione atomica chiamata *aggiunta\_blocco*. Differisce dall'Algoritmo 13.1 solo nell'uso delle istruzioni **begin atomic action** ed **end atomic action**. Se l'azione atomica *aggiunta\_blocco* è commit, il blocco  $d_j$  viene aggiunto al file *alpha* ed *alpha* si compone dei blocchi  $\dots d_1, d_j, d_2, \dots$ . Se fallisce, il blocco  $d_j$  non viene aggiunto al file *alpha*; ovvero, *alpha* continua a essere composto dei blocchi  $\dots d_1, d_2, \dots$ . Per cui evita il problema descritto nel Paragrafo 13.11.1 e illustrato in [Figura 13.24](#).

---

```

begin atomic action add_a_block;
   $d_j.next := d_1.next;$ 
   $d_1.next := address(d_j);$ 
  write  $d_1;$ 
  write  $d_j;$ 
end atomic action add_a_block;

```

---

**Figura 13.29** Azione atomica *add\_a\_block*.

Le azioni atomiche possono essere implementate in molti modi. In un approccio implementativo, i file o i metadati non sono aggiornati durante l'esecuzione dell'azione atomica. Sono aggiornati solo dopo che l'azione atomica è commit. Questa organizzazione automaticamente tollera i guasti che si verificano prima che un'azione atomica sia commit poiché non vengono effettuate modifiche nel file. Per cui implementa la parte "niente" della proprietà tutto o niente. Per implementare la parte "tutto" della proprietà tutto o niente, è necessario assicurare che tutte le modifiche saranno effettuate anche se si verifica un guasto. Per garantire questa proprietà vengono utilizzate due strutture dati chiamate *lista delle intenzioni* e *commit flag*. Entrambe le strutture dati sono mantenute in memorizzazione stabile per proteggerle contro la corruzione e la perdita in seguito a guasti.

Ogni volta che l'azione atomica modifica un file o i metadati, il file system crea un elemento del tipo (*<id del blocco del disco>*, *<nuovo contenuto>*) nella lista delle intenzioni per aggiornare il file quando l'azione atomica è commit. Questa azione viene chiamata *elaborazione del commit*. La commit flag contiene due campi, l'*id della transazione* e il *valore*. Questa flag viene creata quando viene eseguita l'istruzione **begin atomic action** di un'azione atomica  $A_i$  e i suoi campi sono inizializzati, rispettivamente, ad  $A_i$  e "not committed". Il valore nella commit flag viene cambiato a "committed" quando viene eseguita l'istruzione **end atomic action**. La flag viene distrutta dopo che tutte le modifiche descritte nella lista delle intenzioni sono state eseguite.

Se si verifica un malfunzionamento, quando il file system riprende l'esecuzione verifica la presenza della commit flag. Se esiste una commit flag per  $A_i$  e ha valore "not committed", il file system semplicemente elimina la commit flag e la lista delle intenzioni ed esegue l'operazione atomica  $A_i$  nuovamente partendo con l'istruzione **begin atomic action**. L'esistenza di una commit flag per  $A_i$  con valore "committed" implica che l'elaborazione del commit di  $A_i$  era in esecuzione quando si è verificato un guasto che ha portato al malfunzionamento. Poiché non è possibile sapere se ogni elemento della lista delle intenzioni è stato elaborato prima del guasto, l'elaborazione della commit viene ripetuta interamente.

Se si verificano dei guasti durante l'elaborazione della commit, alcuni elementi della lista delle intenzioni possono essere elaborati più volte. Tuttavia, ciò non pone alcun problema di consistenza dei dati poiché l'operazione di scrittura di *<nuovo contenuto>* in *<id del blocco del disco>* è un'operazione *idempotente*, che presenta la proprietà per cui eseguirla molte volte ha lo stesso effetto che eseguirla una sola volta. Il seguente algoritmo riassume tutte le azioni che riguardano l'implementazione di un'azione atomica.

### Algoritmo 13.2 Implementazione di un'azione atomica

#### 1. Esecuzione di un'azione atomica $A_i$ :

- quando viene eseguita l'istruzione **begin atomic action**, si crea una *commit flag* e una *lista delle intenzioni* in memorizzazione stabile e le inizializza come segue:

- ```

commit flag := (Ai, "not committed");
lista delle intenzioni := "vuota";
b. per ogni modifica al file eseguita da una sottoazione, aggiunge una coppia (d, v)
alla lista delle intenzioni, dove d è l'id di un blocco del disco e v è il suo contenuto;
c. quando viene eseguita l'istruzione end atomic action, imposta il valore della
commit flag di Ai, a "committed" ed esegue il Passo 2.

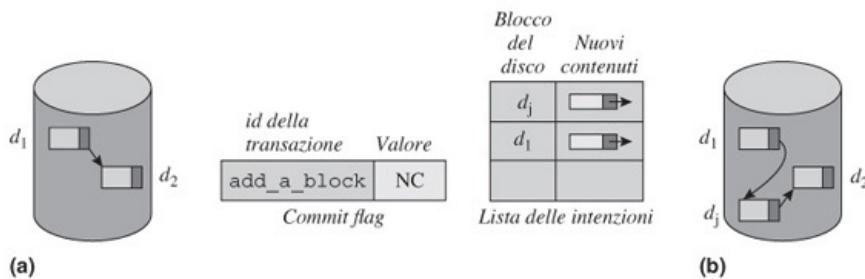
2. Esecuzione della commit:
a. per ogni coppia (d, v) nella lista delle intenzioni, scrive v nel blocco con id d;
b. cancella la commit flag e la lista delle intenzioni.

3. Nel caso di ripristino dopo un malfunzionamento:
se esiste la commit flag per un'azione atomica Ai,
a. se il valore della commit flag è "not committed": cancella la commit flag e la lista
delle intenzioni. Esegue nuovamente l'azione atomica Ai;
b. esegue il Passo 2 se il valore nella commit flag è "committed."

```

### Esempio 13.11 - Implementazione di un'azione atomica

La [Figura 13.30\(a\)](#) mostra il file alpha, la commit flag e la lista delle intenzioni quando viene applicato l'Algoritmo 13.2 all'azione atomica *add\_a\_block* di [Figura 13.29](#). I nuovi contenuti dei blocchi d<sub>j</sub> e d<sub>1</sub> vengono mantenuti nella lista delle intenzioni fino all'elaborazione della commit. L'atomicità dell'azione è garantita come segue: se si verifica un guasto durante il Passo 1 dell'algoritmo, nessuna modifica viene eseguita sul disco. Dunque il file contiene la sequenza originale dei blocchi d<sub>1</sub>, d<sub>2</sub>, .... Un guasto che si verifica al Passo 2 non può danneggiare né la commit flag né la lista delle intenzioni poiché queste strutture dati sono registrate in memorizzazione stabile. In questo modo, l'elaborazione della lista delle intenzioni alla fine viene completata; il file contiene la sequenza dei blocchi d<sub>1</sub>, d<sub>j</sub>, d<sub>2</sub>... al termine dell'elaborazione della commit, come mostrato in [Figura 13.30\(b\)](#).



**Figura 13.30** (a) Prima e (b) dopo l'esecuzione della commit. (Nota: NC significa *not committed*.)

## 13.12 Journaling file system

Come discusso nel Paragrafo 13.11.1, un file system, durante l'esecuzione, mantiene in memoria una parte dei file dati e dei metadati: i file control block, le file map table e le free list. Quando l'esecuzione di un file system viene terminata da un amministratore di sistema, il file system copia tutti i dati e i metadati dalla memoria al disco, in modo che la copia sul disco sia completa e consistente. Tuttavia, quando si verifica una mancanza di corrente elettrica, o quando il sistema viene spento all'improvviso, il file system non ha l'opportunità di copiare i file dati e i metadati dalla memoria al disco. Questo spegnimento viene chiamato spegnimento *sporco* (unclean shutdown) e causa una perdita dei dati dei file e dei metadati mantenuti in memoria.

Tradizionalmente, i file system utilizzavano delle tecniche di ripristino per proteggersi contro la perdita di dati e metadati poiché erano molto semplici da implementare. Per questo motivo, si creavano backup periodici e i file erano ripristinati a partire dalle copie di backup nel momento in cui si verificavano dei malfunzionamenti. I metadati venivano ripristinati con ricerche laboriose per trovare e correggere le inconsistenze. L'uso delle tecniche di ripristino imponeva un piccolo overhead durante il normale funzionamento

del sistema. Quando si verificava un malfunzionamento, tuttavia, l'overhead per la correzione delle inconsistenze era elevato e inoltre il sistema non era disponibile durante il ripristino. Come esempio, si consideri cosa succedeva quando un sistema Unix che utilizzava il file system ext2 veniva spento all'improvviso. Al momento del riavvio, il file system rilevava lo spegnimento non corretto e dunque i suoi metadati verosimilmente erano inconsistenti. A questo punto lanciava il programma `fsck` per ripristinare i metadati. `fsck` cercava ogni struttura dati del file system sul disco e cercava di correggere ogni inconsistenza rilevata. L'esecuzione del SO era ritardata durante l'esecuzione di `fsck`.

Un file system moderno utilizza tecniche di fault tolerance in modo da poter riprendere l'esecuzione velocemente dopo uno spegnimento improvviso. Un *journaling file system* implementa la fault tolerance mantenendo un *journal* (diario giornaliero), che ricorda la lista delle intenzioni utilizzata per implementare le azioni atomiche (Paragrafo 13.11.2). Il file system memorizza le azioni che si accinge a eseguire nel journal prima di eseguirle effettivamente. Quando l'esecuzione del file system viene ripristinata dopo uno spegnimento improvviso, il file system consulta il journal per identificare le azioni non ancora eseguite e le esegue, garantendo in questo modo la correttezza dei dati dei file e dei metadati. I file system ext3 di Linux, XFS della Silicon Graphics, JFS della IBM e VxFS della Veritas sono alcuni esempi di journaling file system. L'uso delle tecniche di fault tolerance per proteggere la consistenza sia dei metadati che dei dati genera un elevato overhead - pari all'elaborazione di ogni modifica di file come azione atomica. Dunque un journaling file system offre una serie di modalità journaling; ogni modalità fornisce un differente tipo di protezione dei metadati e dei dati. Un amministratore di sistema può scegliere una modalità journaling da adattare al tipo di affidabilità necessaria nell'ambiente di elaborazione. La [Tabella 13.6](#) descrive tre modalità journaling.

Nella modalità *write behind*, i metadati sono protetti, ma i dati non lo sono. Quando un nuovo dato è aggiunto al file, questa modalità garantisce che il blocco allocato per memorizzare i nuovi dati venga aggiunto nella copia su disco della FMT del file. Tuttavia, non garantisce che i nuovi dati aggiunti al file vengano memorizzati in questi blocchi prima che si verifichi un guasto. Di conseguenza, se si verifica un guasto durante l'elaborazione del file, la copia su disco del file può contenere dati spuri. La modalità *ordered data* evita questo problema garantendo che i dati vengano scritti sul disco prima dei metadati. Tuttavia, quando si utilizza questa modalità, si può creare una situazione in cui i blocchi dove sono scritti i nuovi dati non vengono aggiunti alla file map table. La modalità *full data* protegge sia i metadati che i dati.

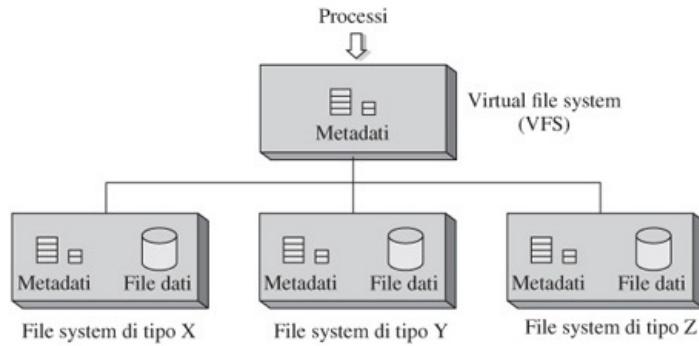
| Modalità     | Descrizione                                                                                                    |
|--------------|----------------------------------------------------------------------------------------------------------------|
| Write behind | Protegge solo i metadati. Non fornisce alcuna protezione per i dati.                                           |
| Ordered data | Protegge i metadati. Protezione limitata per i file dati - vengono scritti prima dei metadati a essi relativi. |
| Full data    | Protegge sia i dati che i metadati.                                                                            |

**Tabella 13.6** Modalità journaling.

### 13.13 File system virtuale

Gli utenti richiedono requisiti differenti a un file system, come convenienza, alta affidabilità, risposte veloci e accesso ai file su altri sistemi. Un singolo file system non può fornire tutte queste caratteristiche, per cui un sistema operativo fornisce un *file system virtuale* (VFS), che facilita l'esecuzione simultanea di diversi file system. In questo modo ogni utente può usare il file system che preferisce.

Un file system virtuale (VFS) è un'astrazione che supporta un modello di file generico. L'astrazione viene implementata da un layer (strato) VFS situato tra un processo e un file system ([Figura 13.31](#)). Il layer VFS ha due interfacce - un'interfaccia con il file system e un'interfaccia con i processi. Ogni file system conforme alle specifiche dell'interfaccia del file system VFS può essere installato per funzionare con il VFS.



**Figura 13.31** File system virtuale.

Questa caratteristica rende facile aggiungere un nuovo file system. L'interfaccia del VFS con il processo fornisce le funzionalità per eseguire le generiche operazioni sui file open, close, read e write e le operazioni mount e umount sul file system. Queste funzionalità sono invocate mediante chiamate di sistema. Il VFS determina a quale file system un file di fatto appartiene e invoca le operazioni open, close, read e write dello specifico file system mediante l'interfaccia del VFS con il file system. Inoltre, richiama le funzioni dello specifico file system per implementare le operazioni di mount e umount.

Tutti i file system in esecuzione nel VFS sono disponibili per essere utilizzati simultaneamente. Nel sistema di [Figura 13.31](#), un processo può usare un file system del tipo X mentre un altro processo simultaneamente utilizza un file system di tipo Y. Il file system virtuale può anche essere usato per comporre un file system eterogeneo. Per esempio, un utente può montare un file system di tipo X in una directory del file system di tipo Y. Questa caratteristica è utile con i dispositivi rimovibili come i CD; consente a un utente di montare il file system presente in un CD nella sua directory corrente e accedere ai file senza preoccuparsi del fatto che i dati sono memorizzati in un formato differente. Questa caratteristica è utile anche in un ambiente distribuito per montare un file system remoto in un file system di un computer. Questa caratteristica è descritta nel [Paragrafo 19.6.1](#).

Come mostrato nel diagramma di [Figura 13.31](#), il file system virtuale non contiene alcun dato. Semplicemente contiene strutture dati che costituiscono i metadati del VFS. Ogni file system contiene i propri metadati e i propri file. La struttura dati chiave usata dal file system virtuale è il *virtual node* (nodo virtuale), generalmente chiamato *vnode*, che contiene le informazioni necessarie per eseguire le operazioni su un file. Può essere visto come un oggetto file con le seguenti tre parti.

- I dati indipendenti dal file system come un id di file unico nel dominio del VFS, che può essere un singolo computer o una rete; il tipo di file, per esempio, i dati o un file speciale; e altri campi come un contatore di open, lock e flag.
- Dati specifici del file system come la file map table.
- Gli indirizzi delle funzioni nel file system che contiene questo file. Queste funzioni implementano le operazioni di open, close, read e write sui file di questo tipo di file.

I sistemi operativi hanno fornito file system virtuali sin dagli anni '90. I sistemi operativi Sun OS e Solaris della Sun, Unix System V version 4, Unix 4.2 BSD e Linux mettono a disposizione un file system virtuale.

## 13.14 Casi di studio

### 13.14.1 File system Unix

Il progetto del file system di Unix è fortemente influenzato dal file system MULTICS. In questo paragrafo descriviamo le caratteristiche importanti comuni a molte versioni di Unix, nel contesto della descrizione generica dell'elaborazione dei file dei [Paragrafi 13.4](#) e [13.8](#).

#### **Inode, descrittore di file e struttura file**

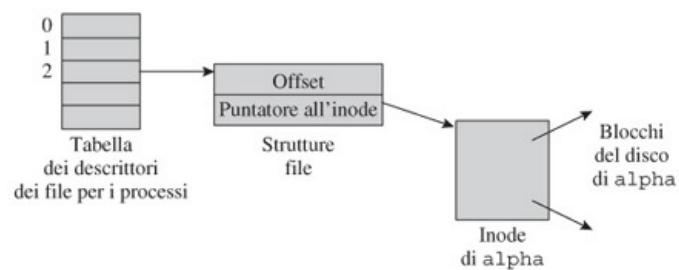
Le informazioni che costituiscono l'elemento della directory relativo a un file in [Figura 13.6](#) in Unix sono divise tra l'elemento della directory e l'*inode* del file. L'elemento della directory contiene solo il nome del file e il numero di inode; la maggior parte delle informazioni relative a un file sono contenute nel suo inode. I file sono considerati come flussi di caratteri a cui si accede in maniera sequenziale. L'amministratore di sistema può specificare una quota disco per ogni utente. Questo impedisce a un utente di occupare uno spazio elevato sul disco rigido oppure per assurdo di occupare uno spazio elevato sul disco rigido oppure, per assurdo, tutto lo spazio disponibile.

La struttura dati inode viene mantenuta sul disco. Alcuni campi contengono le seguenti informazioni.

- Tipo di file, per esempio, directory, link o file speciale.
- Numero di link al file.
- Dimensione del file.
- Id del dispositivo su cui è memorizzato il file.
- Numero seriale dell'inode.
- Id utente e gruppo del proprietario.
- Permessi di accesso.
- Informazione sull'allocazione.

La divisione dell'elemento della directory convenzionale tra l'elemento della directory e l'inode facilita la creazione e la cancellazione dei link. Un file può essere cancellato quando il suo numero di link va a zero. Si noti la similitudine tra i campi dell'inode e quelli del FCB ([Tabella 13.3](#)).

La [Figura 13.32](#) illustra l'organizzazione in memoria durante l'elaborazione di un file. Si compone degli inode, *strutture file* e *descrittori di file*. Una struttura file contiene due campi - la posizione corrente in un file aperto, nella forma di un offset a partire dall'inzio del file; e un puntatore all'inode del file. In questo modo un inode e una struttura file insieme contengono tutte le informazioni necessarie per accedere al file. Un descrittore del file punta alla struttura file. I descrittori di file sono memorizzati in una tabella per ogni processo. Questa tabella ricorda la *active file table* (AFT) descritta nel Paragrafo 13.8.



**Figura 13.32** Struttura dati del file system Unix.

Quando un processo apre un file alpha, l'elemento della directory relativo ad alpha viene localizzato. Una cache di ricerca della directory viene utilizzata per velocizzare questa operazione. Una volta che l'elemento relativo ad alpha è stato localizzato, il suo inode viene copiato in memoria, a meno che non si trovi già in memoria. Viene impostata questa configurazione, mostrata in [Figura 13.32](#), e l'indice del descrittore del file nella tabella dei descrittori, un intero, viene restituito al processo che ha aperto il file. Il processo può utilizzarlo in un modo che ricorda l'uso dell'id interno di un file nella generica organizzazione, dei Paragrafi 13.4 e 13.8.

Quando un processo crea un processo figlio, viene creata una tabella dei descrittori per il processo figlio e i descrittori dei file del processo genitore sono copiati al suo interno. In questo modo più di un descrittore di file può puntare alla struttura di file. I processi che detengono questi descrittori di file condividono l'offset nel file. Una read o una write da parte di un processo modificherà anche l'offset per gli altri processi.

#### **Sematica della condivisione dei file**

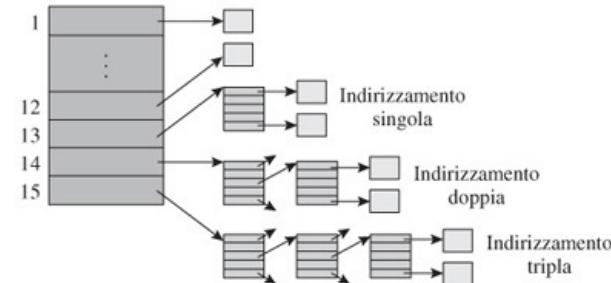
Diversi processi possono aprire indipendentemente lo stesso file. In questo caso,

l'organizzazione di [Figura 13.32](#) viene impostata per ogni processo. In questo modo, due o più strutture file possono puntare allo stesso inode. I processi che utilizzano queste strutture dati hanno il proprio offset nel file, per cui una read o una write da parte di un processo non modifica l'offset usato dagli altri processi.

Unix fornisce la semantica del file modificabile con singola immagine per la condivisione concorrente dei file. Come mostrato in [Figura 13.32](#), ogni processo che apre un file punta alla copia del suo inode mediante il descrittore del file e la struttura file. In questo modo, tutti i processi che condividono un file usano la stessa copia del file; i cambiamenti fatti da un processo sono immediatamente visibili agli altri processi che condividono il file. L'implementazione di questa semantica è assistita dal fatto che Unix utilizza una cache del disco chiamata *buffer cache* piuttosto che buffer per singole elaborazioni del file (Paragrafo 14.13.1). Per evitare race condition durante l'accesso all'inode di un file condiviso, nella copia in memoria di un inode è presente un campo lock. Un processo che prova ad accedere a un inode deve sospendersi se il lock è stato impostato da qualche altro processo. I processi che utilizzano in maniera concorrente un file devono adottare tecniche di sincronizzazione per evitare il verificarsi di race condition sui dati contenuti nel file.

### **Allocazione dello spazio su disco**

Unix utilizza l'allocazione indicizzata dello spazio su disco, con dimensione del blocco di 4 KB. Ogni file ha una *file allocation table* analoga a una FMT, memorizzata nel proprio inode. La tabella di allocazione contiene 15 elementi ([Figura 13.33](#)). Dodici di questi puntano direttamente ai blocchi dati del file. L'elemento successivo punta a un blocco indiretto, ovvero, un blocco che contiene puntatori a blocchi dati. I due elementi successivi puntano a blocchi indiretti, rispettivamente, con due e tre livelli di indirizzamento. In questo modo, la dimensione totale del file può arrivare a un massimo di  $2^{42}$  byte. Tuttavia, l'informazione relativa alla dimensione è memorizzata in una parola a 32 bit all'interno dell'inode. Dunque, la dimensione di un file è limitata a  $2^{32}-1$  byte, per la quale sono sufficienti i blocchi diretti, con singolo e doppio livello di indirizzamento.



**Figura 13.33** File allocation table in Unix.

Per dimensioni di file inferiori a 48 KB, questa organizzazione ha la stessa efficienza della FMT flat discussa nel Paragrafo 13.7. Questi file hanno anche una piccola tabella di allocazione che può entrare nell'inode. I blocchi indiretti consentono alla dimensione del file di crescere, sebbene l'accesso a questi blocchi implica l'attraversamento dell'indirizzamento nella tabella di allocazione. Un'indagine sulla dimensione dei file in Unix condotta nel 1996 riportò che la dimensione media dei file era di 22 KB e che più del 93 per cento dei file aveva un dimensione inferiore a 32 KB. Quindi la file allocation table di Unix ha la stessa efficienza della FMT flat per molti file.

Unix mantiene una *free list* dei blocchi su disco. Ogni elemento nella lista è simile a un blocco indiretto in una FMT e, pertanto, contiene gli indirizzi dei blocchi liberi e l'id del prossimo blocco nella free list. Questa organizzazione minimizza l'overhead dovuto all'aggiunta di blocchi alla free list quando viene cancellato un file; è richiesta solo poca elaborazione per i file che contengono solo blocchi diretti e con singolo indirizzamento. Un campo lock è associato alla free list per evitare race condition quando i blocchi sono aggiunti e cancellati. Un programma del file system chiamato mkfs è utilizzato dalla free list quando viene creato un nuovo file. mkfs elenca i blocchi liberi in ordine crescente di numero di blocco durante la creazione della free list. Tuttavia, questo ordinamento viene

perso quando vengono aggiunti e rimossi dalla free list durante il funzionamento del file system. Il file system non mette in atto alcuna azione per ripristinare questo ordinamento. Quindi i blocchi allocati a un file possono essere sparsi nel disco, cosa che riduce l'efficienza dell'accesso a un file. Unix BSD utilizza i *gruppi di cilindri* per affrontare questa problematica (Paragrafo 13.7).

### **File system multipli**

La radice di un file system è chiamata *superblock*. Essa contiene la dimensione del file system, la free list e la dimensione della lista degli inode. Nell'ottica dell'efficienza, Unix mantiene il superblocco in memoria ma lo copia sul disco periodicamente. Questa organizzazione implica che alcune parti dello stato del file system vengono perse nel caso di crash del sistema. Il file system può ricostruire alcune di queste informazioni, per esempio la free list, analizzando lo stato del disco. Questa procedura è implementata come parte della procedura di boot.

Possono coesistere più file system in un sistema Unix. Ogni file system deve essere mantenuto su un singolo disco logico; per tale motivo i file di un file system non possono essere sparsi su più dischi logici. Un disco fisico può essere partizionato in molti dischi logici e un file system può essere costruito su ognuno di essi. Questo partizionamento fornisce un buon livello di protezione tra i vari file system e inoltre non consente a un file system di occupare uno spazio elevato sul disco rigido. Un file system deve essere "montato" prima di potervi avere accesso. Solo un utente con la password di root, tipicamente l'amministratore di sistema, può montare un file system.

I comandi di mount e umount del file system funzionano come segue. Dapprima, a un disco logico contenente un file system viene assegnato un nome di file speciale per dispositivo. Questo nome viene indicato come *nome\_Fs* in un comando *mount* (Paragrafo 13.5). Quando viene montato un file system, il superblocco del file system montato viene caricato in memoria. L'allocazione di un blocco del disco per un file nel file system montato viene eseguito nel dispositivo logico del file system montato. Ai file presenti in un file system montato si accede come descritto nel Paragrafo 13.9.1.

Una chiamata open in Unix specifica tre parametri: il path, le flag e le modalità. Le flag indicano che tipo di operazioni saranno effettuate sul file - se read, write o read/write. Il parametro modalità viene passato solo quando un file deve essere creato. Questa informazione tipicamente è copiata dalla maschera di creazione dei file dell'utente. Il proprietario di un file può modificare le informazioni di protezione del file in ogni momento mediante il comando *chmod*.

### **Fast file system di Berkeley**

Il fast file system (FFS) di Berkeley per Unix fu sviluppato per risolvere i limiti del file system s5fs. Supporta i *link simbolici*, ovvero un file che contiene un riferimento a un altro file. Se il link simbolico viene incontrato durante la risoluzione di un path, la risoluzione del path continua dal file referenziato. Inoltre include diverse innovazioni che riguardano l'allocazione dei blocchi del disco e l'accesso al disco, che descriveremo in seguito.

FFS consente l'uso di grandi blocchi, per cui i blocchi possono avere una dimensione massima di 8 KB. Differenti file system possono usare blocchi di diverse dimensioni; tuttavia, la dimensione del blocco non può variare all'interno di un file system. Un blocco di grandi dimensioni consente l'accesso a file di grandi dimensioni attraverso i blocchi diretti nella file allocation table. Un blocco di grandi dimensioni inoltre rende le operazioni di I/O più efficienti e rende efficiente l'uso del disco. Tuttavia, un blocco di grandi dimensioni porta a una grande frammentazione interna nell'ultimo blocco di un file. FFS tiene conto di questo effetto allocando una parte di blocco all'ultima porzione di un file. In questo modo, un blocco può essere condiviso da più file. Per facilitare questa allocazione, un blocco viene diviso in parti uguali chiamate *frammenti*. Il numero di frammenti in blocco è un parametro del file system e può valere 1, 2, 4 o 8. FFS utilizza una bitmap per tenere traccia dei frammenti liberi di un blocco. La crescita di un file richiede un'attenzione speciale in questo schema, poiché un file può aver bisogno di più frammenti, che potrebbero non essere disponibili nello stesso blocco. In questi casi, tutti i frammenti vengono spostati in un altro blocco e i frammenti precedentemente allocati sono liberati.

FFS utilizza la nozione di *gruppi di cilindri* per ridurre i movimenti delle testine del disco (Paragrafo 13.7). Per ridurre ulteriormente il movimento delle testine del disco, tutti gli inode di un file system vengono memorizzati nello stesso gruppo di cilindri e

inoltre si cerca di memorizzare l'inode di un file e il file nello stesso cilindro. Inoltre non consente a un file di occupare tutto il gruppo di cilindri. Se un file cresce fino a violare questo vincolo, viene allocato in un gruppo di cilindri più capiente. Questa tecnica aumenta la possibilità che i file cui si accede in maniera concorrente si trovino nello stesso gruppo di cilindri, cosa che ridurrebbe il movimento della testina del disco.

FFS cerca di minimizzare la latenza rotazionale durante la lettura di un file sequenziale. Come descritto successivamente nel Paragrafo 14.3.2, trascorre un certo periodo di tempo tra la fine di un'operazione di lettura e l'inizio della successiva. Durante questo tempo, i blocchi successivi inevitabilmente passano sotto la testina del disco. Anche se viene lanciato immediatamente un comando per leggere il blocco successivo, il blocco può essere letto solo durante la prossima rivoluzione del disco. Per garantire che i blocchi consecutivi vengano letti durante la stessa rivoluzione del disco, FFS li separa inserendo alcuni altri blocchi tra di loro. Questa caratteristica è simile alla tecnica di interallacciamento dei settori in una traccia discussa successivamente nel Paragrafo 14.3.2. Come sarà illustrato, questa tecnica ha un impatto significativo sul throughput del disco.

### 13.14.2 File system di Linux

Linux fornisce un file system virtuale (VFS) che supporta un modello di file comune che è simile a quello di Unix. Questo modello di file è implementato utilizzando delle strutture dati simili a quelle di Unix come i superblocchi e gli inode. Quando si apre un file, il VFS trasforma l'elemento della directory a esso relativo in un oggetto *dentry*. Questo oggetto *dentry* viene memorizzato in modo da evitare l'overhead per la sua costruzione se un file venisse aperto continuamente durante l'elaborazione. Il file system standard di Linux è chiamato ext2. Il file system ext3 incorpora il journaling, che fornisce l'integrità dei file e dei metadati e il boot veloce dopo un arresto improvviso (Paragrafo 13.12).

Ext2 fornisce una varietà di lock dei file per la sincronizzazione dei processi. I lock *consigliati* sono quelli che dovrebbero essere utilizzati dai processi per garantire la mutua esclusione; tuttavia, il file system non obbliga il loro uso. I lock dei file in Unix appartengono a questa categoria di lock. I lock *obbligatori* sono quelli controllati dal file system; se un processo tenta di accedere a dati protetti da un lock obbligatorio, il processo viene bloccato finché il lock non è resettato dal detentore. Un *lease* è uno speciale tipo di lock di file, valido per una quantità di tempo specificata, in seguito alla richiesta effettuata da un altro processo, di accedere ai dati da esso protetti. Viene implementato come segue: se un processo accede a dati protetti da un lease, il file system intima al detentore di rilasciare il lock. A questo punto il processo ha un intervallo di tempo determinato per terminare l'accesso al file e rilasciare il lease. Se non lo fa, il lease gli viene sottratto e dato al processo che aveva tentato di accedere ai dati protetti.

Il progetto di ext2 è stato influenzato dal fast file system di BSD (Paragrafo 13.14.1). Ext2 sfrutta la nozione di *gruppo di blocchi*, ovvero un insieme di blocchi consecutivi, per ridurre il movimento delle testine del disco quando un file viene aperto e si accede ai suoi dati. Utilizza una bitmap per tenere traccia dei blocchi liberi all'interno del gruppo di blocchi. Quando viene creato un file, ext2 cerca di allocare spazio su disco per l'inode del file all'interno dello stesso gruppo di blocchi che contengono la directory genitrice e inoltre memorizza i dati all'interno dello stesso gruppo di blocchi. Ogni volta che la dimensione di un file aumenta in seguito all'aggiunta di dati, il file system cerca nella bitmap del gruppo di blocchi un blocco vicino a un blocco target. Se questo blocco viene trovato, controlla se anche i blocchi vicini sono liberi e li prealloca al file. Se questo blocco non viene trovato, prealloca alcuni blocchi contigui posizionati in un'altra parte del gruppo di blocchi. In questo modo è possibile leggere grandi sezioni di dati senza dover muovere la testina del disco. Quando il file viene chiuso, i blocchi preallocati ma non utilizzati vengono liberati. Questa strategia di allocazione di spazio sul disco garantisce l'uso di blocchi contigui per sezioni contigue di dati quando i file sono creati e cancellati a un tasso elevato; questo contribuisce alle elevate prestazioni nell'accesso ai file.

### 13.14.3 File system di Solaris

Il file system di Solaris utilizza i permessi di accesso ai file nello stile Unix in cui esistono tre coppie per il controllo degli accessi in ogni lista per il controllo degli accessi: per il

proprietario del file, per gli altri utenti nel gruppo del proprietario e per tutti gli altri utenti del sistema (Paragrafo 13.6). Al fine di garantire la flessibilità mancante nello schema base, permette inoltre l'aggiunta di nuove coppie contenenti la *<lista\_degli\_id\_utente>* e la *<lista\_dei\_privilegi\_di\_accesso>* alla lista di controllo degli accessi del file; l'amministratore di sistema specifica una nuova coppia mediante il comando *setfacl*.

Solaris offre convenienza e flessibilità nell'elaborazione dei file, mediante un file system virtuale come descritto nel Paragrafo 13.13 e mediante una varietà di modalità di elaborazione dei file. Un'operazione *exclusive open* su un file fallisce se il file già esiste; altrimenti, crea il file e restituisce il suo descrittore in una singola azione indivisibile. Questa operazione evita race condition durante la creazione di un nuovo file; viene usata dai processi che creano un lock sul file per sincronizzare le proprie attività. Il lock a livello di record viene fornito per implementare la sincronizzazione a grana fine tra i processi che accedono in modo concorrente ai file; quando un processo tenta di accedere a un record il cui lock è stato impostato da un altro processo, viene bloccato finché il lock non viene rilasciato. La modalità *I/O non bloccante* viene proposta per evitare attese indefinite dovute a questa caratteristica. In questa modalità, un'operazione di I/O che tenta di accedere a un record il cui lock è impostato da un altro processo semplicemente fallisce. Il processo che ha lanciato l'operazione ha ora l'opportunità di eseguire alcune altre azioni e riprovare l'operazione di I/O successivamente. La modalità di *I/O asincrono* viene fornita per non bloccare un processo nell'attesa che venga completata l'operazione di I/O. Questa modalità è utile nelle applicazioni real-time. Nella modalità *I/O diretta*, il file system non bufferizza i dati dei file; questa modalità facilita le applicazioni, come i database che vogliono eseguire le proprie procedure di buffering.

La sincronizzazione dei dati e le flag di integrità dei file possono essere impostati nell'elemento della directory relativo a un file per ottenere operazioni affidabili. Quando alcune di queste flag sono impostati per un file, le operazioni di I/O sul file assicurano l'integrità dei metadati e/o dei file in un modo che ricorda la modalità journaling riassunta nella [Tabella 13.6](#).

### 13.14.4 File system di Windows

Il file system NTFS di Windows è progettato per soddisfare le esigenze di server e workstation. Fornisce supporto per le applicazioni client-server dei server di file e database. Una caratteristica chiave del NTFS è la possibilità di ripristinare il file system, che verrà discusso successivamente in questo paragrafo.

Una *partizione* è un grande insieme di settori contigui su di un disco; un *volume* è una partizione logica su disco; ovvero è un disco virtuale. Un volume semplice contiene una singola partizione, mentre un volume multipartizione chiamato *volume spanned* può contenere fino a 32 parzioni posizionate su uno o più dischi. NTFS esegue l'allocazione dello spazio su disco in unità chiamate *cluster*. Ogni cluster è un gruppo di settori contigui; il numero di settori in un cluster è una potenza di 2. A un cluster in un volume viene assegnato un *numero di cluster logico* (LCN), mentre a quello in un file viene assegnato un *numero di cluster virtuale* (VCN).

Un volume NTFS contiene un settore di boot, una *master file table* (MFT), alcuni file di sistema e file utente. La presenza del settore di boot rende ogni volume avviabile. Tipicamente la MFT contiene un record di 1 KB per ogni file e directory nel volume, sebbene i file di grandi dimensioni possano aver bisogno di più record MFT. La MFT contiene anche le informazioni sulle aree non utilizzate nel volume. Ogni file in un volume ha un unico *riferimento a file*, che contiene due componenti: un *numero di file* a 48 bit, che è semplicemente il numero del record MFT occupato e un *numero di sequenza* a 16 bit, ovvero il numero totale di volte che il record MFT è stato usato. Il numero di sequenza viene usato per prevenire di confondere due file che hanno usato lo stesso record MFT in momenti differenti.

Ogni file ha un insieme di attributi, in cui ogni attributo è uno stream di byte indipendente che può essere editato. Alcuni attributi standard sono comuni a tutti i file. In più, un file può avere attributi speciali richiesti da un'applicazione. Ogni file ha un record MFT chiamato *record base del file*, che contiene il riferimento al file, il tempo dell'ultimo aggiornamento e i suoi attributi. Un attributo senza nome di un file contiene i dati del file. Questa organizzazione consente ai dati di un piccolo file o di una directory di essere memorizzati nel suo record base del file, cosa che garantisce un'elevata efficienza nell'accesso al file. Se un attributo non può essere memorizzato nel record base del file,

viene memorizzato come attributo *nonresidente* - viene memorizzato in un altro record MFT e un puntatore a esso viene inserito nel record base del file. Se l'attributo nonresidente non può essere memorizzato in un record MFT, viene memorizzato nei cluster sul disco e il record MFT puntato dal record base del file contiene un mapping da VCN a LCN per i suoi cluster. Quando un processo apre un file, NTFS imposta uno *stream control block* (SCB) per ogni suo attributo. Un SCB contiene un puntatore a un *file control block* per il file, contenente il riferimento al file e un offset in un attributo. Quando il processo vuole accedere a un attributo di un file, NTFS usa lo SCB per localizzare il record base del file, trova le informazioni relative alla posizione dell'attributo e gli applica l'offset per accedere alla parte dell'attributo richiesta.

Una directory è organizzata come un albero B+ con i file come foglie ed è implementato utilizzando un file indice. La struttura dati albero B+ ha la proprietà che la lunghezza di ogni percorso nell'albero è la stessa. Questa caratteristica garantisce la ricerca efficiente per un file in una directory (Paragrafo 13.4.4). B+ NTFS mette a disposizione gli hard link per impostare più percorsi per un file. Inoltre supporta i link simbolici, chiamati *giunzioni*, che redirigono la traduzione del percorso da una directory a un'altra. Questa caratteristica fornisce un effetto analogo al mount di un file system.

NTFS adotta due tecniche per risparmiare spazio su disco. Se un file è sparso, non alloca spazio su disco alla porzione del file in cui non sono stati scritti dati o i dati scritti sono tali che uno o più settori contengono zeri. Adotta la compressione dei settori per i file non sparsi, utilizzando 16 cluster virtuali consecutivi in un file come unità. Li sostituisce con una forma compressa solo se l'azione consente di liberare almeno un cluster e annota questa situazione in modo che possa eseguire la decompressione automaticamente quando si accede al file.

NTFS memorizza anche i metadati nei file. Alcuni di questi file sono come segue.

- Il *file MFT* contiene i record MFT.
- Il *file log* contiene informazioni usate per il ripristino; il suo utilizzo è descritto successivamente in questo paragrafo.
- La *attribute definition table* contiene informazioni relative agli attributi.
- Un *file bitmap* indica quali cluster in un volume sono allocati e quali sono liberi.
- Il *file boot* contiene il settore di boot.
- Un file per i *cluster danneggiati* tiene traccia dei cluster non utilizzabili a causa di problemi hardware.

NTFS fornisce robustezza garantendo la consistenza dei metadati quando si verifica un crash di sistema. Questo risultato è ottenuto trattando ogni modifica dei metadati come una transazione atomica. Dalla discussione delle azioni atomiche nel Paragrafo 13.11.2, sembrerebbe che le transazioni atomiche possano essere implementate semplicemente scrivendo le "intenzioni" di una transazione in un file log scritto in precedenza ed eseguendo di fatto le intenzioni quando la transazione verrà compiuta (commit). Tuttavia, alcune azioni come la creazione di un nuovo record di un file nella MFT non può essere ritardata finché una transazione non è commit, per cui NTFS utilizza un log combinato ripeti/annulla che contiene due tipi di record. La collezione di record *ripeti* (redo) nel log ricorda la lista delle intenzioni del Paragrafo 13.11.2, mentre i record *annulla* (undo) riguardano le azioni che già sono state eseguite da transazioni che sono già commit. Durante il normale funzionamento, solo i record redo sono utilizzati - sono elaborati per eseguire le modifiche ai metadati NTFS quando una transazione è commit. I record undo sono usati solo durante il ripristino da un crash, come descritto di seguito.

NTFS esegue il ripristino come segue: modifica i metadati secondo gli elementi contenuti nel log relativo alle transazioni che sono state completate prima del crash. Successivamente elimina gli elementi che hanno compiuto transazioni che non sono state portate a compimento prima del crash. I metadati si trovano in uno stato consistente alla fine di queste azioni, per cui NTFS riprende il normale funzionamento. Questa caratteristica fornisce le funzionalità di *write behind* del file system journaling discusso nel Paragrafo 13.12.

In linea di principio, gli elementi del log relativi a una transazione possono essere scartate dopo che tutte le azioni sono state eseguite durante il normale funzionamento o dopo un ripristino o dopo che tutte le azioni sono state annullate durante un ripristino. Tuttavia, NTFS non può scartare gli elementi di log in questo modo per due ragioni: conserva i metadati in file e utilizza una *file cache* (Paragrafo 14.13.3) per velocizzare le attività di elaborazione dei file. In questo modo, i cambiamenti in un file che contiene i

metadati durante l'elaborazione di un elemento ripeti o annulla del log rimarrebbero nella file cache per un lungo periodo e potrebbero essere persi nel caso in cui si verificasse un crash prima della scrittura su disco. Per prevenire la crescita incontrollata del file log, NTFS fissa un *checkpoint* ogni 5 secondi. Inserisce un record checkpoint nel log nell'istante, in cui scrive il contenuto dei blocchi modificati presenti nella file cache. Quando si verifica un crash, NTFS localizza l'ultimo record checkpoint nel log, ripristina i valori dei blocchi sul disco nella file cache e successivamente elabora gli elementi ripeti/annulla delle transazioni che erano in esecuzione al momento del crash. Questa procedura di ripristino non richiede che vengano conservati gli elementi del log relativi alle transazioni committed o annullate, dunque NTFS cancella questi elementi del log conservando i record checkpoint.

I dati dei file possono essere persi se un crash danneggiasse dei blocchi del disco. Il driver del gestore del volume che gira sotto NTFS adotta la tecnologia RAID per gestire questi problemi. Il *mirroring del disco* implica la memorizzazione di dati identici nei blocchi del disco in due dischi, in modo che uno dei blocchi del disco sarebbe accessibile anche se l'altro fosse danneggiato in seguito a un guasto. (Il mirroring del disco e altre configurazioni RAID sono discusse nel Paragrafo 14.3.5.)

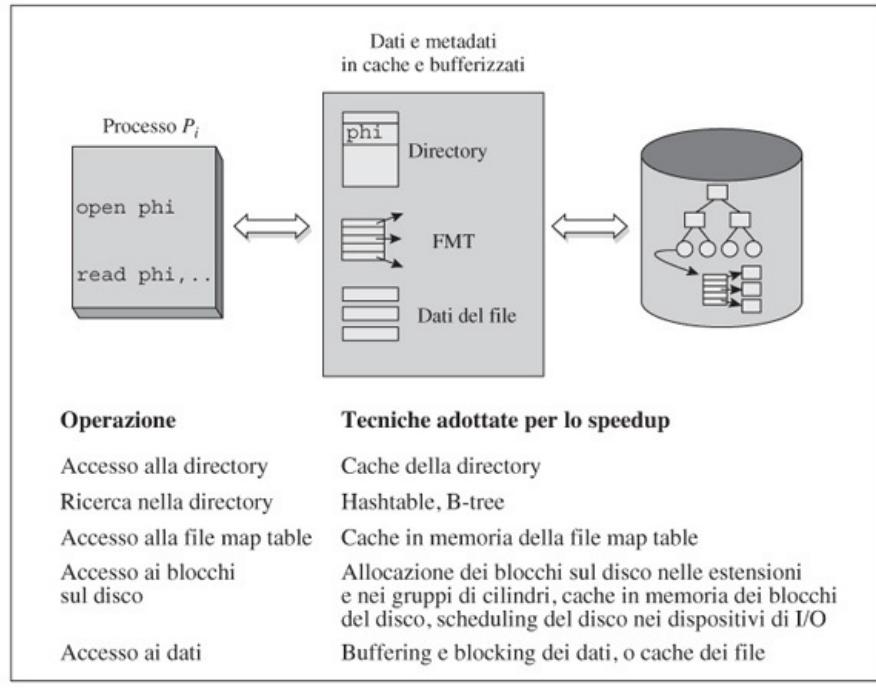
Windows Vista presenta molte nuove funzionalità per il ripristino. Il *kernel transaction manager* implementa la semantica delle transazioni sui file e sugli oggetti che possono trovarsi su diversi computer. Il *centro di backup e ripristino* consente a un utente di specificare quando e quanto frequentemente ogni file deve essere salvato in un backup e di richiedere il ripristino di una versione precedente del file. Per risparmiare spazio sul disco, memorizza solo i cambiamenti eseguiti in un file presente in un backup.

### 13.15 Prestazioni dei file system

I file system adottano cinque tecniche per fornire alte prestazioni nell'accesso ai file.

- *Uso di strutture dati efficienti*: le direcotry sono organizzate utilizzando strutture dati che consentono ricerche veloci.
- *Allocazione effettiva dello spazio su disco*: lo spazio su disco è allocato a un file in modo da ridurre i movimenti della testina del disco e i ritardi rotazionali durante l'elaborazione di un file sequenziale.
- *Caching*: parte della memoria è usata come *cache* per i dati memorizzati su un dispositivo di I/O. Come discusso nel Paragrafo 2.2.3, il caching velocizza gli accessi alle informazioni che esibiscono la *località temporale* o la *località spaziale*, ovvero i dati cui si accede di frequente o posizionati in prossimità di dati cui si è avuto accesso precedentemente.
- *Buffering*: un *buffer* è un'area di memoria usata per memorizzare i dati in maniera temporanea. Il file system carica i dati da un dispositivo di I/O in un buffer prima che un processo li richieda, in modo che il processo possa accedere ai dati senza dover aspettare il completamento di un'operazione di I/O. L'azione contraria è eseguita quando un processo vuole scrivere i dati in un file.
- *Scheduling del disco*: le operazioni di I/O su un disco sono eseguite in un ordine che riduce i movimenti della testina del disco; assicura un throughput elevato del disco.

La [Figura 13.34](#) riassume come un file system utilizza queste tecniche per velocizzare l'elaborazione di un file. Le hash table e gli alberi B+ consentono ricerche veloci in una directory (Paragrafo 13.4.3). L'allocazione dello spazio su disco di un file è confinata alle *estensioni* e ai *gruppi di cilindri* per ridurre il movimento della testina del disco e la latenza rotazionale (Paragrafo 13.7). Le altre tecniche forniscono accessi veloci ai dati nel file e ai metadati di un file system, come gli elementi di una directory e le FMT.



**Figura 13.34** Tecniche adottate per fornire alte prestazioni nell'accesso ai file.

Le directory sono poste in memoria cache quando vi si accede per la prima volta. In questo modo una directory utilizzata per risolvere un percorso viene mantenuta nella cache per velocizzare i riferimenti futuri ai file in essa contenuti. Questa cache è chiamata *cache dei nomi di una directory*. Una FMT viene bufferizzata in memoria quando il file è aperto, in anticipo rispetto agli accessi. Può essere mantenuta in cache dopo il primo accesso. Il buffering non può essere utilizzato nel caso in cui la FMT sia troppo grande. In questo caso, alcune sue parti possono essere mantenute in memoria dopo il primo riferimento.

Una *disk cache* memorizza i blocchi del disco in memoria in seguito al primo utilizzo nell'attività di elaborazione di un file. Hit ratio migliori di 0.9 sono possibili nella disk cache. Dunque il suo utilizzo riduce il numero di operazioni di I/O su un disco. Un metodo di accesso utilizza il *buffering* e il *blocking* dei dati del file o memorizza i dati dei file in una *file cache* per ridurre le attese durante le operazioni di I/O. Lo *scheduling del disco* viene usato per ridurre i movimenti delle testine del disco e il tempo medio di attesa per le operazioni di I/O. Queste tecniche sono adottate dal IOCS; sono discusse successivamente nel [Capitolo 14](#).

Con l'avanzare della tecnologia, le tecniche sviluppate per essere usate nel software vengono implementate in hardware. La tecnologia dei moderni dispositivi di I/O incorpora alcune tecniche rappresentate in [Figura 13.34](#). I dischi SCSI forniscono lo scheduling del disco all'interno dello stesso dispositivo. Le unità RAID contengono un buffer per i blocchi del disco, che può essere usato sia come buffer che come cache dei blocchi del disco. Queste tecnologie saranno discusse nel [Capitolo 14](#).

### 13.15.1 File system di tipo log-structured

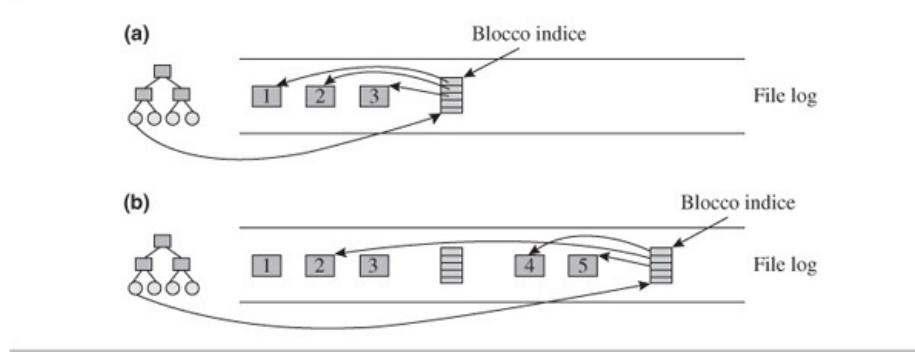
La cache del disco riduce il numero delle operazioni di lettura dirette al disco. Dunque l'uso del disco è dominato dal movimento della testina del disco e dalle operazioni di scrittura. Il movimento della testina del disco può essere ridotto utilizzando lo scheduling del disco e utilizzando i *gruppi di cilindri* nell'allocazione dello spazio del disco per i file. Tuttavia, queste tecniche sono meno efficaci quando i file posizionati in parti differenti del disco sono elaborati simultaneamente, situazione che si verifica spesso in un sistema condiviso. Per esempio, in un sistema Unix, le operazioni di scrittura utilizzano solo il 10 per cento del tempo del disco; il resto del tempo viene speso per il movimento della testina del disco, cosa che porta a uno scarso throughput del disco.

Un *file system* del tipo *log-structured* riduce il movimento della testina del disco

impiegando un'organizzazione dei file radicalmente diversa. Scrive i dati di *tutti* i file insieme in una singola struttura sequenziale che ricorda un journal. Chiamiamo questa struttura il *file log*. Quando un'operazione di modifica o scrittura viene eseguita su un qualsiasi file, i nuovi dati sono semplicemente aggiunti alla fine del file log. Dunque questa operazione richiede solo un limitato movimento della testina del disco. Il file system scrive speciali *blocchi indice* nel file log per contenere i metadati relativi alla posizione dei dati di ogni file nel file log. Questi blocchi indice sono usati quando i dati di un file devono essere letti dal disco. In questo modo, sono richiesti pochi movimenti della testina del disco per la lettura dei dati per dati scritti di recente in un file; tuttavia, per i dati più vecchi sono necessari più movimenti. Gli studi sulle prestazioni del file system Sprite di tipo log-structured hanno mostrato che i movimenti della testina del disco ammontano solo al 30 per cento del tempo del disco impiegato durante l'elaborazione del file e che le sue prestazioni sono superiori al file system convenzionale per scritture brevi e frequenti. L'Esempio 13.12 illustra il funzionamento di un file system di tipo log-structured.

#### Esempio 13.12 - File system di tipo log-structured

La [Figura 13.35\(a\)](#) mostra uno schema dell'organizzazione utilizzata in un file system di tipo log-structured. Per semplicità, mostra i metadati e i dati di un singolo file nel file log. I blocchi dati nel file log sono numerati per convenienza. L'elemento della directory di un file punta a un blocco indice nel file log; assumiamo che il blocco indice contenga la FMT del file. Quando il file che si trova nel blocco 1 viene aggiornato, i nuovi valori vengono scritti in nuovo blocco del disco, ovvero, il blocco 4. In modo simile alcuni dati vengono scritti nel blocco 5 quando i dati nel blocco 3 vengono aggiornati. Il file system scrive un nuovo blocco indice che contiene la FMT aggiornata del file e imposta il puntatore alla FMT nell'elemento della directory relativo al file per puntare al nuovo blocco indice. La nuova FMT contiene i puntatori ai due nuovi blocchi dati e al blocco dati 2 che non è stato modificato [[Figura 13.35\(b\)](#)]. Il vecchio blocco indice e i blocchi 1 e 3 sono ora liberi.



**Figura 13.35** Aggiornamento del file in un file system log-structured.

Dal momento che il file log è scritto come un file ad accesso sequenziale, il file system deve garantire che un'area del disco sufficientemente grande sia sempre disponibile per scrivere il file log. Questo obiettivo è raggiunto spostando i blocchi dati sul disco per creare un'area di memoria libera disponibile per il file log. Questa operazione è analoga alla compattazione della memoria (Paragrafo 11.5.1). Coinvolge considerevoli movimenti della testina del disco, che domina l'uso del disco; tuttavia, la compattazione viene eseguita come attività di background per cui non ritarda le attività di elaborazione dei file nei processi.

## Riepilogo

Gli utenti di un computer hanno molte aspettative relativamente a un file system: convenienza, buone prestazioni dell'attività di elaborazione di un file e un uso efficiente dei dispositivi di I/O. Per soddisfare queste richieste in maniera efficace, il file system è strutturato in due strati: lo strato del file system che affronta le problematiche relative alla convenienza come la condivisione e la protezione dei file e l'affidabilità; lo strato,

chiamato *input-output control system* (IOCS), che implementa le operazioni sui file e affronta le problematiche di efficienza. In questo capitolo, abbiamo discusso le tecniche dei file system.

Un file può essere un file strutturato, ovvero può contenere record di dati, o può essere un file non strutturato anche detto *stream di byte*. Un file system fornisce convenienza per gli utenti in tre modi. Primo, fornisce differenti *organizzazioni dei file*, dove ogni organizzazione si adatta a uno specifico pattern per l'accesso al record di un file, e fornisce un metodo per organizzare i record di un file su un dispositivo di I/O e per accedervi in maniera efficiente. L'organizzazione *sequenziale* e *diretta* dei file si adatta all'accesso ai record di un file, rispettivamente, sequenziale e casuale. Sono, d'altra parte, molto utilizzate diverse organizzazioni ibride, come la *sequenziale indicizzata*. Secondo, un file system consente a un utente di raggruppare file correlati in maniera conveniente creando file e directory fino al livello desiderato. Terzo, consente a un utente di specificare quali altri utenti possono accedere ai propri file e in che modo, cosa che facilita la *condivisione* e la *protezione* dei file.

Il file system alloca lo spazio su disco a un file in modo da evitare la frammentazione dello spazio su disco e in modo da accedere in maniera efficiente ai dati dei file. L'*allocazione indicizzata* dello spazio sul disco a un file utilizza un blocco del disco o un'estensione come unità di allocazione dello spazio su disco. I blocchi del disco o le estensioni allocate a un file sono ristrette a un *gruppo di cilindri* per garantire l'accesso efficiente ai dati. Le informazioni riguardanti lo spazio sul disco allocato a un file sono memorizzate in una *file map table* (FMT).

Prima di leggere o scrivere in un file, un processo deve aprire il file specificando il *percorso* nella struttura delle directory. Il file system attraversa il percorso, determina quale file deve essere aperto e imposta un *file control block* (FCB) per contenere le informazioni come il tipo di file e l'organizzazione, l'indirizzo della sua FMT e l'indirizzo del prossimo record. Quando un processo vuole eseguire un'operazione di lettura o scrittura, il file system passa il FCB al IOCS e il IOCS implementa l'operazione, utilizzando le informazioni accessibili mediante il FCB. Il file system specifica la *semantica della condivisione dei file*, che determina come i risultati della modifica di un file fatti da un processo devono essere resi visibili agli altri processi che utilizzano il file in maniera concorrente.

Il file system garantisce l'affidabilità del funzionamento assicurando che i dati e i metadati come le FMT e i FCB non siano persi o resi inconsistenti dai guasti come le interruzioni di corrente. Questo obiettivo è ottenuto utilizzando un'*azione atomica*, la quale garantisce che tutte le azioni in un insieme di azioni correlate siano completate anche nel caso di un guasto. Un'azione atomica genera un overhead considerevole, per cui un *file system journaling* fornisce una serie di modalità che garantiscono dati e metadati in misura differente, in modo che un amministratore di sistema possa scegliere la modalità che più si adatta a un ambiente di elaborazione.

Un *file system virtuale* (VFS) è un livello software che consente a diversi file system di essere in funzione su un computer simultaneamente, in modo che un utente possa scegliere il file system che è più adatto per le sue applicazioni. Il VFS fornisce un metodo unificato per l'accesso ai differenti file system. Un processo invoca il livello VFS utilizzando comandi generali per l'accesso ai file e il livello VFS redireziona il comando al file system appropriato.

## Domande

- 13.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
  - a. L'allocazione di spazio contiguo sul disco per un file ad accesso sequenziale porta a un'elaborazione dei file più efficiente rispetto all'allocazione non contigua.
  - b. I cicli nella struttura delle directory creano difficoltà con l'operazione di cancellazione.
  - c. I percorsi assoluti di due file non possono essere identici, mentre i loro path relativi possono essere identici.
  - d. Lo scopo del file control block (FCB) è di facilitare l'operazione di apertura di un file; il FCB può essere cancellato immediatamente dopo l'apertura del file.
  - e. Quando un file viene chiuso dopo un aggiornamento, la directory che contiene il file può dover essere aggiornata.

- f. Memorizzare la file map table (FMT) di un file in memoria mentre il file è elaborato riduce il numero di accessi al disco durante l'elaborazione del file.
  - g. Durante la creazione di un nuovo file in un file system montato, al file è allocato spazio sul disco logico usato dal file system montato.
  - h. L'effetto del mounting di un file system è simile a quello di impostare un link nella struttura delle directory, fatta eccezione per il fatto che l'effetto viene perso quando si smonta il file system.
  - i. Quando un utente aggiorna i dati in un file modificabile a singola immagine, i cambiamenti effettuati nel file non sono immediatamente visibili agli utenti che accedono al file in maniera concorrente.
  - j. Quando si verifica un guasto, un solo backup incrementale è sufficiente a ripristinare l'intero sistema a un precedente stato consistente.
  - k. Il journaling genera overhead durante il normale funzionamento di un sistema.
  - l. Un file system virtuale consente l'uso di molti file system in un computer; tuttavia, questi file system non possono essere usati in maniera concorrente.
- 13.2. Scegliere la corretta alternativa in ognuna delle seguenti domande.
- a. Il file control block (FCB) di un file *alpha*:
    - i. contiene solo informazioni copiate dall'elemento della directory relativo al file *alpha*;
    - ii. è utilizzato per evitare accessi frequenti all'elemento della directory relativo al file *alpha*;
    - iii. è usato solo per proteggere il file *alpha* dagli accessi non consentiti.
  - b. La tecnica della memorizzazione stabile è:
    - i. una tecnica di fault tolerance usata per ripristinare due blocchi danneggiati in un disco;
    - ii. una tecnica di ripristino usata per ripristinare il file system dopo una mancanza di corrente;
    - iii. una tecnica di fault tolerance usata per ripristinare un blocco danneggiato in un disco;
    - iv. nessuna delle precedenti.

## Problemi

- 13.1. Gli utenti di un sistema accedono frequentemente a un file *alpha*. Le seguenti alternative sono proposte per semplificare l'accesso a *data*.
- a. Creare un link ad *alpha* alla home directory di ogni utente.
  - b. Copiare *data* nella directory home di ogni utente.
- Confrontare i vantaggi e gli svantaggi di questi approcci.
- 13.2. Un file sequenziale indicizzato contiene 10 000 record. Il suo indice contiene 100 elementi. Ogni elemento dell'indice descrive un'area del file che contiene 100 record. Se la probabilità di accesso è uguale per tutti i record nel file, calcolare il numero medio di operazioni del disco necessarie per accedere a un record.
- Confrontare questo numero con il numero di operazioni del disco richieste se gli stessi record fossero memorizzati in un file sequenziale.
- 13.3. Si consideri il file sequenziale indicizzato di [Figura 13.5](#). Il seguente problema si verifica quando viene aggiunto un nuovo record, per esempio il record per il dipendente numero 8 (chiamiamolo record 8). Non c'è spazio per memorizzare il nuovo record sulla traccia. Dunque il metodo di accesso estrae il record 13 dalla traccia e sposta i record 10 e 12 per fare spazio al nuovo record. Il record 13 viene inserito in una *overflow area*. Un nuovo campo chiamato *puntatore alla overflow area* viene aggiunto a ogni elemento nella traccia indice. Questo campo nel primo elemento della traccia indice punta al record 13 nella overflow area. Se più record escono dalla traccia indice, sono inseriti nella lista concatenata e il puntatore alla overflow area della traccia indice punta alla testa della lista. Liste concatenate simili possono essere create per diverse tracce.
- Se la probabilità di accesso è uguale per tutti i record nel file sequenziale indicizzato, mostrare che l'efficienza nell'accesso al file sarà influenzata dalla presenza di record nella overflow area. L'efficienza dell'accesso può essere ripristinata riscrivendo il file come nuovo file che non contiene alcuna overflow area?

- 13.4. Il sistema operativo distribuito Amoeba utilizza l'allocazione contigua dello spazio sul disco. Quando un file viene modificato, il sistema scrive il file modificato come nuovo file e cancella la vecchia copia. Commentare i vantaggi e gli svantaggi di questo approccio.
- 13.5. L'allocazione non contigua dello spazio sul disco influenza l'efficienza delle organizzazioni fondamentali dei file discusse nel Paragrafo 13.3?
- 13.6. Un file system utilizza l'allocazione indicizzata dello spazio sul disco. La dimensione di ogni blocco è di 4 KB e l'indirizzo di ogni blocco è di 4 byte. La dimensione della FMT è di un blocco. Contiene 12 puntatori a blocchi dati. Tutti gli altri puntatori puntano a blocchi indice. Un file sequenziale info contiene 5000 record, ognuno di dimensione pari a 4 KB. Le caratteristiche del disco e di un processo che utilizza il file info sono le seguenti.  
 Tempo medio di lettura di un blocco = 3 ms  
 Tempo medio di elaborazione di un blocco = 5 ms  
 Calcolare il tempo trascorso nelle seguenti condizioni:  
 a. il file system mantiene la FMT in memoria, ma non mantiene alcun blocco indice in memoria durante l'elaborazione di info;  
 b. il file system mantiene in memoria la FMT e un blocco indice di info.
- 13.7. Un nuovo record deve essere aggiunto al file `info` del Problema 13.6. Qual è il numero minimo di operazioni del disco necessarie per portare questo cambiamento in `info` sul disco? Qual è il numero massimo?
- 13.8. Un file system utilizza l'allocazione indicizzata dello spazio su disco; tuttavia, consente a un file sequenziale di contenere blocchi parzialmente pieni. Quali sono i vantaggi e gli svantaggi di questo schema?
- 13.9. Un file system usa l'allocazione contigua dello spazio su disco. Il metodo di accesso sequenziale gestisce i blocchi danneggiati sul disco come segue: se si verifica un errore durante la lettura/scrittura di un blocco, consulta la tabella dei blocchi danneggiati memorizzata sul disco e accede al blocco alternativo assegnato al blocco danneggiato. Supponendo che tutti gli accessi al disco richiedano tempi di accesso identici, calcolare il degrado delle prestazioni di accesso ai file nel caso in cui il 2 per cento dei blocchi allocati a un file siano blocchi danneggiati. Suggerire un metodo per migliorare le prestazioni dell'accesso.
- 13.10. Per ridurre l'overhead della validazione dell'accesso ai file (Passo 2 del Paragrafo 13.9.2), il progettista di un SO propone di eseguire la validazione solo al momento dell'apertura del file. Come menzionato nel Paragrafo 13.9.1, l'istruzione `open` specifica il tipo di accessi che saranno eseguiti al file, per esempio, `open (abc, 'read', ...)`. Una singola validazione dell'accesso al momento dell'apertura del file è adeguata? In caso negativo, spiegare perché. In ogni caso, suggerire un'implementazione.
- 13.11. Il Passo 2 del Paragrafo 13.9.1 crea un FCB per ogni directory incontrata in un path.  
 a. Questa organizzazione è adeguata nel caso in cui venga usato un path relativo?  
 b. Questi elementi sono necessari se un file deve essere aperto in lettura?  
 c. Il numero di FCB creati per ogni file può essere ridotto?
- 13.12. Spiegare come le seguenti caratteristiche possono essere incorporate in un file system.  
 a. *Mount a cascata*: la directory `C` contiene un file `D`. La struttura delle directory radicata in `C` viene montata nel mount point `X/B`. Successivamente, la struttura delle directory radicata in `X` viene montata nella directory `Y/A`. Dovrebbe essere possibile accedere al file `D` come `..Y/A/B/D`.  
 b. *Mount multipli*: la struttura delle directory radicata in qualche directory, per esempio, `W`, viene montata in molti mount point simultaneamente.
- 13.13. Quando viene utilizzata l'allocazione indicizzata per i file, spiegare come un blocco del disco si può trovare in più di un file se si verifica un guasto.
- 13.14. Sia l'Algoritmo 13.1 riscritto come segue:  
 1.  $d_j.next := d_1.next;$   
 2.  $d_1.next := \text{address } (d_j);$   
 3.  $\text{write } d_j \text{ to disk};$   
 4.  $\text{write } d_1 \text{ to disk}.$

Questo algoritmo modificato previene la confusione tra i file in caso di guasto?

- 13.15. Spiegare come il byte di offset in un file Unix può essere convertito nella coppia (*<id del blocco del disco>*, *<byte di offset>*).
- 13.16. Di default, Unix assegna i file *stdin* e *stdout*, rispettivamente, alla tastiera e al terminale. Un utente può usare gli operatori di redirezione *< e >* in un comando per modificare le assegnazioni di default e usare altri file come input e output. L'operatore di "redirezione e aggiunta" *>>* aggiunge l'output di un processo alla fine di un file esistente. Le assegnazioni di default dei file sono ripristinate al termine del comando. Queste caratteristiche possono essere implementate associando in maniera permanente i FCB dello *stdin* e dello *stdout* con ogni processo.
- Descrivere le azioni del file system necessarie per implementare le assegnazioni di default per lo *stdin* e lo *stdout* e gli operatori di redirezione *< e >*.
  - Descrivere le azioni del file system necessarie per implementare l'operatore *>>*.
- 13.17. I blocchi del disco allocati a un file sono aggiunti alla free list quando il file viene cancellato. Scrivere un algoritmo per eseguire questa operazione in Unix.
- 13.18. Il file system di Unix associa un campo lock alla free list (Paragrafo 13.14.1). Classificare la seguente affermazione come vera o falsa: "Il lock della free list è necessario a causa della natura dei processi Unix. Questo lock non è necessario in un SO che utilizza il modello di processo convenzionale".

## Problemi avanzati

Si consideri un file formato da 70 record e le sue possibili allocazioni su disco di tipo contigua, concatenata e con tabella indice. In ognuno di questi casi i record del file sono memorizzati uno per blocco.

Le informazioni che riguardano il file sono già in memoria centrale e la tabella indice è contenuta in un unico blocco. Si dica quanti accessi al disco (lettura e/o scrittura) sono necessari, in ognuna di queste situazioni, per cancellare:

1. il primo record;
2. il 40-esimo record;
3. l'ultimo record.

## Note bibliografiche

Organick (1972) storicamente è il più importante documento riguardante le strutture delle directory, poiché la struttura delle directory di MULTICS ha influenzato molti file system contemporanei come Unix, Linux, Solaris e Windows. USENIX (1992) contiene i proceeding di un workshop per file system. Grosshans (1986), Weiderhold (1987) e Livadas (1990) discutono le organizzazioni dei file e dei file system.

McKusick et al. (1990) descrivono un file system basato sulla memoria, che fornisce file mappati in memoria e strutture di directory implementate nella memoria paginabile. Levy e Silberschatz (1990) discutono le semantiche di condivisione dei file. Lampson (1981) descrive la tecnica della memorizzazione stabile per l'affidabilità dei dati su disco, mentre Svobodova (1984) descrive come le azioni atomiche sono effettuate in vari file server. Florido (2000) discute la progettazione del journaling file system. Kleiman (1986) descrive la progettazione del file system virtuale. Vahalia (1996) descrive l'interfaccia del file system virtuale di Unix. Rosenblum e Ousterhout (1992) discutono la progettazione del file system Sprite, mentre Matthews et al. (1997) discutono i metodi adattivi per migliorare le prestazioni dei file system di tipo log-structured. McKusick et al. (1996) discutono il file system di tipo logstructured di Unix 4.4 BSD.

Bach (1986) e Vahalia (1996) descrivono il file system di Unix. Kowalski (1978) descrive il programma Unix utilizzato per verificare l'integrità del file system. Questo programma controlla ogni struttura dati del file system presente sul disco. Bina e Emrath (1989) spiegano come i controlli dell'integrità del file system possano essere velocizzati nel file system di Unix. Beck et al. (2002) e Bovet e Cesati (2005) descrivono il file system ext2 di Linux. Mauro e McDougall (2006) descrivono il file system di Solaris. Nagar (1997) e Russinovich e Solomon (2005) descrivono il file system NTFS di Windows.

1. Bach, M.J. (1986): *The Design of the Unix Operating System*, Prentice Hall, Englewood Cliffs, N.J.
2. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verwörner (2002): *Linux Kernel Programming*, Pearson Education, New York.
3. Bina, E.J., and P.A. Emrath (1989): "A faster *fsck* for BSD UNIX," *Proceedings of the Winter 1989 USENIX Technical Conference*, 173-185.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, Calif.
5. Burrows, M., C. Jerian, B. Lampson, and T. Mann (1992): "On-line data compression in a log-structured file system," *ACM Sigplan Notices*, **27**, 9, 2-9.
6. Florido, J.I.S. (2000): "Journal file systems," *Linux Gazette*, issue 55.
7. Grosshans, D. (1986): *File Systems: Design and Implementation*, Prentice Hall, Englewood Cliffs, N.J.
8. Kleiman, S.R. (1986): "Vnodes: an architecture for multiple file system types in Sun Unix," *Proceedings of the Summer 1986 USENIX Technical Conference*, 238-247.
9. Kowalski, T. (1978): "Fsck - the Unix system check program," Bell Laboratories, Murray Hill, N.J.
10. Lampson, B.W. (1981): "Atomic transactions," in *Distributed systems - Architecture and Implementation: An Advanced Course*, Goos, G., and J. Hartmanis (eds), Springer Verlag, Berlin, 246-265.
11. Levy, H.M., and A. Silberschatz (1990): "Distributed file systems: concepts and examples," *ACM Computing Surveys*, **22**, 4, 321-374.
12. Livadas, P. (1990): *File Structures: Theory and Practice*, Prentice Hall, Englewood Cliffs, N.J.
13. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
14. Matthews, J.N., D. Roselli, A.M. Costello, R.Y. Wang, and T.E. Anderson (1997): "Improving the performance of log-structured file systems with adaptive methods," *Proceedings of Sixteenth Symposium on Operating Systems Principles*, 238-251.
15. Mauro, J., and R. McDougall (2006): *Solaris Internals*, 2nd ed., Prentice-Hall, Englewood Cliffs, N.J.
16. McKusick, M.K., K. Bostic, M. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading, Mass.
17. McKusick, M.K., M. Karels, and K. Bostic (1990): "A pageable memory based filesystem," *Proceedings of the Summer 1990 USENIX Technical Conference*, 137-144.
18. Nagar, R. (1997): *Windows NT File System Internals*, O'Reilly, Sebastopol, Calif.
19. Organick, E.I. (1972): *The MULTICS System*, MIT Press, Cambridge, Mass.
20. Rosenblum, M., and J.K. Ousterhout (1992): "The design and implementation of a logstructured file system," *ACM Transactions on Computer Systems*, **10**, 2, 26-52.
21. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
22. Svobodova, L. (1984): "File servers for network-based distributed systems," *ACM Computing Surveys*, **16**, 4, 353-398.
23. USENIX (1992): *Proceedings of the File Systems Workshop*, Ann Arbor, Mich., May 1992.
24. Vahalia, U. (1996): *Unix Internals: The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.
25. Weiderhold, G. (1987): *File Organization for Database Design*, McGraw-Hill, New York.

---

# CAPITOLO 14

## Implementazione delle operazioni su file

---

### Obiettivi di apprendimento

- Organizzazione dell'I/O
- Classificazione dei dispositivi di I/O
- Tecnica dell'I/O programmato
- IOCS fisico
- Progettazione e implementazione dei driver di dispositivo
- Algoritmi di schedulazione del disco
- Buffering e blocking dei record
- File cache e disco cache. Cache unificata
- Gestione dei file nei sistemi operativi: Unix, Linux e Windows

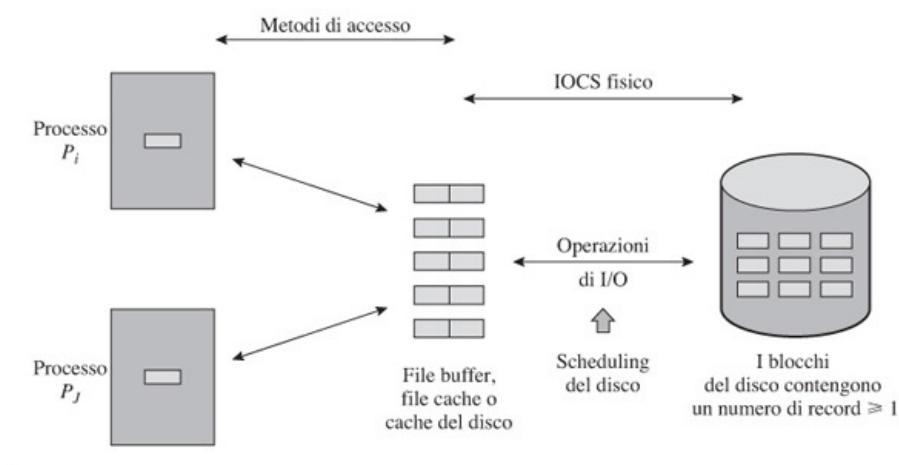
Come visto nel [Capitolo 13](#), l'elaborazione di un file è implementata utilizzando i moduli del file system e il sistema di controllo input/output (input/output control system - IOCS). I moduli del file system consentono di avere libertà nella scelta dei nomi, la condivisione e la protezione dei file e l'affidabilità. Le operazioni sui file vengono implementate dal IOCS.

Il IOCS ha due obiettivi principali - l'implementazione efficiente dell'attività di elaborazione di un file in un processo e il throughput elevato dei dispositivi di I/O. Per realizzare questi obiettivi, il IOCS è organizzato in due livelli chiamati *metodo di accesso* e *IOCS fisico*. Il modulo di un metodo di accesso gestisce la lettura e la scrittura dei dati per implementare in maniera efficiente l'elaborazione di un file in un processo. Questo modulo invoca il IOCS fisico per eseguire la lettura e la scrittura dei dati. Il IOCS fisico esegue l'input/output a livello di dispositivo e utilizza le politiche di scheduling per migliorare il throughput dei dispositivi di I/O.

Inizialmente affronteremo le caratteristiche di un dispositivo di I/O e le configurazioni che forniscono alta affidabilità, accessi veloci ed elevati tassi di trasferimento dei dati. Successivamente discuteremo di come le operazioni di I/O sono eseguite al livello dei dispositivi di I/O, delle funzionalità offerte dal IOCS fisico per semplificare le operazioni di I/O e di come lo *scheduling del disco* consente di ottenere valori elevati di throughput del disco. Infine, spiegheremo di come le tecniche di *buffering*, *blocking* e *caching* dei dati velocizzano l'attività di elaborazione dei file.

### 14.1 Livelli dell'input-output control system

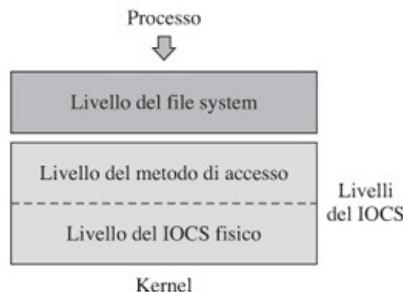
La semantica della [Figura 14.1](#) mostra come l'input-output control system (IOCS) implementa le operazioni sui file. I processi  $P_i$  e  $P_j$  sono impegnati nell'elaborazione di file ed hanno già aperto altri file. Quando uno di questi processi effettua una richiesta di lettura o scrittura di dati da un file, il file system passa la richiesta al IOCS. Si ricordi dal Paragrafo 13.1 che il IOCS mantiene alcuni dati in aree di memoria chiamate *buffer*, *file cache* o *cache del disco* per velocizzare l'elaborazione dei file. Per un'operazione di lettura, il IOCS controlla se i dati richiesti dal processo sono presenti in memoria. In caso affermativo, il processo può accedere ai dati direttamente; altrimenti, il IOCS esegue una o più operazioni di I/O per caricare i dati in un file buffer o nella cache del disco e il processo deve attendere finché questa operazione di I/O non viene completata. Dal momento che molti processi eseguono operazioni di I/O in modo concorrente, le operazioni di I/O vengono schedulate da un algoritmo di *schedulazione del disco*, il cui scopo è fornire un elevato throughput del disco. In questo modo il IOCS implementa le operazioni di I/O in modo da fornire efficienza nell'attività di elaborazione dei file nei processi e un elevato throughput dei dispositivi di I/O.



**Figura 14.1** Implementazione delle operazioni sui file da parte del IOCS.

Il IOCS è strutturato in due livelli chiamati *metodo di accesso* e *IOCS fisico*. Il metodo di accesso fornisce l'elaborazione efficiente dei file e il IOCS fisico fornisce il throughput elevato dei dispositivi. Questa struttura del IOCS consente di separare le problematiche relative all'implementazione delle operazioni sui file a livello di processo da quelle a livello di dispositivo.

La [Figura 14.2](#) mostra la gerarchia dei livelli file system e IOCS. Il numero di livelli dell'IOCS e le loro interfacce variano in base al sistema operativo. Nei vecchi sistemi operativi, il IOCS fisico tipicamente era una parte del kernel; tuttavia, i moderni sistemi operativi lo posizionano fuori dal kernel per migliorare l'estendibilità e l'affidabilità del SO. Assumeremo che il IOCS fisico venga richiamato mediante chiamate di sistema e che richiami altre funzionalità del kernel attraverso l'uso di chiamate di sistema.



**Figura 14.2** Livelli del file system e del IOCS.

La [Tabella 14.1](#) riassume i meccanismi fondamentali e le politiche implementate dal livello IOCS in un tipico IOCS a due livelli. I moduli che implementano le politiche determinano l'ordine in cui le operazioni di I/O devono essere eseguite per ottenere un elevato throughput del dispositivo. Questi moduli richiamano i meccanismi del IOCS fisico per eseguire le operazioni di I/O. Il livello del metodo di accesso utilizza alcuni moduli che implementano le politiche per garantire l'elaborazione efficiente dei file, inoltre utilizzano i meccanismi che implementano l'I/O a livello di file mediante l'uso dei moduli che implementano la politica del IOCS fisico. Il livello file system implementa la condivisione e la protezione dei file, utilizzando i moduli del metodo di accesso.

Si noti che la [Tabella 14.1](#) elenca solo quei meccanismi cui un livello superiore può avere accesso in modo significativo. Altri meccanismi, che sono "privati" a un livello, non sono elencati. Per esempio, i meccanismi per il *buffering* e il *blocking* dei dati di un file e per la gestione dei file cache si trovano nel livello che implementa il metodo di accesso. Tuttavia, sono disponibili solo i moduli relativi alle politiche del metodo di accesso; il file system non può accedervi direttamente. In modo simile, il IOCS fisico utilizza meccanismi per la gestione della cache del disco, cui non si ha accesso al di fuori del IOCS fisico.

|                   |                                                                                                                                                                                                                                                                                                                                                                        |
|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IOCS Fisico       | <ul style="list-style-type: none"> <li><i>Meccanismo</i>: inizializzazione dell'I/O, stato del funzionamento dell'I/O, elaborazione del completamento dell'I/O, ripristino dagli errori.</li> <li><i>Politica</i>: ottimizzazione delle prestazioni del dispositivo di I/O attraverso l'uso di uno <i>scheduler del disco</i> e una <i>cache del disco</i>.</li> </ul> |
| Metodi di accesso | <ul style="list-style-type: none"> <li><i>Meccanismo</i>: apertura e chiusura di un file, lettura e scrittura.</li> <li><i>Politica</i>: ottimizzazione delle prestazioni relative all'accesso ai file mediante <i>buffering</i> e <i>blocking</i> dei dati e utilizzo di un <i>file cache</i>.</li> </ul>                                                             |
| File System       | <ul style="list-style-type: none"> <li><i>Meccanismo</i>: allocazione dei blocchi del disco, manutenzione delle directory, impostazione e controllo delle informazioni di protezione dei file.</li> <li><i>Politiche</i>: allocazione dello spazio su disco per l'efficienza nell'accesso, condivisione e protezione dei file.</li> </ul>                              |

**Tabella 14.1** Meccanismi e politiche nel livello file system e nel livello IOCS.

## 14.2 Panoramica dell'organizzazione dell'I/O

Il Paragrafo 2.2.4 presenta una panoramica dell'organizzazione dell'I/O. Nella [Tabella 2.1](#) sono riassunti tre modi di effettuare le operazioni di I/O: modalità programmata, mediante interrupt e direct memory access (DMA). Focalizzeremo l'attenzione sul DMA per le operazioni di I/O. La [Figura 2.1](#) mostra come i dispositivi di I/O sono connessi ai controller di dispositivo che, a loro volta, sono connessi al controller DMA. Ogni controller di dispositivo ha un id numerico unico. In modo simile, ogni dispositivo a esso connesso ha un id numerico univoco. Un indirizzo di dispositivo è una coppia nella forma (*controller\_id*, *device\_id*).

Un'operazione di I/O coinvolge i seguenti dettagli:

- operazione da eseguire: read, write, ecc.;
- indirizzo del dispositivo di I/O;
- numero di byte di dati da trasferire;
- indirizzi delle aree di memoria e del dispositivo di I/O che sono coinvolti nel trasferimento dei dati.

Quando un'operazione di I/O viene eseguita in modalità DMA, la CPU avvia l'operazione di I/O, ma non è coinvolta nel trasferimento dei dati tra il dispositivo di I/O e la memoria. Per facilitare questa modalità di I/O, un'operazione di I/O viene avviata eseguendo un'*istruzione di I/O*. La CPU, il controller DMA, il controller del dispositivo e il dispositivo di I/O partecipano all'esecuzione di un'istruzione di I/O. L'istruzione punta a un insieme di *comandi di I/O* che specificano le singole operazioni coinvolte nel trasferimento dei dati. L'implementazione di un comando di I/O richiede la partecipazione del controller DMA, del controller del dispositivo e del dispositivo di I/O, ma non richiede la partecipazione della CPU. In questo modo, la CPU è libera di eseguire altre istruzioni mentre l'operazione di I/O è in corso.

Tipicamente, i comandi di I/O sono memorizzati in memoria e l'indirizzo dell'area di memoria contenente i comandi viene utilizzato come operando nell'istruzione di I/O (in alcuni computer, l'indirizzo è letto in una locazione di memoria standard quando viene eseguita l'istruzione di I/O). Quando l'istruzione di I/O è eseguita, la CPU passa questo indirizzo al controller DMA. Il controller DMA a questo punto realizza i comandi di I/O. Il prossimo esempio fornisce i dettagli di questa organizzazione.

### Esempio 14.1 - Operazioni di I/O

L'operazione di I/O per leggere i dati memorizzati in un blocco del disco con id

*(id\_traccia, id\_blocco)* è eseguita con la seguente operazione di I/O:

*I/O-init (id\_controller, id\_dispositivo), indirizzo\_comando\_I/O*

dove *indirizzo\_comando\_I/O* è l'indirizzo di partenza dell'area di memoria che contiene i seguenti due comandi.

1. Posiziona le testine del disco sulla traccia *id\_traccia*.
2. Legge il record *id\_record* nell'area di memoria con indirizzo di partenza *indirizzo\_memoria*.

La configurazione chiamata *third party DMA* funziona come segue: i controller di dispositivo sono connessi al controller DMA come mostrato in [Figura 2.1](#). Quando si esegue un'istruzione di I/O, il controller DMA passa i dettagli dei comandi dell'I/O al controller del dispositivo. Il dispositivo trasferisce i dati al controller. Il trasferimento dei dati tra il controller e la memoria è organizzato come segue: il controller del dispositivo invia un segnale DMA-request quando è pronto a eseguire il trasferimento dei dati. Alla ricezione di questo segnale, il controller DMA ottiene il controllo del bus, scrive sul bus l'indirizzo della locazione di memoria da utilizzare per il trasferimento dei dati e invia un segnale DMA-acknowledgment al controller del dispositivo. Il controller a questo punto trasferisce i dati nella o dalla memoria. Al termine del trasferimento, il controller DMA genera un interrupt di completamento dell'I/O con l'indirizzo del dispositivo come codice di interrupt. La routine di servizio degli interrupt analizza il codice per trovare il dispositivo che ha completato l'operazione di I/O ed esegue le azioni appropriate.

Poiché la CPU continua a eseguire le istruzioni mentre è in corso un'operazione di I/O, la CPU e il controller DMA sono in competizione per l'utilizzo del bus. La tecnica del *cycle stealing* garantisce a entrambi l'uso del bus senza grandi ritardi. La CPU concede l'uso del bus al controller DMA in alcuni punti precisi del ciclo di istruzione, solitamente quando sta per leggere un'istruzione o i suoi dati dalla memoria. Quando il DMA vuole trasferire dati nella o dalla memoria, aspetta finché la CPU raggiunge uno di questi punti. A questo punto sottrae un ciclo di memoria dalla CPU per eseguire il trasferimento dei dati.

Il *first party DMA* è più efficiente del *third party DMA*. In questa configurazione, il controller del dispositivo e il controller del DMA sono assemblati in una singola unità. Questa tecnica è chiamata *bus mastering*. Ottiene tassi di trasferimento dati più elevati rispetto al *third party DMA*.

### 14.3 Dispositivi di I/O

I dispositivi di I/O funzionano secondo una serie di principi, come la generazione di segnali elettromeccanici e la memorizzazione elettromagnetica o ottica dei dati. I dispositivi di I/O funzionano con diversi supporti di I/O, servono per scopi differenti e organizzano e accedono ai dati in modi differenti, per cui possono essere classificati secondo i seguenti criteri.

- *Scopo*: dispositivi di input, output e memorizzazione.
- *Tipo di accesso*: dispositivi sequenziali e ad accesso casuale.
- *Modalità di trasferimento dati*: dispositivo a caratteri e a blocchi.

L'informazione scritta (o letta) con un comando di I/O si dice che forma un *record*. Un dispositivo ad accesso sequenziale utilizza il supporto in maniera sequenziale; dunque un'operazione è sempre eseguita su un record contiguo a un record cui si è avuto accesso nell'operazione precedente. L'accesso a ogni altro record richiede comandi addizionali per saltare i record intermedi. Un dispositivo ad accesso casuale può eseguire operazioni di lettura e scrittura su un record posizionato in una qualsiasi parte del supporto di I/O. Una tastiera, un mouse, una rete e un drive a nastro sono dispositivi ad accesso sequenziale. Ai dischi si può accedere sia in maniera sequenziale che in maniera casuale.

Un'unità del supporto di I/O è chiamata *volume di I/O*; quindi, una cartuccia e un disco possono essere chiamati, rispettivamente, volume del nastro e volume del disco. Alcuni volumi di I/O sono rimovibili, per esempio, i floppy disk, i compact disk (CD) o le cartucce a nastro digitali (DAT); mentre i supporti per altri dispositivi come gli hard disk non possono essere rimossi dal dispositivo.

### **Modalità di trasferimento dei dati**

La modalità di trasferimento dei dati di un dispositivo dipende dalla velocità di trasferimento. Un dispositivo di I/O lento funziona in *modalità a carattere*, ovvero trasferisce un carattere alla volta tra la memoria e il dispositivo. Il dispositivo contiene un registro buffer che può memorizzare un carattere. Il controller del dispositivo genera un interrupt dopo che un dispositivo di input legge un carattere dal buffer o un dispositivo di output scrive un carattere nel buffer. I controller di questi dispositivi possono essere connessi direttamente al buffer. La tastiera, il mouse e la stampante sono dispositivi a caratteri.

Un dispositivo in grado di elevati tassi di trasferimento opera in *modalità a blocchi* per il trasferimento dei dati. È connesso al controller DMA. Le unità a nastro e i dischi sono dispositivi a blocchi. Un dispositivo a blocchi deve leggere o scrivere dati a una data velocità. Si possono verificare due tipi di problemi se un trasferimento viene ritardato a causa di una contesa del bus: i dati sarebbero persi durante un'operazione di lettura se il bus non fosse in grado di accettare dati da un dispositivo di I/O al tasso richiesto per il trasferimento in memoria. Un'operazione di write fallirebbe nel caso in cui il bus non fosse in grado di consegnare i dati al dispositivo di I/O al tasso richiesto.

Per prevenire i problemi dovuti alla contesa del buffer, i dati non sono trasferiti sul bus durante il funzionamento; invece, vengono trasferiti tra un dispositivo di I/O e un buffer. Durante un'operazione di input, i dati consegnati dal dispositivo di I/O sono memorizzati in un buffer nel controller DMA, che chiameremo *buffer del DMA*.

I dati sono trasferiti dal buffer del DMA alla memoria al termine dell'operazione di I/O. Per eseguire un'operazione di output, i dati da scrivere sul dispositivo di I/O vengono prima trasferiti dalla memoria al buffer del DMA. Durante l'operazione di I/O, i dati vengono trasferiti dal buffer del DMA al dispositivo di I/O. Questa organizzazione risolve i problemi dovuti alla contesa del bus.

Il trasferimento dei dati tra la CPU e un dispositivo di I/O può anche essere realizzato utilizzando la tecnica dell'I/O *mappato in memoria* (*memory-mapped I/O*). In questo approccio, un insieme di indirizzi di memoria sono riservati a un dispositivo di I/O. Questi indirizzi sono mappati in alcuni dei registri del dispositivo di I/O in modo che quando la CPU scrive dei dati in una locazione di memoria con uno degli indirizzi riservati, i dati, di fatto, vengono scritti nel corrispondente registro del dispositivo. In modo simile, quando la CPU esegue un'istruzione di lettura da una locazione di memoria con uno degli indirizzi riservati, i dati, di fatto, vengono letti dal corrispondente registro del dispositivo di I/O. In questo modo il trasferimento dei dati avviene senza un DMA, ma non impegna troppo la CPU. L'I/O mappato in memoria è implementato come segue: un dispositivo di I/O si mette in ascolto sul bus cui è connessa la memoria. Quando uno dei suoi indirizzi riservati appare sul bus, semplicemente trasferisce i dati tra il bus e il registro corrispondente all'indirizzo riservato. L'I/O mappato in memoria è utilizzato sui PC poiché non è necessario utilizzare uno speciale bus di I/O e la CPU non deve fornire nessuna istruzione speciale per avviare le operazioni di I/O e per controllare lo stato dei dispositivi di I/O. Tuttavia, è necessario più hardware sul bus di memoria per decodificare gli indirizzi riservati.

### **Tempo di accesso e tempo di trasferimento**

Usiamo la seguente notazione per le operazioni di I/O.

$t_{io}$  *tempo di I/O*, ovvero, intervallo di tempo tra l'esecuzione di un'istruzione per avviare un'operazione di I/O e il completamento dell'operazione di I/O.

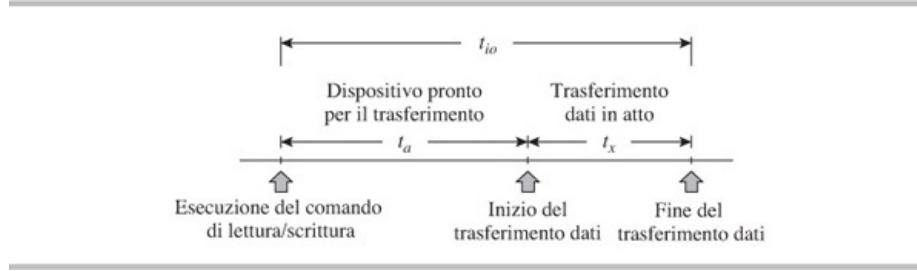
$t_a$  *tempo di accesso*, ovvero intervallo di tempo tra l'esecuzione di un comando di lettura o di scrittura e l'inizio del trasferimento dei dati.

$t_x$  *tempo di trasferimento*, ovvero tempo necessario a trasferire i dati da/a un dispositivo di I/O durante un'operazione di lettura o scrittura. È il tempo intercorso dall'inizio del trasferimento del primo byte alla fine del trasferimento dell'ultimo byte.

Il tempo di I/O per un record è la somma del tempo di accesso e del tempo di trasferimento, ovvero:

$$t_{io} = t_a + t_x \quad (14.1)$$

La [Figura 14.3](#) illustra i fattori che influenzano  $t_{io}$ . Il tempo di accesso in un dispositivo sequenziale è costante poiché il dispositivo può solo leggere o saltare un record su entrambi i lati a partire dalla sua posizione corrente. Il tempo di accesso in un dispositivo ad accesso casuale varia poiché può leggere o scrivere *qualunque* blocco in un volume di I/O, per cui deve riposizionare o la testina di lettura/scrrittura o il supporto di I/O prima di cominciare un'operazione di lettura o scrittura.

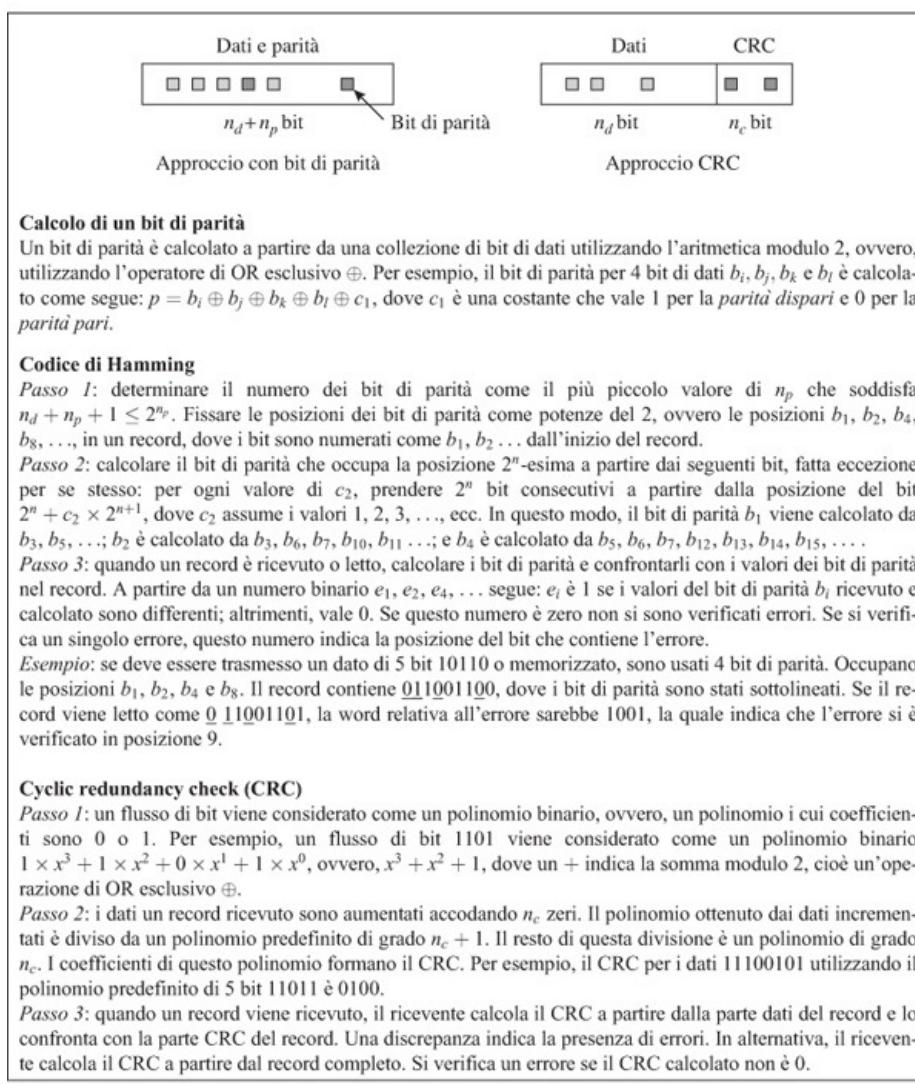


**Figura 14.3** Tempi di accesso e trasferimento in un'operazione di I/O.

### Rilevamento e correzione degli errori

Errori possono verificarsi durante la memorizzazione o la lettura dei dati o il loro trasferimento tra un supporto di I/O e la memoria. Per facilitare il rilevamento e la correzione di questi errori, i dati da memorizzare o trasmettere sono visti come un *flusso di bit*, ovvero come un flusso di 1 e 0. Codici speciali sono usati per rappresentare lo stream di bit. Illustreremo alcuni di questi codici successivamente. Il rilevamento degli errori viene eseguito memorizzando informazioni ridondanti all'interno dei dati. Queste informazioni, che chiameremo informazioni per il rilevamento degli errori, sono estratte dai dati usando una tecnica standard. Quando i dati vengono letti da un supporto di I/O, anche queste informazioni sono lette dal supporto. A questo punto, le informazioni per il rilevamento degli errori vengono ricalcolate a partire dai dati, utilizzando la stessa tecnica, e vengono confrontate con le informazioni lette dal supporto. Una discordanza indica la presenza di errori di memorizzazione. La correzione degli errori è eseguita in modo analogo, fatta eccezione per il fatto che vengono usati algoritmi più robusti per generare le informazioni per la correzione degli errori. Queste informazioni possono sia rilevare un errore che indicare come correggerlo. La memorizzazione e la lettura di informazioni ridondanti causa overhead. La correzione degli errori genera maggiore overhead rispetto al solo rilevamento degli errori.

La [Figura 14.4](#) descrive due approcci al rilevamento e alla correzione degli errori. Nell'approccio con *bit di parità*,  $n_p$  bit di parità sono calcolati per  $n_d$  bit di dati. I bit di parità sono inseriti in locazioni prefissate di un record. Questi bit sono indistinguibili dai dati, tranne dall'algoritmo di rilevamento/correzione degli errori. Nell'approccio *cyclic redundancy check* (CRC), un numero di  $n_c$  bit chiamato CRC viene memorizzato nel campo CRC di un record. Una differenza chiave tra i due approcci è che  $n_p$  dipende da  $n_d$ , mentre  $n_c$  è indipendente da  $n_d$ .



**Figura 14.4** Approcci al rilevamento e alla correzione degli errori.

Entrambi gli approcci utilizzano l'aritmetica modulo 2. Questa aritmetica è analoga all'aritmetica binaria, eccetto per il fatto che ignora i riporti generati nelle posizioni dei bit. Questa proprietà la rende molto veloce. Un'addizione modulo 2 viene realizzata con un'operazione di OR esclusivo  $\oplus$ . Usa le seguenti regole:  $0 \oplus 0 = 0$ ,  $1 \oplus 0 = 1$ ,  $0 \oplus 1 = 1$ , e  $1 \oplus 1 = 0$ .

Un'altra variante all'approccio con bit di parità, utilizzata nelle RAM e nei vecchi nastri magnetici associa un singolo bit di parità a un byte di dati. Come descritto in [Figura 14.4](#), viene generato a partire da tutti i bit di un byte utilizzando l'operazione  $\oplus$ . Può rilevare un singolo errore in un byte, ma fallisce se si verificano due errori. Inoltre non può correggere qualunque errore. L'overhead dovuto al rilevamento degli errori è di 1 bit di parità per 8 bit di dati, ovvero il 12.5 per cento. Un codice di Hamming può rilevare fino a due errori in un record e può correggere un singolo errore. Il nome tecnico corretto del codice è codice di Hamming ( $n_d + n_p, n_d$ ). Il confronto dei valori dei bit di parità in un record letto da un supporto con valori di parità calcolati a partire dai dati letti applicando le regole del codice indica quale bit contiene l'errore. Il valore in questo bit è invertito per correggere l'errore. La [Figura 14.4](#) mostra le regole per determinare il numero dei bit di parità e per calcolarne i valori. Un codice di Hamming (12,8) può eseguire il rilevamento e la correzione degli errori per un 1 byte. Utilizza 12 - 8, ovvero 4 bit di parità. Dunque, l'overhead è del 50 per cento. L'overhead diminuisce con il numero di bit di dati; per esempio, 8 bit di parità sono adeguati per 30 byte di dati.

Il CRC viene calcolato a partire dai dati che devono essere trasmessi o memorizzati ed è inserito nel campo CRC di un record. Può indicare se si sono verificati uno o più errori

in un *qualsiasi* byte dei dati, o se i byte sono stati swappati o memorizzati. Quando viene letto un record, si calcola un CRC a partire dai dati e lo si confronta con il numero contenuto nel campo CRC. Si è verificato un errore se non c'è corrispondenza tra i due. Un valore pratico  $n_c$  è di 16 o 32 bit, indipendentemente dal valore di  $n_d$ . Con  $n_c < n_d$ , il rilevamento non è infallibile poiché due flussi di bit, per esempio  $s_1$  ed  $s_2$ , potrebbero generare lo stesso CRC. Se uno dei due è trasformato nell'altro in conseguenza di errori, gli errori non possono essere rilevati utilizzando il CRC. La probabilità che questo accada è  $\frac{1}{2^{n_c}}$ . Dunque, l'affidabilità del CRC è  $1 - \frac{1}{2^{n_c}}$ . Per CRC a 16 bit, l'affidabilità è del 99.9985 per cento. Per un CRC a 32 bit, l'affidabilità è del 99.9999 per cento.

### 14.3.1 Nastri magnetici

Il supporto di I/O in un nastro o una cartuccia è una striscia di materiale magnetico su cui sono memorizzate le informazioni nella forma di 1 e 0, in base ai principi dell'elettromagnetismo. La memorizzazione su di un nastro è multi traccia; ogni traccia memorizza un bit di un byte o un bit di parità. Una testina di lettura-scrittura viene posizionata su ogni traccia. I drive a nastro sono dispositivi ad accesso sequenziale. Le operazioni che possono essere effettuate su questi dispositivi sono: *read/write* di un numero di byte specificato, *skip* e *rewind*. A causa della natura sequenziale, le cartucce a nastro e i DAT sono tipicamente usati per l'archiviazione dei dati, che consiste nella lettura o nella scrittura di tutti i record del supporto.

Nei nastri più vecchi, i record adiacenti su un nastro sono separati da un *gap inter record*. Questo gap è utile per il movimento di avvio-arresto del supporto tra la lettura o la scrittura di record successivi. Il *tempo di accesso* ( $t_a$ ) durante un'operazione di lettura o scrittura è generato sia dalla necessità di ottenere una velocità di spostamento uniforme del supporto di I/O prima che il trasferimento dei dati possa essere avviato sia dalla necessità di posizionare il prossimo record sotto la testina di lettura scrittura. Il tempo totale di I/O per un record di dimensione  $s$  byte è dato dalla formula:

$$t_{io} = t_a + \frac{s}{d \times v}$$

dove  $d$  densità di memorizzazione;

$v$  velocità del supporto di I/O.

I gap inter record causano notevoli penalizzazioni - portano a uno scarso utilizzo del supporto di memorizzazione e rallentano le attività di elaborazione dei file. Nonostante lo svantaggio dello scarso utilizzo del supporto di memorizzazione, negli anni '90 i nastri avevano un costo per megabyte pari a un decimo di quello degli hard disk. Tuttavia, i nastri hanno perduto questa convenienza nel decennio successivo poiché la tecnologia dei dischi ha fatto rapidi progressi e i dischi di grandi dimensioni sono diventati pratici ed economici. Per riacquisire il vantaggio del costo, è stata sviluppata una tecnologia di streaming basata su nastro.

Uno *streaming tape* contiene un singolo record che viene memorizzato senza interruzioni indipendentemente dalla sua dimensione. Dunque non esistono gap interrecord anche quando viene memorizzata su nastro una grande quantità di dati. Un dispositivo per streaming tape contiene un buffer. Un'operazione di scrittura viene avviata dopo aver inserito dei dati nel buffer. Il dispositivo scrive i dati dal buffer sul nastro. Per consentire allo streaming tape di lavorare a piena velocità, è importante inserire nuovi dati a una velocità pari a quella di scrittura del nastro. Il drive a nastro termina la scrittura quando trova il buffer vuoto. Quando vengono inseriti nuovi dati nel buffer, il drive a nastro riprende l'operazione di scrittura. Per evitare la creazione di gap inter record, il nastro viene spostato indietro e poi nuovamente avanti in modo da raggiungere la velocità di memorizzazione nel momento in cui la testina passa sull'ultimo bit che è stato scritto. A questo punto riprende la scrittura. Operativamente, il ripristino della scrittura consuma pochi millisecondi.

Lo streaming tape fornisce un tasso di trasferimento molto elevato se il buffer non è mai vuoto. Tuttavia, se il nastro si ferma di frequente, la velocità di scrittura effettiva può scendere a un valore molto più basso. Il IOCS fisico deve garantire che questo non accada. L'operazione di arresto-avvio-ripristino del nastro richiede anche un posizionamento e un allineamento preciso, che rende lo streaming tape costoso.

### 14.3.2 Dischi magnetici

L'elemento di memorizzazione fondamentale di un disco magnetico è un sottile oggetto circolare chiamato *piatto*, che ruota sul suo asse. Le superfici circolari di un piatto sono ricoperte con un materiale magnetico. Un singola testina di lettura-scrrittura scrive e legge da una superficie, per cui un byte viene memorizzato in modo seriale lungo una *traccia* circolare sulla superficie del disco. La testina di lettura-scrrittura può muoversi radialmente sul piatto. Per ogni posizione della testina, l'informazione memorizzata forma una traccia circolare separata. L'informazione di parità non viene utilizzata in un disco; un CRC è scritto insieme a ogni record per supportare il rilevamento degli errori.

Una posizione di inizio traccia è segnata su ogni traccia e ai record di una traccia sono assegnati dei numeri seriali a partire da questo punto. Il disco può accedere a ogni record il cui indirizzo è specificato dalla coppia (*numero di traccia, numero di record*). Il tempo di accesso per un record di un disco è dato da:

$$t_a = t_s + t_r + t_t \quad (14.2)$$

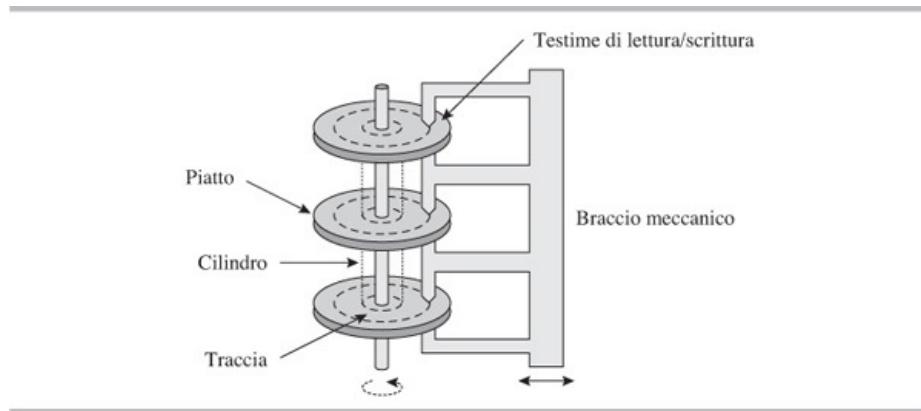
dove  $t_s$  *tempo di ricerca*, ovvero, il tempo necessario per posizionare la testina sulla traccia richiesta;

$t_r$  *latenza rotazionale*, ovvero, il tempo per accedere al record selezionato sulla traccia;

$t_t$  *tempo di trasferimento*, ovvero, il tempo necessario per trasferire un numero di blocchi (o settori) da leggere/scrivere.

Il tempo di ricerca è il tempo necessario per il movimento meccanico della testina. La latenza rotazionale deriva dal fatto che un'operazione di I/O può cominciare solo quando il record richiesto sta per passare sotto la testina. La latenza rotazionale media è il tempo impiegato per mezza rivoluzione del disco. Valori rappresentativi per la latenza rotazionale media sono 3-4 ms, per i tempi di ricerca sono nell'intervallo di 5-15 ms e per i tassi di trasferimento dati sono dell'ordine delle decine di megabyte al secondo.

Le variazioni nell'organizzazione del disco sono state motivate dalla necessità di ridurre il tempo di accesso al disco, di incrementarne la capienza e il tasso di trasferimento dei dati e di ridurne il prezzo. Il disco più economico è un floppy disk che è lento ed ha una capienza ridotta. Un hard disk ha un'elevata capienza; capienze ancora maggiori sono ottenute montando molti piatti sullo stesso perno. Una testina di lettura-scrrittura è usata per ogni superficie circolare di un piatto, ovvero una testina sopra e una sotto ogni piatto. Tutte le testine nel disco sono montate su un singolo braccio, chiamato *attuatore*, per cui in ogni momento tutte le testine sono posizionate sulle stesse tracce di differenti superfici. L'insieme di queste tracce forma un *cilindro* (Figura 14.5), una forma che può essere sfruttata per l'organizzazione dei dati. Tutte le tracce in un cilindro sono accessibili dalla stessa posizione del braccio; in questo modo, i cilindri consentono di accedere a diverse tracce del disco senza richiedere alcun movimento delle testine e dunque le operazioni di I/O sui record posizionati nello stesso cilindro possono essere eseguite senza dover attendere i tempi di ricerca.



**Figura 14.5** Un disco.

Un disco può essere considerato come un insieme di cilindri concentrici, dal più

interno al più esterno. L'indirizzo di un record può così essere specificato dalla tripla (*numero di cilindro, numero della superficie, numero del record*). I comandi necessari per il funzionamento di un disco sono *seek* (*numero di cilindro, numero della superficie*) e *read/write* di un record specifico.

La capienza del disco può essere aumentata incrementando il numero di piatti. Tuttavia, un numero maggiore di piatti richiede più testine, che a loro volta richiedono un attuatore più pesante e impongono un maggiore stress meccanico. Dunque i dischi tendono ad avere solo pochi piatti. Quando è necessaria una grande capienza, le applicazioni usano più dischi. (Nel Paragrafo 14.3.5, discuteremo di come l'uso di più dischi può anche essere sfruttato per ottenere elevati tassi di trasferimento dati e alta affidabilità). I tempi di ricerca possono essere ridotti utilizzando velocità di rotazione più elevate, ma le alte velocità fanno aumentare il costo delle componenti meccaniche per cui i dischi veloci tendono ad avere piatti più piccoli per compensare i costi. I PC e i desktop generalmente utilizzano dischi più economici. Questi dischi hanno piatti più grandi, che offrono maggiore capienza, ma minori velocità rotazionali. Di contro, i server tendono a utilizzare dischi più costosi che sono più piccoli e ruotano più velocemente.

Per ottimizzare l'uso della superficie del disco, le tracce sono organizzate in *settori*. Un settore è uno "slot" di dimensione standard in una traccia per un record del disco. La dimensione del settore viene scelta per garantire il minimo spreco della capienza di memorizzazione dovuto ai gap inter record sulla superficie. La divisione in settori può essere eseguita dal hardware del disco (divisione hard), o può essere implementata via software (divisione soft).

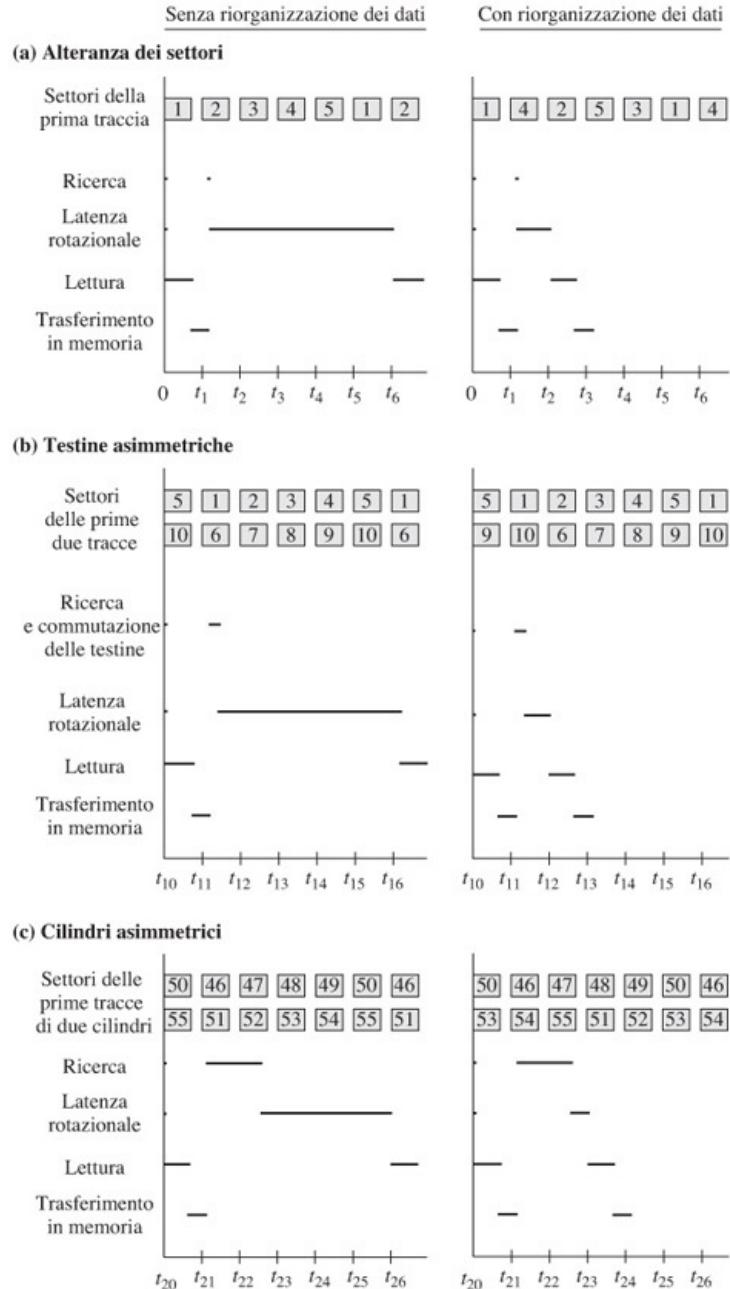
### 14.3.3 Tecniche di organizzazione dei dati

Si ricordi dal Paragrafo 14.3 che i dati letti da un dispositivo di I/O durante un'operazione di lettura sono memorizzati nel buffer del DMA e da qui il DMA li trasferisce in memoria come blocco singolo. Ma mentre è in atto questo trasferimento, il disco continua a ruotare e uno o più dei settori seguenti possono passare sotto le testine fino al momento in cui il trasferimento è completato. Dunque se viene eseguita immediatamente un'operazione di lettura sul settore consecutivo, il settore richiesto può essere già passato sotto la testina nel momento in cui il DMA può avviare l'operazione di lettura. Questa operazione di lettura può essere eseguita solo nella successiva rivoluzione del disco. Analogamente, durante un'operazione di scrittura, la memorizzazione dei dati è avviata solo dopo che i dati sono stati trasferiti dalla memoria al buffer del DMA, per cui la memorizzazione nel settore successivo non può avvenire nella stessa rivoluzione se il settore passa sotto la testina prima che il trasferimento dei dati sia completato. Un problema simile è causato dal *tempo di commutazione della testina*, ovvero il tempo necessario per commutare l'accesso alle testine posizionate su piatti differenti. Durante questo tempo pochi settori del piatto successivo sono passati sotto la testina di lettura-scrittura. Il tempo di ricerca necessario per spostare la testina sul prossimo cilindro genera un problema simile. Tutti questi problemi in maniera differente influenzano il throughput del disco.

Le tecniche di alternanza dei settori, della testina asimmetrica e del cilindro asimmetrico risolvono i problemi causati, rispettivamente, dal tempo di trasferimento dei dati, dal tempo di commutazione della testina e dal tempo di ricerca. Queste tecniche, collettivamente chiamate tecniche di *organizzazione dei dati*, assicurano che il settore con il numero successivo non passerà sotto la testina di lettura-scrittura prima che la testina sia in posizione per eseguire un'operazione di lettura/scrittura su di esso, in modo che l'operazione possa essere eseguita nell'attuale rivoluzione del disco. L'*alternanza dei settori* organizza i settori lungo una traccia in modo tale che i settori con numeri successivi siano separati da alcuni altri settori. Questa organizzazione consente di terminare l'operazione di I/O su un settore prima che il settore con indirizzo successivo passi sotto la testina. La tecnica della *testina asimmetrica* organizza le posizioni di "inizio traccia" su differenti piatti di un cilindro in modo che i tempi in cui l'ultimo settore di una traccia e il primo settore della traccia successiva passano sotto le rispettive testine siano separati dal tempo di commutazione della testina. La tecnica della *testina asimmetrica* analogamente organizza le posizioni di "inizio traccia" su cilindri consecutivi per tenere conto del tempo di ricerca successivo alla lettura dell'ultimo settore di un cilindro.

La [Figura 14.6](#) illustra come le tecniche di alternanza dei settori, delle testine asimmetriche, dei cilindri asimmetrici riducono il ritardo rotazionale mediante la riorganizzazione dei dati. Si assume che il disco abbia cinque settori in una traccia e

utilizzi dieci piatti; dunque un cilindro è composto di cinquanta settori. Per ogni tecnica, la parte sinistra e la parte destra della figura mostrano il funzionamento del disco con e senza la riorganizzazione dei dati. La prima riga in ogni parte mostra quale settore passa sotto le testine di lettura-scrittura del disco in momenti differenti. Le righe successive mostrano quale attività dell'operazione di I/O sono in esecuzione durante la rotazione del disco - costituiscono un diagramma dei tempi per l'operazione di I/O.



**Figura 14.6** Effetto della riorganizzazione dei dati: (a) alternanza dei settori, (b) testine asimmetriche e (c) cilindri asimmetrici.

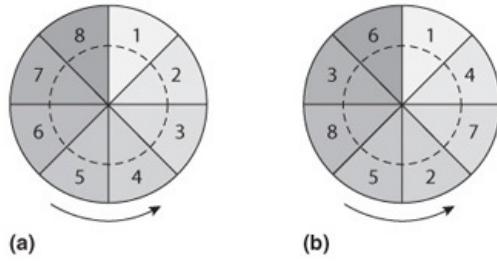
La [Figura 14.6\(a\)](#) illustra la tecnica dei settori asimmetrici. Assumiamo che la testina del disco sia posizionata immediatamente prima del primo settore sul primo cilindro in cui è memorizzato un file, in modo che il comando per leggere il primo settore non debba attendere né il tempo di ricerca né quello rotazionale. La lettura del settore nel buffer del DMA viene completata poco prima del tempo  $t_1$  e il trasferimento di questi dati in memoria dal controller DMA è completato poco dopo il tempo  $t_1$ . Il comando per leggere

il settore successivo viene eseguito immediatamente dopo il completamento della lettura del settore precedente, ovvero, poco dopo il tempo  $t_1$ . In quel momento la testina è posizionata da qualche parte sul settore 2, per cui il settore 2 non può essere letto immediatamente. Si verifica un ritardo rotazionale che dura finché il settore 2 non passa sotto la testina nella successiva rivoluzione, ovvero al tempo  $t_6$ . La parte destra della figura mostra l'organizzazione dei settori quando viene adottata l'alternanza dei settori; i settori 1 e 2 sono separati sulla traccia dal settore 4. Quando viene eseguito il comando per leggere il settore 2, la testina di lettura-scrittura è posizionata quasi alla fine del settore 4. In questo caso, la latenza rotazionale dura solo fino al tempo  $t_2$ , momento in cui il settore 2 passa sotto la testina.

La [Figura 14.6\(b\)](#) illustra la tecnica della testina asimmetrica. Mostriamo la configurazione dei settori nelle prime due tracce allocate a un file. Il comando di lettura del settore 5, l'ultimo settore della prima traccia, viene lanciato al tempo  $t_{10}$ . La lettura di questo settore e il trasferimento dei dati in memoria sono completati prima del tempo  $t_{11}$ . Tuttavia, è necessaria la commutazione della testina poiché al tempo  $t_{11}$  si trova su una traccia differente; la commutazione della testina non è stata completata prima del tempo  $t_{11}$  quando il settore 6 inizia a passare sotto la testina. Per cui la lettura del settore 6 non può essere iniziata immediatamente; è necessario attendere finché il settore 6 non passa sotto la testina durante la rivoluzione successiva del disco al tempo  $t_{16}$ . Questo ritardo rotazionale è ridotto spostando la memorizzazione sulla seconda traccia di un settore, come mostrato nella parte destra della figura. Ora, la lettura del settore 6 può cominciare al tempo  $t_{12}$ , provocando un ritardo rotazionale molto più piccolo. La [Figura 14.6\(c\)](#) illustra la tecnica dei cilindri asimmetrici. Mostriamo l'organizzazione dei settori nella prima traccia dei primi due cilindri allocati al file. L'operazione di ricerca per leggere il settore 51 risulta nel movimento della testina di lettura-scrittura di un cilindro. L'operazione di ricerca viene completata poco prima del tempo  $t_{23}$ ; tuttavia, il settore 51 è passato sotto la testina di lettura-scrittura in quel momento, dunque si genera un ritardo rotazionale finché il settore 51 non passa sotto la testina durante la successiva rivoluzione al tempo  $t_{26}$ . Come mostrato nella parte destra della figura, lo spostamento dei dati di due settori consente al settore 51 di essere letto a partire dal tempo  $t_{23}$ .

L'alternanza dei settori ha un forte impatto sul throughput dei vecchi dischi. I dischi moderni sono dotati di controller che trasferiscono dati per e dalla memoria a tassi molto elevati, per cui l'alternanza dei settori non è necessaria. Tuttavia, parliamo di questa tecnica perché fornisce un approfondimento all'ottimizzazione del throughput dei dischi mediante l'organizzazione dei dati. Le testine e i cilindri asimmetrici sono ancora usati per ottimizzare le prestazioni dei dischi.

La [Figura 14.7](#) illustra l'alternanza dei settori. Il fattore di alternanza ( $F_{int}$ ) è il numero di settori che separano settori successivi sulla stessa traccia del disco. La parte (b) della [Figura 14.7](#) illustra la configurazione quando  $F_{int} = 2$ , ovvero settori consecutivi sono separati da altri due settori. L'alternanza è uniforme, ovvero ogni coppia di settori consecutivi è separata dallo stesso numero di settori, se  $n - 1$  o  $n + 1$  sono un multiplo di  $F_{int+1}$ , dove  $n$  è il numero di settori per traccia. La configurazione nella figura, dove ci sono 8 settori per traccia, è uniforme, mentre l'alternanza con  $F_{int} = 1$  o 3 non è uniforme (vedi la seconda colonna nella [Tabella 14.2](#) – alcuni settori consecutivi sono separati da più di  $F_{int}$  settori). Come vedremo nell'Esempio 14.2, si incorre in una penalizzazione delle prestazioni quando l'alternanza non è uniforme.



**Figura 14.7** Settori in una traccia del disco: (a) senza alternanza; (b) con fattore di alternanza = 2.

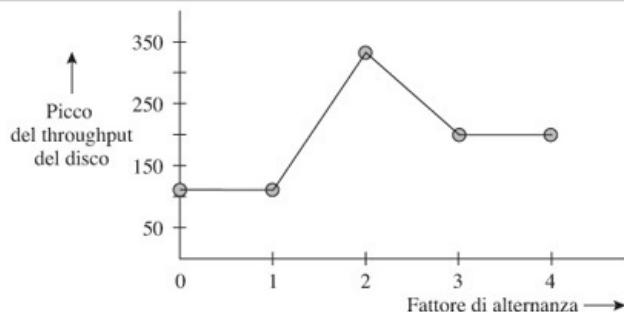
| $F_{int}$ | Configurazione dei settori | $t_{io}$ per i settori (ms)  | $t_{io}$ medio (ms) | Picco del throughput (kB/s) |
|-----------|----------------------------|------------------------------|---------------------|-----------------------------|
| 0         | 1, 2, 3, 4, 5, 6, 7, 8     | 9, 9, 9, 9, 9, 9, 9, 9       | 9                   | 111.1                       |
| 1         | 1, 5, 2, 6, 3, 7, 4, 8     | 9, 3, 10, 10, 10, 10, 10, 10 | 9                   | 111.1                       |
| 2         | 1, 4, 7, 2, 5, 8, 3, 6     | 3, 3, 3, 3, 3, 3, 3, 3       | 3                   | 333.3                       |
| 3         | 1, 3, 5, 7, 2, 4, 6, 8     | 9, 5, 5, 5, 4, 4, 4, 4       | 5                   | 200.0                       |
| 4         | 1, 6, 3, 8, 5, 2, 7, 4     | 5, 5, 5, 5, 5, 5, 5, 5       | 5                   | 200.0                       |

**Tabella 14.2** Configurazione dei settori e prestazioni nell'alternanza dei settori.

Sia  $t_{st}$  il tempo necessario per trasferire un settore tra il controller del DMA e la memoria e sia  $t_{sect}$  il tempo necessario affinché un settore passi sotto la testina del disco. Le prestazioni ottimali si ottengono se  $t_{si} = F_{int} \times t_{sect}$ , poiché l'I/O sul prossimo settore può essere avviato immediatamente dopo che il DMA ha terminato il trasferimento del settore precedente. Se  $t_{st} > F_{int} \times t_{sect}$  il prossimo settore passa sotto la testina prima che il DMA finisca il trasferimento dei dati del settore precedente. Dunque si può accedere al settore successivo solo durante la successiva rivoluzione del disco.  $t_{st} < F_{int} \times t_{sect}$  implica che il disco resta idle per del tempo prima che si acceda al prossimo settore nella stessa rivoluzione. Il throughput ne risente in entrambi questi casi. Analogamente, il throughput ne risentirebbe, nel caso delle altre tecniche di riorganizzazione dei dati, se i dati fossero spostati troppo o troppo poco. L'esempio seguente illustra come varia il picco di throughput del disco al variare del fattore di alternanza.

### Esempio 14.2 - Alternanza dei settori

Un disco completa una rivoluzione in 8 ms ed ha 8 settori per traccia, ognuna contenente 1000 byte. I valori di  $t_{st}$  e  $t_{sect}$  soddisfano la relazione  $t_{sect} < t_{st} < 2 \times t_{sect}$ . Per ottenere il picco di throughput del disco con un valore di  $F_{int}$ , leggiamo i settori nell'ordine 1, ..., 8 continuamente e osserviamo il numero di byte trasferiti in un secondo. La **Figura 14.8** mostra la variazione del picco di throughput del disco per differenti valori di  $F_{int}$ .



**Figura 14.8** Variazione del throughput al variare del fattore di alternanza.

La [Tabella 14.2](#) mostra la configurazione dei settori e del throughput del disco per valori differenti di  $F_{int}$  e i corrispondenti throughput del disco rappresentati in kB/s dove 1 kB/s è 1000 byte al secondo. L'alternanza con  $F_{int} = 1$  o 3 non è uniforme. Per  $F_{int} = 1$  la configurazione dei settori su una traccia è 1, 5, 2, 6, 3, 7, 4, 8. Dopo la lettura del settore 1, il settore 2 non può essere letto durante la stessa rivoluzione. Dunque il disco impiega 10 ms per leggere il settore 2. In modo simile, i settori 3 e 4 richiedono 10 ms. I settori 4 e 5 sono separati da due settori. Dunque possono essere letti durante la stessa rivoluzione del disco; il disco impiega solo 3 ms a leggere il settore 5 dopo che è stato letto il settore 4. La lettura dei settori 6, 7 e 8 richiede 10 ms per ognuno, mentre la lettura del settore 1 richiede 9 ms.

La [Figura 14.8](#) mostra la variazione del throughput per differenti valori di  $F_{int}$ .  $F_{int} = 2$  è adeguato per soddisfare  $t_{st} \leq F_{int} \times t_{sect}$ , per cui il throughput aumenta nettamente. I valori di  $F_{int} > 2$  sono controproducenti poiché il disco passa del tempo idle prima che il prossimo settore passi sotto la testina. Dunque il throughput diminuisce per  $F_{int} > 2$ .

#### 14.3.4 Tecnologie di collegamento dei dischi

##### Interfacce EIDE e SCSI

*Enhanced integrated device electronics* (EIDE) e *small computer system interconnect* (SCSI) sono le principali interfacce per il collegamento dei dischi ai computer, da cui i termini dischi EIDE e, rispettivamente dischi SCSI. I dischi collegati in questo modo sono chiamati *host-attached storage*. *Integrated device electronics* (IDE, o anche “advanced technology attachment”, ATA) è stato predecessore di EIDE. Prima che EIDE fosse sviluppato, le differenti caratteristiche dei dischi IDE e SCSI li rendevano ideali per applicazioni specifiche. Per esempio, il disco IDE era considerato eccellente per l’I/O sequenziale, mentre SCSI era superiore per l’I/O random. Di conseguenza, i dischi IDE erano utilizzati nei PC di fascia bassa e negli ambienti desktop mentre i dischi SCSI erano usati negli ambienti server. Con EIDE, il gap delle prestazioni nell’accesso causale si è ridotto notevolmente. Entrambi mantengono le aree tradizionali, ma ora EIDE e SCSI competono in alcuni segmenti di mercato, come quello dei supporti di memorizzazione per il backup. Entrambi i tipi di dischi sono dotati di un grande buffer di alcuni megabyte.

I dischi IDE funzionano principalmente in modalità I/O programmato, sebbene supportino la modalità DMA. EIDE supporta nuove modalità DMA inclusa la prima parte, ovvero il bus mastering; la modalità ultra ATA di EIDE supporta tassi di trasferimento di 33.3 MB al secondo, 8 volte più veloce del tasso di trasferimento IDE. I dischi EIDE utilizzano piatti più grandi, ruotano più lentamente e sono economici. Ad un EIDE possono essere connessi fino a due dischi; tuttavia, può esserne attivo solo uno alla volta.

SCSI supporta diverse modalità DMA; la più veloce fornisce un tasso di trasferimento dati di 80 MB al secondo. SCSI consente di avere fino a 7 dischi connessi. SCSI è chiamata interfaccia, ma tecnicamente si tratta di un bus di I/O poiché consente il funzionamento simultaneo di molti dischi. I dischi SCSI sono più piccoli, ruotano più velocemente e sono più costosi. Dunque, hanno tempi di ricerca più brevi e tassi di trasferimento più elevati. Un disco SCSI supporta l’I/O *scatter/gather* con cui può trasferire dati da un blocco del disco in aree di memoria non contigue o collezionare dati da aree non contigue e scriverli in un blocco del disco (Paragrafo 12.2.4). Inoltre fornisce diverse funzionalità che erano tradizionalmente eseguite dal IOCS, incluse le seguenti.

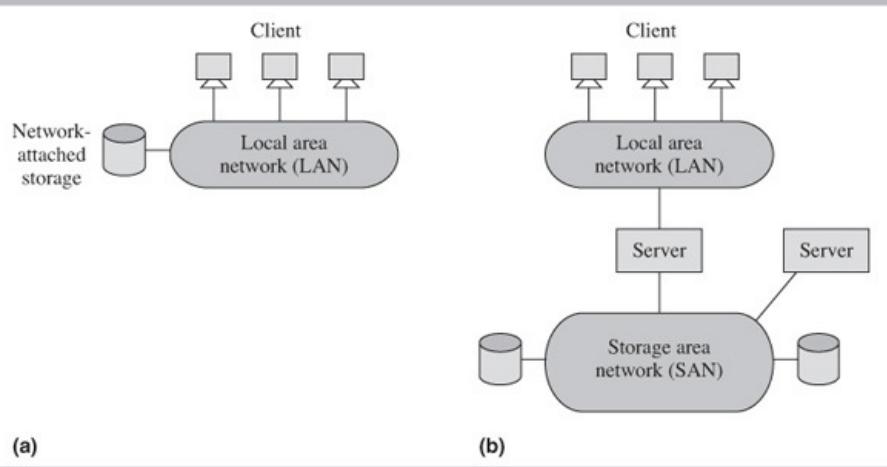
- *Scheduling del disco*: un disco SCSI accetta diverse richieste di I/O in maniera concorrente e le memorizza in una coda di richieste. Sfrutta la conoscenza dell’attuale posizione delle testine del disco e la posizione rotazionale dei piatti per selezionare un’operazione di I/O che incorra nel minor ritardo in conseguenza della latenza di ricerca e rotazione. Questa caratteristica è descritta nel Paragrafo 14.7.
- *Ripristino dei blocchi danneggiati*: un disco SCSI rileva i blocchi danneggiati e gli assegna dei blocchi sostitutivi. Mantiene una tabella che mostra gli indirizzi dei blocchi danneggiati e dei rispettivi sostituti. Se un comando di I/O coinvolge l’uso di un settore danneggiato, il disco lo redirige automaticamente al blocco sostitutivo. Questa caratteristica velocizza le operazioni di I/O eseguendo una gestione dei blocchi danneggiati nel dispositivo piuttosto che nel livello del metodo di accesso dell’IOCS.
- *Pre-estrazione dei dati*: un disco SCSI contiene un buffer. Ad ogni operazione di I/O,

legge alcuni blocchi successivi e li memorizza nel buffer. Questa operazione velocizza le successive operazioni di lettura durante l'elaborazione di un file sequenziale.

### **Network-attached storage e storage area network**

I dischi collegati a un host presentano un problema di scalabilità poiché la dimensione dei dischi è limitata dal progresso tecnologico e il numero di dischi che possono essere collegati a un host è limitato dall'interfaccia. Per questo motivo, è necessario costantemente sostituire i dischi o aggiungere più server per soddisfare le richieste di spazio di memorizzazione. Questo problema viene risolto favorendo l'uso di dischi remoti collegati in rete. Questo approccio permette di avere una capacità di memorizzazione incrementabile e che inoltre può essere condivisa da applicazioni in esecuzione su molti server.

Un *network-attached storage* (NAS) è un disco o un *redundant array of inexpensive disks* (RAID), che verrà discusso nel prossimo paragrafo, collegato direttamente a un rete locale (LAN) [Figura 14.9(a)]. NAS è un metodo economico per fornire grandi capacità di memorizzazione, poiché utilizza le componenti hardware e il software presenti in una LAN. Un file server o un file system distribuito (Capitolo 20) possono essere realizzati utilizzando il NAS. Tuttavia, l'uso del NAS nella pratica presenta alcune difficoltà: le LAN utilizzano protocolli che ottimizzano i trasferimenti tra applicazioni mentre il file server o il file system distribuito utilizzano protocolli basati su file, come per esempio il protocollo NFS della SUN discusso nel Paragrafo 19.6.1, o il protocollo della Microsoft "common interface file system" (CIFS). Il carico creato dal protocollo basato su file rallenta le applicazioni di rete.



**Figura 14.9** (a) Network-attached storage; (b) storage area network.

Una *storage area network* (SAN) è una configurazione alternativa che evita il rallentamento delle applicazioni di rete. Una SAN è una rete composta di dischi che fornisce un grande ampiezza di banda [Figura 14.9(b)]. La rete potrebbe essere un canale dedicato in fibra ottica che utilizza il protocollo SCSI, o una rete basata su IP che utilizza il protocollo iSCSI. Diversi server possono essere connessi a una SAN; ogni server può accedere a tutta l'area di memorizzazione. Questa caratteristica facilita la creazione di *cluster* di computer ad alte prestazioni (Paragrafo 16.2). L'integrità e la disponibilità dei dati è garantita dalla ridondanza dei dischi e dei server connessi alla SAN.

Stanno emergendo nuove tecnologie che adottano il protocollo iSCSI su rete IP per combinare le caratteristiche delle tecnologie NAS e SAN. Queste tecnologie supportano sia dispositivi SAN con accesso ai blocchi che dispositivi NAS per l'accesso ai file senza la necessità del canale in fibra.

### **14.3.5 RAID**

Gli utenti dei computer necessitano di dischi con maggiore capienza, accesso ai dati più veloce, tassi di trasferimento maggiori e maggiore affidabilità. Tutti questi requisiti sono risolti con configurazioni che coinvolgono più dischi. La tecnologia *redundant array of inexpensive disks* (RAID) fu adottata originariamente per ottenere un grande spazio di

memorizzazione a basso costo utilizzando dischi economici. Tuttavia, il recente andamento è di aumentare lo spazio di memorizzazione utilizzando NAS e SAN (Paragrafo 14.3.4). Dunque l'odierna tecnologia RAID viene utilizzata per ottenere accessi veloci, tassi di trasferimento elevati e alta affidabilità; per questo motivo è chiamata più propriamente *redundant array of independent disks*.

La tecnologia RAID distribuisce i dati coinvolti in un'operazione di I/O su diversi dischi ed esegue le operazioni di I/O su questi dischi in parallelo. Questa caratteristica può fornire accessi veloci o alti tassi di trasferimento, in base alla configurazione adottata. L'alta affidabilità è ottenuta memorizzando informazioni ridondanti; tuttavia, la ridondanza adottata in un RAID è qualitativamente differente da quella adottata nei dischi convenzionali: un disco convenzionale fornisce affidabilità solo scrivendo un CRC alla fine di ogni record (Paragrafo 14.3), mentre le tecniche di ridondanza in un RAID adottano più dischi per memorizzare informazioni ridondanti così che i dati possano essere ripristinati anche quando alcuni dischi risultano danneggiati. L'accesso alle informazioni ridondanti non necessita di tempo di I/O aggiuntivo poiché si può accedere in parallelo sia ai dati che alle informazioni ridondanti.

La memorizzazione in un RAID viene eseguita come segue: una *disk strip* è una unità di dati su un disco, che può essere un settore, un blocco o una traccia. Le strip posizionate in maniera identica su differenti dischi formano una *disk stripe*. A un file viene allocato un numero intero di stripe. I dati memorizzati nelle stripe della stessa stripe possono essere letti o scritti simultaneamente poiché si trovano su dischi differenti. Se l'array di dischi contiene  $n$  dischi, teoricamente il tasso di trasferimento potrebbe essere  $n$  volte quello di un singolo disco. I valori reali dei tassi di trasferimento dipendono dall'overhead e da alcuni fattori che possono limitare il parallelismo delle operazioni di I/O durante l'elaborazione di un file.

Sono state proposte diverse configurazioni RAID che utilizzano differenti tecniche di ridondanza e organizzazione di disk striping. Queste configurazioni sono chiamate *livelli RAID*. La [Tabella 14.3](#) riassume le proprietà dei vari livelli RAID. I livelli RAID 0 + 1 e 1 + 0, che sono configurazioni ibride basate sui livelli 0 e 1, e il livello RAID 5 sono le configurazioni RAID più utilizzate.

| Livello   | Tecnica                                                                                                                     | Descrizione                                                                                                                                                                                                                                                                                                                                                                       |
|-----------|-----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Livello 0 | Disk striping<br>                          | I dati sono alternati su diversi dischi. Durante un'operazione di I/O, l'accesso ai dischi avviene in parallelo. Potenzialmente, utilizzando $n$ dischi questa configurazione può garantire un incremento dell'ordine di $n$ nel trasferimento dei dati.                                                                                                                          |
| Livello 1 | Disk mirroring<br>                         | Gli stessi dati sono memorizzati su due dischi. Durante la lettura dei dati, viene utilizzata la copia accessibile più velocemente. Una delle copie è accessibile anche dopo un guasto. Le operazioni di lettura possono essere eseguite in parallelo se non si verificano errori.                                                                                                |
| Livello 2 | Codici di correzione degli errori<br>      | Le informazioni di ridondanza sono memorizzate per rilevare e correggere gli errori. Ogni bit di dati o di informazione ridondante è memorizzato su un disco differente e letto o scritto in parallelo. Garantisce tassi di trasferimento elevati.                                                                                                                                |
| Livello 3 | Bit-interleaved parity<br>                 | Analogo al livello 2, fatta eccezione per il fatto che utilizza un singolo disco di parità per la correzione degli errori. Un errore che si verifica durante la lettura dei dati da un disco viene rilevato dal controller. Il bit di parità viene utilizzato per ripristinare i dati persi.                                                                                      |
| Livello 4 | Block-interleaved parity<br>               | Scrive un <i>block</i> di dati, ovvero byte di dati consecutivi, in una strip e calcola una singola strip di parità per le strip di una stripe. Garantisce tassi di trasferimento elevati per operazioni di lettura molto grandi. Le operazioni di lettura piccole hanno bassi tassi di trasferimento; tuttavia, molte di queste operazioni possono essere eseguite in parallelo. |
| Livello 5 | Block-interleaved distributed parity<br> | Analogo al livello 4, fatta eccezione per il fatto che le informazioni sono distribuite su tutti i dischi. Consente di evitare che il disco di parità diventi un collo di bottiglia per l'I/O come nel livello 4. Inoltre garantisce migliori prestazioni in lettura rispetto al livello 4.                                                                                       |
| Livello 6 | P + Q redundancy<br>                     | Analogo al livello 5, fatta eccezione per il fatto che utilizza due schemi di parità distribuita indipendenti. Supporta il ripristino dal guasto di due dischi.                                                                                                                                                                                                                   |

Nota: D e P indicano dischi che contengono, rispettivamente, solo dati e solo informazioni di parità.  indica una strip.  indica i bit di un byte memorizzati su dischi differenti e i loro bit di parità.  indica una strip contenente solo informazioni di parità.

**Tabella 14.3** Livelli RAID.

#### **RAID Livello 0**

Il livello 0 adotta esclusivamente il disk striping; non è propriamente una configurazione RAID poiché non usa nessuna memorizzazione ridondante dei dati. Garantisce tassi di trasferimento elevati, soprattutto se ogni disco è collegato a un controller separato. Tuttavia, ha una scarsa affidabilità. I dati diventano inaccessibili anche nel caso di un singolo disco spento. Inoltre, la mancanza di ridondanza causa la perdita dei dati nel caso di malfunzionamento di un disco, per cui l'affidabilità deve essere ottenuta con altre tecniche oltre al RAID.

#### **RAID Livello 1**

La configurazione RAID di Livello 1 scrive le stesse informazioni su due dischi; questa tecnica è chiamata *disk mirroring*. Quando un processo scrive o aggiorna un record in un file, una copia del record è scritta su ogni disco. In questo modo, il RAID 1 genera il 100 per cento di overhead; tuttavia, una copia di un record è accessibile anche se si verifica un singolo malfunzionamento. Durante una lettura, il RAID legge semplicemente la copia

che può essere letta prima. Possono essere ottenuti alti tassi di trasferimento durante le operazioni di lettura poiché entrambi i dischi possono funzionare in parallelo quando non si verificano errori.

Le configurazioni ibride che usano le caratteristiche dei RAID di livello 0 e 1 sono spesso usate in pratica per ottenere sia elevati tassi di trasferimento, come nel RAID 0, sia elevata affidabilità, come nel RAID 1. Il RAID 0 + 1 adotta il disk striping, come nel RAID 0, e replica le stripe, come nel RAID 1. Il RAID 1 + 0 prima replica ogni disco e poi esegue lo striping. Queste configurazioni forniscono differenti tipi di fault tolerance: nel RAID 0 + 1, un singolo errore in una copia di una stripe rende l'intera copia inaccessibile, per cui gli errori in entrambe le copie di una stripe renderebbero la stripe inaccessibile. Nel RAID 1 + 0, un errore su un disco sarebbe risolto accedendo al suo disco mirror. Una stripe è inaccessibile solo se un disco e il suo mirror risultano inaccessibili.

### **RAID Livello 2**

Questa configurazione RAID utilizza *bit striping*, ovvero memorizza ogni bit di dati o di informazione ridondante su un disco differente. Quando i dati devono essere scritti, l'*i*-esima strip di dati contiene l'*i*-esimo bit di ogni byte e una strip di parità contiene uno dei bit di parità calcolati a partire dai corrispondenti bit in tutte le strip della stripe. Un codice di correzione dell'errore è utilizzato per calcolare e memorizzare le informazioni ridondanti per ogni byte (Paragrafo 14.3). In questo modo, sono utilizzati 8 dischi per memorizzare i bit di un byte e pochi altri dischi sono utilizzati per memorizzare le informazioni ridondanti. Per esempio, il codice di Hamming (12, 8), sufficiente a ripristinare un singolo errore, richiederebbe 4 bit ridondanti. La configurazione RAID 2, che adotta questo codice, consisterebbe di 8 dischi dati e 4 dischi contenenti informazioni ridondanti, ognuno contenente 1 bit di dati o di informazione di parità. Questa configurazione RAID può leggere o scrivere 8 volte più velocemente rispetto a un singolo disco. Tuttavia, è costosa poiché molti dischi sono necessari per memorizzare le informazioni ridondanti, dunque non è pratica.

### **RAID Livello 3**

Il livello 3 adotta il disk striping con uno shema di *parità con bit alternato*; ovvero adotta il *bit interleaving* - scrive i bit di un byte su dischi differenti - e utilizza un singolo bit di parità per ogni byte. Le strip di una stripe di dati sono memorizzati su un disco di parità. In questo modo, il RAID di livello 3 usa una quantità notevolmente ridotta di informazione ridondante rispetto al RAID di livello 2. Un'operazione di lettura viene eseguita come segue: il controller del disco controlla se si è verificato un errore in una strip. In caso affermativo, ignora l'intera strip e ripristina i dati utilizzando la strip di parità - il valore di un bit di dati è la differenza modulo 2 tra il bit di parità e la somma modulo 2 dei corrispondenti bit delle altre strip della stripe. Tutti i dischi dati sono utilizzati in un'operazione di I/O. Questa caratteristica fornisce elevati tassi di trasferimento. Tuttavia, solo un'operazione di I/O può essere in esecuzione in ogni momento. Un altro svantaggio del RAID di livello 3 è che il calcolo della parità può richiedere molta potenza di CPU. Dunque il calcolo della parità è affidato allo stesso RAID.

### **RAID Livello 4**

Il livello 4 è analogo al livello 3 fatta eccezione per il fatto che adotta la *block-interleaved parity*. Ogni strip contiene un *blocco* di dati, ovvero un certo numero di byte consecutivi di dati. Se un'operazione di I/O coinvolge una grande quantità di dati, utilizzerà tutti i dischi come nel RAID 3, dunque il RAID di livello 4 può fornire elevati tassi di trasferimento per grandi operazioni di I/O. Un'operazione di lettura senza errori i cui dati sono contenuti in un blocco coinvolgerà un singolo disco dati, per cui brevi operazioni di I/O presentano bassi tassi di trasferimento; tuttavia, più operazioni di I/O di questo tipo possono essere eseguite in parallelo.

Un'operazione di scrittura necessita del calcolo delle informazioni di parità sulla base dei dati memorizzati in tutte le strip di una stripe. Questo procedimento consiste prima nel leggere i dati contenuti in tutte le strip di una stripe, sostituire i dati in alcune strip con i nuovi dati da scrivere, calcolare le nuove informazioni di parità e scrivere i nuovi dati e le informazioni di parità su tutti i dischi. Tuttavia, questa procedura limita il parallelismo poiché tutti i dischi sono coinvolti nell'operazione di scrittura anche quando i nuovi dati devono essere scritti in un singolo blocco  $block_i$  della  $stripe_i$ . Dunque, le

informazioni di parità sono calcolate da un metodo più semplice che utilizza l'OR esclusivo di tre elementi - la vecchia informazione nel blocco di parità, i vecchi dati nel blocco  $block_i$ , e i nuovi dati da scrivere nel blocco  $block_i$ . In questo modo, solo il disco o i dischi che contengono il blocco o i blocchi da scrivere e il blocco di parità sono coinvolti nell'operazione di scrittura, per cui diverse operazioni di scrittura possono essere eseguite su altri dischi in parallelo con l'operazione di scrittura.

#### **RAID Livello 5**

Il livello 5 utilizza la parità a livello di blocco come nel livello 4, ma distribuisce le informazioni di parità su tutti i dischi del RAID. Questa tecnica consente di eseguire in parallelo brevi operazioni di scrittura che coinvolgono un singolo blocco se le informazioni di parità sono posizionate su dischi differenti. Brevi operazioni di lettura senza errori possono essere eseguite in parallelo come nel RAID di livello 4. Dunque questa configurazione è particolarmente adatta per brevi operazioni di I/O eseguite con elevata frequenza. Le operazioni più grandi non possono essere eseguite in parallelo; tuttavia, la configurazione fornisce elevati tassi di trasferimento per queste operazioni. Inoltre fornisce picchi di throughput più elevati per le operazioni di lettura rispetto al livello 4 poiché uno o più dischi possono prendere parte alle operazioni di lettura.

#### **RAID Livello 6**

Questa organizzazione utilizza due schemi di parità distribuiti indipendenti. Questi schemi supportano il ripristino dal guasto di due dischi. Il picco di throughput è leggermente più alto rispetto al livello 5 a causa dell'esistenza di un disco in più.

### **14.3.6 Dischi ottici**

I dati sono memorizzati su un disco ottico modificando la riflettività del disco che viene letto da un laser e una testina fotosensibile che cattura i cambiamenti di riflettività della superficie sotto la testina. Un compact disc (CD) è un disco ottico. La scrittura sul disco avviene memorizzando un 1, modificando la riflettività rispetto al bit nella posizione precedente, e memorizzando uno 0, mantenendo la stessa riflettività del bit precedente.

La memorizzazione su un CD può essere eseguita con vari mezzi. I CD prodotti in massa contenenti musica sono prodotti con mezzi meccanici. Questi sono chiamati *CD stampati*. La memorizzazione può anche essere eseguita utilizzando un raggio laser. Un CD masterizzato con laser contiene tre strati: uno strato di policarbonato, un polimero colorato e uno strato metallico riflettente. Quando un forte raggio laser viene diretto in un punto sul CD, riscalda il polimero e crea un segno permanente sul disco chiamato *pit*, che ha una bassa riflettività. Questo è il motivo per cui il procedimento di masterizzazione è chiamato "burning". I dati sono memorizzati in una scanalatura a forma di spirale sul CD che si estende dal diametro interno del CD al diametro esterno. Un CD contiene 22.188 rivoluzioni della spirale, distanziate di circa 1,6 micron. Ogni rivoluzione è chiamata *traccia*. Il controllo della velocità e l'informazione relativa al tempo assoluto sono preregistrate sul CD.

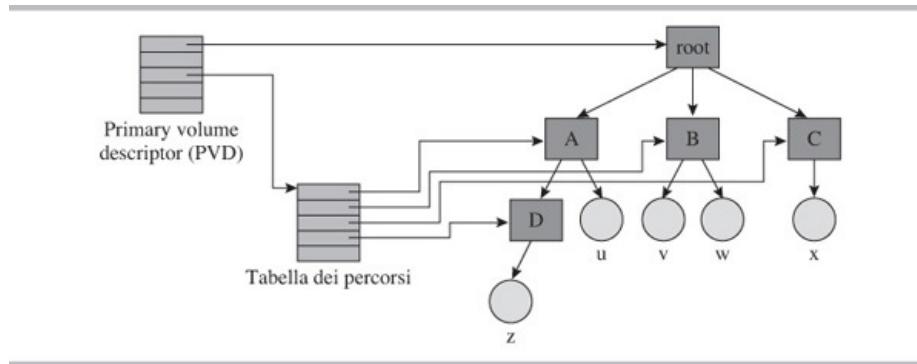
Un CD contiene diverse regioni riservate per essere utilizzate dai masterizzatori. L'area di calibrazione della potenza è utilizzata per calibrare la potenza del laser di scrittura. La program memory area contiene le informazioni della traccia relativa a ogni sessione. La program memory area è seguita dalle aree lead-in, program e lead-out di ogni sessione. Un'area lead-in memorizza il contenuto di una sessione. Indica il numero di tracce, i punti di inizio e fine della traccia e la lunghezza della sessione. L'area program contiene le tracce dei dati della sessione. L'area lead-out indica la fine di una sessione.

Dal punto di vista del sistema operativo due sono le caratteristiche importanti di un CD - la memorizzazione dei dati e la creazione di un file system. I dati sono memorizzati come settori su una traccia. Un CD-ROM da utilizzare su un computer contiene settori di 2 KB ed ha una capienza di circa 650 MB. Un DVD (digital versatile disk), invece, ha una capienza di circa 5 GB. I dati sono memorizzati su ogni tipo di disco utilizzando il metodo di codifica chiamato CIRC (cross-interleaved Reed-Solomon code). CIRC codifica una unità di dati di 24 byte, ovvero 192 bit, per produrre una unità codificata con dimensione di 588 bit. Oltre ai dati, questa unità contiene le informazioni relative ai tempi relativi e assoluti, la disposizione delle tracce e gli indici; la sincronizzazione dei dati e i dati per il rilevamento e la correzione degli errori. Per rendere la correzione degli errori affidabile, i dati vengono mescolati durante la codifica. In questo modo, se alcuni byte di dati memorizzati consecutivamente vengono persi, un grande numero di byte può perdere

solo un bit. Questi dati possono essere ripristinati utilizzando le informazioni per la correzione degli errori.

Lo Standard ISO 9660 definisce un formato logico comune per i file e le directory su un CD. Definisce i requisiti di base per lo scambio di dati e inoltre fornisce le estensioni opzionali per gli ambienti Windows, Unix e Macintosh. L'estensione Rockridge ammette i nomi dei file lunghi specifici di Unix, directory multilivello, privilegi di accesso e tipi di file. L'universal disk format (UDF) è anche progettato per un file system logico comune, ovvero cross-platform. UDF può coesistere con ISO 9660 e molti drive CD possono scrivere informazioni in entrambi i formati.

La [Figura 14.10](#) mostra come viene implementato un file system su un CD. Il descrittore di volume primario (PVD) è memorizzato nel settore logico 16. Indica la posizione della directory di root e la posizione di una tabella dei percorsi. Ogni elemento della tabella dei percorsi contiene informazioni sulla posizione di una directory. Il suo utilizzo per localizzare la directory richiesta evita le ricerche nelle directory intermedie di un percorso; in un sistema Unix, per esempio, evita le ricerche nelle directory `root` e `A` per il percorso `~A/D/z`.



**Figura 14.10** Descrittore di volume primario di un CD.

## 14.4 I/O a livello di dispositivo

Quattro funzioni sono coinvolte nell'implementazione dell'I/O a livello di dispositivo – inizializzazione di un'operazione di I/O, esecuzione delle operazioni di lettura/scrittura, controllo dello stato del dispositivo e gestione degli interrupt generati dal dispositivo. Le prime tre funzioni sono eseguite mediante le *istruzioni di I/O* e i *comandi di I/O* descritti nel Paragrafo 14.2. La [Tabella 14.4](#) descrive le caratteristiche del computer che supportano queste funzioni. Assumiamo che le operazioni di I/O siano eseguite in modalità DMA (Paragrafo 2.2.4). Nel Paragrafo 14.4.1, discuteremo i dettagli dell'I/O a livello di dispositivo e nel Paragrafo 14.5, spiegheremo le funzionalità fornite dal IOCS fisico per semplificare l'I/O a livello di dispositivo.

### 14.4.1 Programmazione dell'I/O

Si usa il termine *programmazione dell'I/O* per descrivere tutte le azioni coinvolte nell'esecuzione di un'operazione di I/O. Per comprendere i due aspetti chiave della programmazione dell'I/O – precisamente, l'inizializzazione dell'I/O e il completamento dell'I/O – consideriamo il programma di [Figura 14.11](#), che rappresenta una versione in linguaggio assembly del seguente programma in un linguaggio di alto livello:

---

|              |                                                                                                                                               |                                                                                                                                                                                |
|--------------|-----------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RETRY:       | SET IO_FLAG, '1'<br>IO_init (cu, d), COMMANDS<br>BC cc <sub>1</sub> , IN_PROGRESS<br>BC cc <sub>2</sub> , RETRY<br>BC cc <sub>3</sub> , ERROR | Per indicare che l'I/O è in esecuzione<br>legge a, b<br>Salta se l'avvio ha successo<br>Cicla se il dispositivo è occupato<br>Errore, informare l'amministratore<br>di sistema |
| IN_PROGRESS: | COMP IO_FLAG, '1'<br>BC EQ, IN_PROGRESS<br>{Esegue result = a+b;}                                                                             | Verifica se l'I/O è ancora in esecuzione<br>Cicla se l'I/O è in esecuzione                                                                                                     |
| COMMANDS:    | {Comandi di I/O}<br>...                                                                                                                       |                                                                                                                                                                                |
| IO_INTRPT:   | SET IO_FLAG, '0'<br>...                                                                                                                       | Elaborazione dell'interrupt:<br>I/O completato                                                                                                                                 |

---

**Figura 14.11** Programmazione dell'I/O.

| Funzione                                 | Descrizione della caratteristica del computer che la supporta                                                                                                                                                                                                  |
|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Inizializzazione di un'operazione di I/O | L'istruzione di I/O <i>I/O-init(cu, d)</i> , <i>command address</i> inizializza un'operazione di I/O (vedi Esempio 14.1). L'istruzione <i>I/O-init</i> imposta un codice di condizione per indicare se l'operazione di I/O è stata inizializzata con successo. |
| Esecuzione della lettura/scrittura       | I comandi di I/O specifici per un dispositivo implementano le operazioni come il posizionamento delle testine di lettura/scrittura su un record e la lettura di un record.                                                                                     |
| Controllo dello stato del dispositivo    | L'istruzione di I/O <i>I/O-status (cu, d)</i> ottiene le informazioni di stato per un dispositivo di I/O. Le informazioni indicano se il dispositivo è occupato, libero o in uno stato di errore e la causa dell'errore, se esiste.                            |
| Gestione degli interrupt                 | L'hardware degli interrupt implementa l'azione di interrupt descritta nel Paragrafo 2.2. La CPU viene commutata all'IOCS fisico quando si verifica un interrupt relativo al completamento dell'I/O.                                                            |

**Tabella 14.4** Caratteristiche del computer che supportano le funzioni nell'I/O a livello di dispositivo.

```
read a, b;  
...  
result := a + b;
```

Il programma utilizza le componenti hardware del computer, ovvero un computer che non ha nessun livello software tra il programma e le componenti hardware. Il programma utilizza il flag *IO\_FLAG* per indicare se l'operazione di I/O è in esecuzione. Imposta *IO\_FLAG* a 1, inizializza un'operazione di I/O e cicla finché non viene completata l'operazione di I/O; a questo punto inizia l'elaborazione dei dati.

### Avvio dell'I/O

Quando è eseguita l'istruzione *I/O-init* di [Figura 14.11](#), la CPU invia l'indirizzo del dispositivo al controller del DMA. Il controller del DMA verifica se il dispositivo è disponibile per l'operazione di I/O e informa la CPU; la CPU imposta il codice di condizione appropriato nel campo *codice di condizione* (anche detto campo *flag*) del PSW.

Se il dispositivo è disponibile, il DMA avvia anche l'operazione di I/O accedendo e decodificando il primo comando di I/O. L'istruzione *I/O-init* è completata. L'operazione di I/O, se avviata, procederà in parallelo con l'esecuzione delle istruzioni della CPU.

Nelle successive istruzioni, il programma esamina il codice di condizione impostato dall'istruzione *io-init* per gestire qualunque situazione eccezionale che potrebbe essersi verificata durante l'esecuzione dell'istruzione *I/O-init*. L'istruzione `BC cc1, IN_PROGRESS` è un'istruzione di salto condizionato. Il codice di condizione *cc<sub>1</sub>* è stato impostato nel caso di successo di avvio dell'operazione. In questo caso l'operazione di I/O è già partita, per cui l'esecuzione del programma viene fatta proseguire con l'istruzione individuata dalla label `IN-PROGRESS`. Il codice di condizione *cc<sub>2</sub>* indica che il dispositivo è occupato, per cui il programma ritenta l'istruzione di I/O finché l'avvio dell'operazione di I/O non va a buon fine. Il codice di condizione *cc<sub>3</sub>* indica che si è verificato un errore, per cui il programma riporta l'errore all'amministratore di sistema. Questi dettagli non sono mostrati nella [Figura 14.11](#).

### **Completamento dell'elaborazione di I/O**

Il programma non può eseguire l'elaborazione `result = a+b`; finché non viene completata l'operazione di I/O. Tuttavia l'esecuzione del programma non può essere sospesa poiché è in esecuzione sul hardware. Il programma risolve questo problema utilizzando il flag `IO_flag` per indicare se l'operazione di I/O è stata completata. Per cominciare, imposta il valore di `IO_FLAG` a 1 per indicare che l'I/O è in esecuzione. Dopo aver avviato l'operazione di I/O, entra in un loop a `IN-PROGRESS` in cui controlla ripetutamente questo flag. Questo è una busy wait - vedi Paragrafo 6.5.1.

Quando si verifica un interrupt di I/O che indica la fine dell'operazione di I/O, il controllo viene trasferito all'istruzione con la label `IO-INTRPT` dall'azione di interrupt (Paragrafo 2.2). Questo è l'inizio della routine di gestione dell'interrupt di I/O, che imposta `IO_FLAG` a 0 e ritorna. Questa azione termina la busy wait a `IN_PROGRESS`.

## **14.5 IOCS fisico**

Lo scopo del IOCS fisico è di semplificare il codice dei processi utente nascondendo la complessità delle operazioni di I/O e di assicurare elevate prestazioni del sistema. Questo è ottenuto mediante le seguenti tre istruzioni.

- *Gestione dell'I/O a livello di dispositivo*: il IOCS fisico fornisce un'interfaccia per l'I/O a livello di dispositivo che elimina la complessità della programmazione dell'I/O discussa nel Paragrafo 14.4.1.
- *Sincronizzazione di un processo con il completamento di un'operazione di I/O*: questa sincronizzazione evita l'attesa attiva successiva all'avvio dell'I/O in [Figura 14.11](#) e rilascia la CPU in modo che possa essere utilizzata da altri processi.
- *Scheduling dell'I/O*: il IOCS fisico schedula le operazioni di I/O da eseguire su un dispositivo in un ordine appropriato per consentire al dispositivo di avere un throughput elevato.

### **Gestione dell'I/O a livello dispositivo**

Durante la richiesta di avvio di un'operazione di I/O, un processo deve specificare solo l'indirizzo del dispositivo e i dettagli dell'operazione di I/O. Il IOCS fisico avvia immediatamente un'operazione di I/O se il dispositivo è disponibile; altrimenti, annota la richiesta di avvio dell'I/O e l'avvia successivamente. In ogni caso, il controllo viene passato al processo che aveva eseguito la richiesta di I/O. Quando si verifica un interrupt, il IOCS fisico annota quale operazione di I/O è stata completata e avvia un'altra operazione sul dispositivo di I/O, se presente.

### **Sincronizzazione di un processo con il completamento di un'operazione di I/O**

Il IOCS fisico fornisce una funzionalità di "attesa del completamento dell'I/O" per bloccare un processo finché un'operazione di I/O non viene completata. I suoi parametri sono l'indirizzo del dispositivo di I/O e i dettagli dell'operazione di I/O. Quando un processo invoca questa funzionalità, il IOCS fisico controlla se l'operazione di I/O è già stata completata. In caso negativo, richiede al kernel di bloccare il processo. Questa azione evita l'attesa attiva di [Figura 14.11](#). Lo stato del processo viene modificato a `ready` al termine dell'operazione di I/O.

### Scheduling dell'I/O

Il throughput di un dispositivo di I/O può essere calcolato come il numero di byte di dati trasferiti per unità di tempo, o come il numero di operazioni di I/O effettuate per unità di tempo. Il throughput può essere ottimizzato minimizzando i tempi di accesso durante le operazioni di I/O. Nei dischi questo risultato può essere ottenuto, ovvero riducendo la latenza rotazionale e il movimento meccanico delle testine del disco eseguendo le operazioni di I/O in un ordine opportuno. Questa funzione è chiamata *scheduling dell'I/O*. Viene eseguito automaticamente dal IOCS fisico e non viene richiamato esplicitamente da un processo.

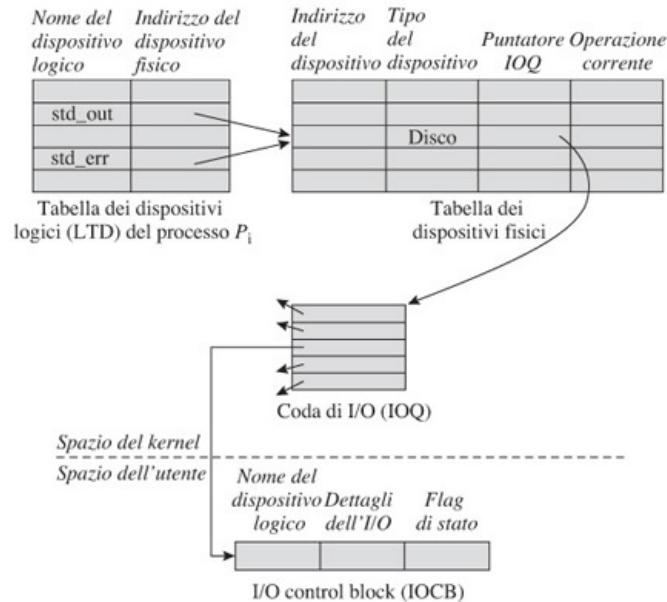
### 14.5.1 Dispositivi logici

Un *dispositivo logico* è un'astrazione utilizzata per una varietà di scopi. Nel caso più semplice, un dispositivo logico è semplicemente il nome di un dispositivo di I/O fisico. L'uso di un dispositivo logico nel codice di un processo risolve una difficoltà pratica - l'indirizzo di un dispositivo logico che un processo utilizzerà non è conosciuto al momento della stesura del codice. Durante la creazione di un processo che utilizza un dispositivo logico, il kernel assegna un dispositivo fisico al dispositivo logico. Quando il processo esegue un'operazione sul dispositivo logico, IOCS fisico implementa l'operazione del dispositivo fisico assegnato al dispositivo logico.

Un dispositivo fisico può anche essere un dispositivo virtuale come descritto nel Paragrafo 1.3.2. In questo caso, il kernel deve mappare il dispositivo logico in una parte del dispositivo fisico. In questo modo molti dischi logici possono essere mappati in un disco fisico; le operazioni di I/O dirette ai dischi logici sarebbero tutte eseguite sullo stesso disco fisico.

### 14.5.2 Strutture dati del IOCS fisico

Il IOCS fisico utilizza le seguenti strutture dati ([Figura 14.12](#)).



**Figura 14.12** Strutture dati del IOCS fisico.

- tabella dei dispositivi fisici (physical device table – PDT).
- tabella dei dispositivi logici (logical device table - LDT).
- I/O control block (IOCB).
- coda di I/O (IOQ).

La *physical device table* (PDT) è una struttura globale. Ogni elemento della tabella contiene le informazioni relative a un dispositivo di I/O. Il campo *puntatore IOQ* di un

elemento punta alla coda delle operazioni di I/O che devono essere eseguite sul dispositivo. Ogni elemento nella coda è un puntatore a un I/O control block (IOCB), che contiene le informazioni relative a un'operazione di I/O. Il campo *operazione corrente* punta all'I/O control block che contiene le informazioni relative all'operazione di I/O avviata sul dispositivo. Queste informazioni sono utili per elaborare il completamento dell'operazione di I/O.

La *logical device table* (LDT) è una struttura dati locale a ogni processo. C'è una copia della LDT per ogni processo nel sistema; questa copia è accessibile dal process control block (PCB) del processo. La LDT contiene un elemento per ogni dispositivo logico utilizzato dal processo. Il campo *indirizzo del dispositivo fisico* dell'elemento contiene l'informazione relativa all'assegnazione corrente, se esiste, per il dispositivo logico. Si noti che molti dispositivi logici, probabilmente appartenenti a diversi processi utente, possono essere assegnati allo stesso dispositivo fisico come, per esempio, un disco.

Un *I/O control block* (IOCB) contiene tutte le informazioni relative a un'operazione di I/O. I campi importanti di un IOCB sono il *nome del dispositivo logico*, i *dettagli dell'I/O* e il *flag di stato*. Il campo dettagli dell'I/O contiene l'indirizzo del primo comando di I/O. Il flag di stato indica se un'operazione di I/O è in "esecuzione" o è "completata"; è l'equivalente di `IO-FLAG` in [Figura 14.11](#).

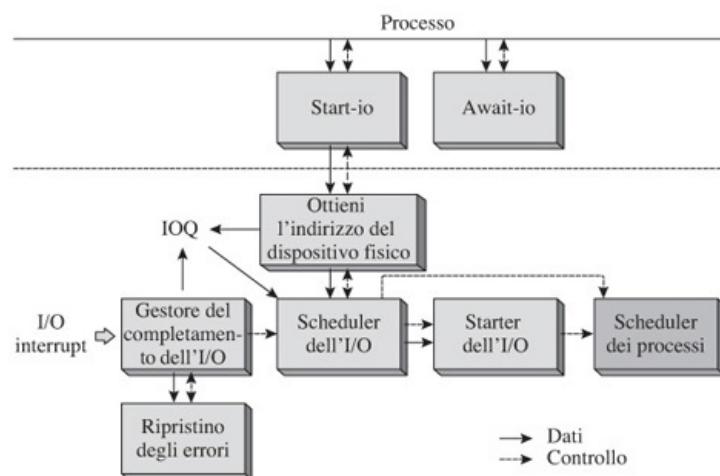
La *coda di I/O* (IOQ) è una lista di tutte le operazioni di I/O in attesa su un dispositivo fisico. Ogni elemento della IOQ contiene un puntatore a un I/O control block. Le informazioni contenute nella IOQ sono utilizzate per lo scheduling dell'I/O.

La PDT viene creata al momento del boot del sistema raccogliendo i dettagli di tutti i dispositivi connessi al sistema. La dimensione della LDT è specificata al momento del boot. Una LDT viene creata al momento della creazione di un processo. Un I/O control block è allocato nel momento in cui deve essere eseguita un'operazione di I/O. La IOQ viene mostrata come un array di puntatori in [Figura 14.12](#). Tuttavia, è più pratico organizzarla come lista linkata di IOCB.

Le strutture dati PDT, LDT e IOQ si trovano nel kernel, mentre un processo crea un IOCB nel suo spazio di indirizzamento, inizializza i suoi campi e lo utilizza come parametro in una chiamata a un modulo del IOCS fisico. La presenza del IOCB nello spazio di indirizzamento del processo consente al processo di controllare lo stato di un'operazione di I/O senza dover richiamare il kernel.

### 14.5.3 Organizzazione del IOCS fisico

La [Figura 14.13](#) mostra l'organizzazione del IOCS fisico. I moduli al di sopra della linea tratteggiata vengono eseguiti con la CPU in modalità utente, mentre quelli al di sotto di questa linea vengono eseguiti con la CPU in modalità kernel. Il IOCS fisico è attivato in uno dei seguenti modi:



**Figura 14.13** Organizzazione del IOCS fisico.

- mediante le chiamate `start-io` o `await-io` ai moduli di libreria del IOCS fisico da parte di

un processo utilizzando come parametro un I/O control block;

- mediante l'occorrenza di un interrupt di completamento di I/O.

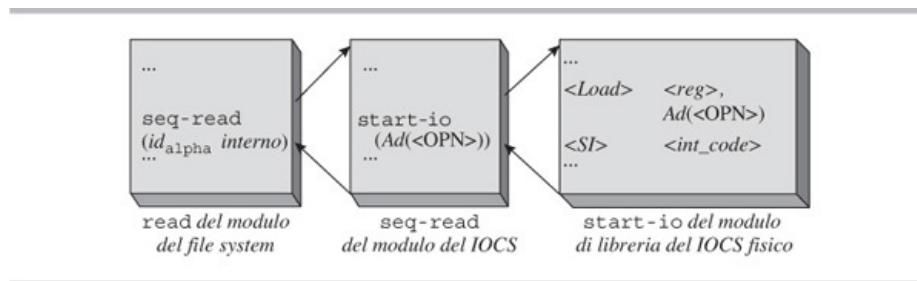
Quando un processo richiama `start-io`, questa richiama lo starter dell'I/O mediante una chiamata di sistema. Lo starter dell'I/O ottiene l'indirizzo del dispositivo fisico su cui deve essere eseguita l'operazione di I/O, inserisce l'operazione di I/O nella IOQ del dispositivo fisico e passa il controllo allo scheduler dell'I/O. Lo scheduler richiama lo starter dell'I/O per avviare immediatamente l'operazione di I/O se non sono presenti altre operazioni nella IOQ del dispositivo. A questo punto il controllo è passato allo scheduler dei processi che a sua volta lo restituisce al processo che aveva richiesto l'operazione di I/O.

Quando viene richiamato il modulo `await-io` del IOCS fisico, questo determina lo stato dell'operazione di I/O esaminando il flag di stato dell'I/O control block. Se l'operazione di I/O è stata completata, il controllo viene restituito immediatamente al processo; altrimenti, il modulo `await-io` effettua una chiamata di sistema per bloccare il processo. Nel momento in cui un dispositivo genera un interrupt di completamento di un I/O, nel caso in cui si sia verificato un errore viene richiamata una routine di ripristino dagli errori; altrimenti, il flag di stato dell'I/O control block che descrive l'operazione corrente sul dispositivo è impostato a "completed", la configurazione di ECB-PCB dell'Esempio 5.4 è utilizzata per attivare un processo (se presente) in attesa del completamento dell'operazione di I/O e infine viene richiamato lo scheduler dell'I/O. Questo seleziona una delle operazioni di I/O pendenti per il dispositivo e la passa allo starter dell'I/O. Lo starter dell'I/O avvia l'operazione di I/O e passa il controllo allo scheduler dei processi.

#### 14.5.4 Implementazione del IOCS fisico

Si ricordi dal Paragrafo 13.1 che il compilatore rimpiazza le istruzioni di elaborazione dei file in un programma con chiamate alle operazioni del file system `open`, `read` e `close`. Come visto nel Paragrafo 13.8, l'operazione sul file system `read` effettua una chiamata al modulo di libreria del IOCS `seq-read`. `seq-read` contiene il codice che contribuisce all'elaborazione efficiente di un file (maggiori dettagli più avanti in questo capitolo). Questo codice effettua una chiamata al modulo di libreria del IOCS fisico `start-io` per eseguire l'I/O a livello di dispositivo. Il linker linka tutti questi moduli del file system, il IOCS e il IOCS fisico con il programma compilato.

Un processo che rappresenta l'esecuzione del programma linkato effettua una chiamata all'operazione `open` del file system per aprire un file `alpha`. `open` crea un file control block (FCB) per `alpha`, ovvero,  $fcb_{alpha}$ , nella active file table (AFT) e restituisce l' $id_{alpha}$  interno, che rappresenta l'offset del FCB nella AFT (Paragrafo 13.9.1). Le seguenti azioni vengono eseguite per un record di `alpha` (Figura 14.14).



**Figura 14.14** Invocazione del modulo di libreria del IOCS fisico `start-io` all'interno di un processo.

1. Il processo richiama il modulo `read` del file system, che richiama il modulo IOCS `seq-read` con  $id_{alpha}$  interno come parametro.
2. Quando `seq-read` decide di leggere un record di `alpha`, usa  $id_{alpha}$  interno per accedere a  $fcb_{alpha}$ , ottiene l'indirizzo di  $fmt_{alpha}$  e trova l'indirizzo del blocco sul disco che contiene il record desiderato. A questo punto crea un I/O control block per l'operazione di I/O e chiama `start-io` con l'indirizzo dell'I/O control block come parametro. L'I/O control block è chiamato *OPN* in Figura 14.14.
3. `start-io` carica l'indirizzo dell'I/O control block in un registro general purpose ed esegue un'istruzione `SI` con un appropriato codice per richiamare il IOCS fisico.

### **Inizio dell'I/O**

Quando viene invocato mediante una chiamata di sistema, il IOCS fisico ottiene l'indirizzo del IOCB dal registro general purpose ed esegue le seguenti azioni.

1. Imposta il campo *status flag* del IOCB a "in progress."
2. Inserisce l'indirizzo dell'I/O control block nella IOQ del dispositivo fisico.
3. Avvia l'operazione di I/O, se il dispositivo di I/O non è occupato.
4. Restituisce il controllo al processo.

Per inserire l'indirizzo dell'I/O control block nella IOQ corretta, il IOCS fisico estrae l'id del dispositivo logico dal IOCB e accede alla tabella dei dispositivi logici (LDT) del processo per ottenere l'indirizzo del dispositivo fisico assegnato al dispositivo logico. Successivamente ottiene l'indirizzo della IOQ del dispositivo fisico dall'elemento della tabella dei dispositivi fisici (PDT) e aggiunge l'indirizzo del IOCB in coda alla IOQ. L'operazione di I/O può essere iniziata immediatamente se non ci sono altri elementi nella IOQ. Se esistono altri elementi, presumibilmente una delle operazioni precedenti è in esecuzione, per cui l'operazione di I/O viene posta in attesa.

L'inizio dell'I/O è eseguito come descritto nel Paragrafo 14.4.1. Il campo *flag di stato* dell'I/O control block è usato in maniera analoga all'IO\_FLAG in [Figura 14.11](#). L'indirizzo dell'I/O control block è memorizzato nel campo *operazione corrente* dell'elemento relativo al dispositivo nella physical device table.

### **Gestione del completamento dell'I/O**

Il gestore del completamento dell'I/O è richiamato implicitamente all'occorrenza di un interrupt di completamento dell'I/O. La componente hardware per la gestione dell'interrupt fornisce l'indirizzo del dispositivo fisico che ha generato l'interrupt di I/O. Il gestore del completamento dell'I/O interroga il dispositivo per ottenere un codice di stato dell'I/O che descrive la causa dell'interrupt. A questo punto esegue le seguenti azioni: se l'operazione di I/O non era andata a buon fine, consulta il campo *tipo di dispositivo* dell'elemento della PDT e richiama un'appropriata routine di ripristino degli errori passandogli come parametro l'indirizzo dell'I/O control block. Altrimenti, imposta il *flag di stato* dell'I/O control block a "completed" e rimuove l'indirizzo dell'I/O control block dalla IOQ del dispositivo. Se ci sono operazioni pendenti per il dispositivo, ne avvia una mediante lo scheduler di I/O (Paragrafo 14.7) e inserisce l'indirizzo del suo I/O control block nel campo *operazione corrente* dell'elemento della PDT. Se il processo che ha segnalato il completamento dell'operazione di I/O è bloccato in attesa del completamento dell'operazione di I/O, cambia lo stato del processo a *ready*. L'organizzazione utilizzata a questo scopo è descritta di seguito.

### **Attesa del completamento di un'operazione di I/O**

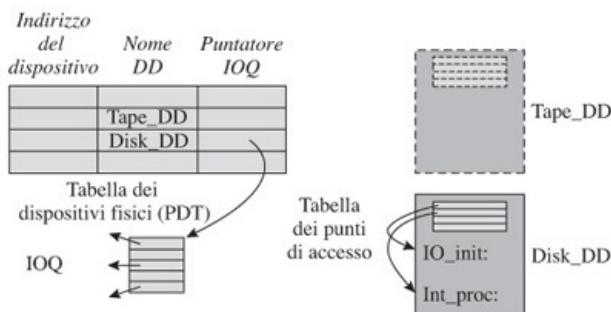
Un processo invoca questa funzione mediane la chiamata di libreria del IOCS fisico *await-io (<IOCB\_address>)* dove l'I/O control block descrive l'operazione di cui si attende il completamento. Il IOCS fisico controlla semplicemente il flag di stato nell'I/O control block e restituisce il controllo al processo se il valore del flag è "completed". In caso contrario, la routine di libreria del IOCS fisico esegue una chiamata di sistema per bloccarsi sull'evento che segnala il completamento dell'I/O. Il kernel crea un event control block (ECB) per l'evento relativo al completamento dell'I/O e lo inserisce nella lista degli event control block. Questo event control block contiene l'id del processo in attesa del completamento dell'operazione di I/O. Quando si verifica l'evento relativo al completamento dell'I/O, il gestore del completamento dell'I/O localizza il corrispondente event control block, estrae l'id del processo e modifica il suo stato. Questa organizzazione garantisce che il processo sia attivato al completamento dell'operazione di I/O e che ritorni dalla chiamata alla routine di libreria del IOCS fisico. (Esempio 5.4 per un spiegazione di questa configurazione.)

## **14.6 Driver di dispositivo**

Nella configurazione descritta nei paragrafi precedenti, il IOCS fisico gestisce l'inizio dell'I/O, il completamento dell'I/O e la gestione degli errori per tutte le classi dei dispositivi di I/O nel sistema. Di conseguenza, l'aggiunta di una nuova classe di dispositivi di I/O richiede delle modifiche al IOCS fisico, operazione che può risultare

complessa e costosa poiché il IOCS fisico può essere parte del kernel. I sistemi operativi moderni risolvono questo problema attraverso una configurazione differente. Il IOCS fisico fornisce solo il supporto generico per le operazioni di I/O e richiama un *driver di dispositivo* (device driver - DD) specifico per gestire le operazioni a livello di dispositivo di una specifica classe di dispositivi. Per questo motivo i device driver non fanno parte del IOCS fisico. Questa configurazione consente di aggiungere al sistema di dispositivi di I/O senza dover modificare il IOCS fisico. I driver di dispositivo sono caricati dalla procedura di boot del sistema in base alla classe dei dispositivi di I/O connessi al sistema. In alternativa, i device driver possono essere caricati quando necessario durante il funzionamento del SO. Questa caratteristica è particolarmente utile per la funzione *plug-and-play*.

La [Figura 14.15](#) illustra come i driver di dispositivo sono utilizzati dal IOCS fisico. L'elemento relativo a un dispositivo nella tabella dei dispositivi fisici (PDT) contiene il nome del driver di dispositivo nel campo *nome DD*. Il *Disk\_DD*, il device driver del disco di sistema, è stato caricato al momento del boot. Il *Tape\_DD* può essere caricato su richiesta, per cui viene mostrato come un rettangolo tratteggiato. Un device driver contiene le funzionalità dei quattro moduli del IOCS fisico mostrati in [Figura 14.13](#), precisamente, lo scheduler dell'I/O, lo starter dell'I/O, il gestore del completamento dell'I/O e il gestore degli errori. Una tabella posizionata all'inizio del codice contiene gli indirizzi di queste funzionalità.



**Figura 14.15** Uso dei device driver.

Quando il IOCS fisico è richiamato per iniziare un'operazione di I/O, localizza l'elemento relativo al dispositivo nella PDT ed esegue la generica operazione di inserimento dei dettagli dell'operazione di I/O nella IOQ del dispositivo. A questo punto consulta il campo *nome DD* dell'elemento della PDT, ottiene l'identità del driver di dispositivo e lo carica in memoria, se non è già stato caricato. Recupera l'indirizzo del punto di ingresso relativo allo starter dell'I/O nel driver e gli passa il controllo. Il driver di dispositivo esegue l'avvio dell'I/O e restituisce il controllo al IOCS fisico, che a sua volta passa il controllo allo scheduler dei processi. Il OCS fisico, richiamato implicitamente nel caso di interrupt di I/O, esegue azioni simili per identificare il punto di entrata della funzione per la gestione degli interrupt nel driver di dispositivo e gli passa il controllo. Al termine della gestione dell'interrupt, il device driver restituisce il controllo al IOCS fisico, che a sua volta lo passa allo scheduler dei processi.

### Ottimizzazione a livello di dispositivo

Una importante ottimizzazione riguarda lo *scheduling del disco* il cui scopo è garantire un buon throughput. Un'altra ottimizzazione riguarda la riduzione del numero di operazioni di ricerca in un disco. Questa ottimizzazione può essere eseguita in molti modi. Un modo semplice consiste nel leggere diversi blocchi vicini al blocco richiesto al momento di un'operazione di lettura; ciò coinvolge il buffering dei dati, particolarmente utile nei file sequenziali. Un altro modo è dato dal modo di operare dei driver di dispositivo per le unità RAID; essi riducono il numero di operazioni di ricerca combinando diverse operazioni di I/O in una singola operazione.

Un device driver può anche supportare un dispositivo di I/O nuovo o non standard. Un buon esempio del primo modo di ottimizzazione è un disco RAM, cioè un disco virtuale mantenuto nella RAM di un computer: un'area in RAM viene riservata per essere utilizzata come un disco. Tutte le operazioni di lettura e scrittura dirette al disco sono di

fatto eseguite in aree della RAM. Le operazioni eseguite nel disco RAM sono estremamente veloci. Tuttavia, i dati memorizzati al suo interno sono persi se il sistema si arresta o se il disco RAM è rimosso. Per questa ragione, i file temporanei dei compilatori e dei processi sono generalmente creati in un disco RAM. I file utilizzati per memorizzare dati per tempi più lunghi sono conservati sui dischi tradizionali.

## 14.7 Scheduling del disco

Il tempo di ricerca di un blocco dipende dalla sua posizione rispetto alla posizione corrente delle testine del disco. Di conseguenza, il tempo totale di ricerca necessario per eseguire un insieme di operazioni di I/O dipende dall'ordine in cui le operazioni sono eseguite. Il throughput di un disco, definito come il numero di operazioni di I/O eseguite in un secondo, dipende dall'ordine in cui le operazioni sono eseguite. Dunque il IOCS fisico e i device driver dei dischi adottano una politica di *scheduling del disco* per eseguire le operazioni di I/O nell'ordine opportuno. Descriveremo le seguenti politiche di schedulazione del disco prima di descrivere lo scheduling del disco nei moderni sistemi operativi.

- *First-come, first-served (FCFS)*: seleziona l'operazione di I/O con tempo di richiesta inferiore.
- *Shortest seek time first (SSTF)*: seleziona l'operazione di I/O con il più breve tempo di ricerca rispetto alla posizione corrente delle testine del disco.
- *SCAN*: questa politica muove le testine del disco da un estremo all'altro del piatto, servendo le operazioni di I/O per i blocchi su ogni traccia/cilindro prima di spostarsi sulla prossima traccia/cilindro. Per questo motivo si chiama *scan*. Quando le testine del disco raggiungono l'altra estremità del piatto, la direzione di movimento viene invertita e le nuove richieste vengono servite nella scansione inversa. Una variante chiamata *look* inverte la direzione delle testine del disco quando non ci sono più richieste di I/O nella direzione corrente; è nota come *algoritmo dell'ascensore*.
- *Circular SCAN o CSCAN*: questa politica esegue la scansione come nello scheduling SCAN. Tuttavia, non esegue mai la scansione inversa; invece, sposta le testine nella posizione di partenza sul piatto e inizia un'altra scansione. La variante *circular look* (che chiameremo *scheduling C-look*) muove le testine solo finché ci sono richieste da eseguire prima di iniziare una nuova scansione.

La politica di scheduling del disco FCFS è facile da implementare ma non garantisce un buon throughput del disco. Per implementare la politica SSTF, il IOCS fisico utilizza un modello del disco per calcolare il tempo di ricerca del blocco coinvolto in un'operazione di I/O a partire dalla posizione corrente delle testine del disco. Tuttavia, la politica SSTF è analoga alla politica di scheduling *shortest request next* (SRN), per cui se da una parte ottiene un buon throughput del disco, dall'altra alcune richieste potrebbero incorrere in starvation. SSTF e le varie politiche di scansione possono essere implementate in maniera efficiente se le IOQ sono ordinate per numero di traccia.

L'Esempio 14.3 descrive il funzionamento di varie politiche di scheduling per un insieme di cinque operazioni di I/O. La politica *look* completa tutte le operazioni di I/O di questo esempio nella minore quantità di tempo. Tuttavia, nessuna di queste politiche in pratica risulta migliore poiché il pattern degli accessi al disco non può essere predetto.

### Esempio 14.3 - Politiche di schedulazione del disco

La [Figura 14.16](#) riassume le prestazioni delle politiche di scheduling del disco FCFS, SSTF, Look e C-Look per cinque operazioni di I/O su un ipotetico disco composto di 200 tracce. Le richieste sono sottomesse in istanti differenti. Si assume che l'operazione di I/O precedente venga completata quando il clock di sistema è a 160 ms. Si assume che il tempo richiesto per spostare le testine del disco dalla traccia<sub>1</sub> alla traccia<sub>2</sub> sia una funzione lineare della differenza tra le rispettive posizioni:

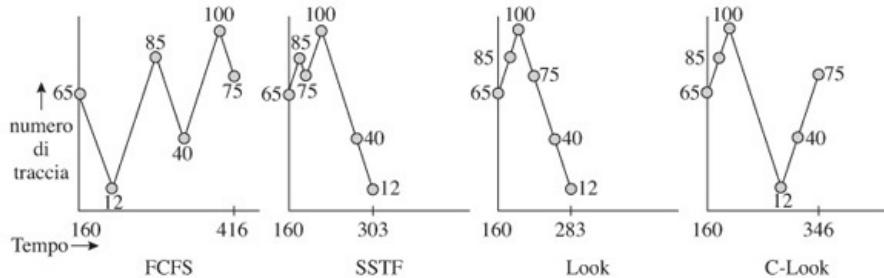
|                                      |                                       |
|--------------------------------------|---------------------------------------|
| $t_{\text{const}}$ e $t_{\text{pt}}$ | = 0 ms e 1 ms, rispettivamente        |
| Posizione corrente della testina     | = Traccia 65                          |
| Direzione dell'ultimo movimento      | = Verso le tracce con numeri maggiori |
| Tempo di clock corrente              | = 160 ms                              |

Operazioni di I/O Richieste:

| Numero di richiesta | 1  | 2  | 3   | 4   | 5   |
|---------------------|----|----|-----|-----|-----|
| Numero di traccia   | 12 | 85 | 40  | 100 | 75  |
| Tempo di arrivo     | 65 | 80 | 110 | 120 | 175 |

Dettagli della schedulazione:

| Politica | Dettagli                | Decisioni di scheduling |            |         |      |     | $\Sigma$ tempo di ricerca |
|----------|-------------------------|-------------------------|------------|---------|------|-----|---------------------------|
|          |                         | 1                       | 2          | 3       | 4    | 5   |                           |
| FCFS     | Tempo di decisione      | 160                     | 213        | 286     | 331  | 391 | 256                       |
|          | Richieste pendenti      | 1, 2, 3, 4              | 2, 3, 4, 5 | 3, 4, 5 | 4, 5 | 5   |                           |
|          | Posizione della testina | 65                      | 12         | 85      | 40   | 100 |                           |
|          | Richiesta selezionata   | 1                       | 2          | 3       | 4    | 5   |                           |
|          | Tempo di ricerca        | 53                      | 73         | 45      | 60   | 25  |                           |
| SSTF     | Tempo di decisione      | 160                     | 180        | 190     | 215  | 275 | 143                       |
|          | Richieste pendenti      | 1, 2, 3, 4              | 1, 3, 4, 5 | 1, 3, 4 | 1, 3 | 1   |                           |
|          | Posizione della testina | 65                      | 85         | 75      | 100  | 40  |                           |
|          | Richiesta selezionata   | 2                       | 5          | 4       | 3    | 1   |                           |
|          | Tempi di ricerca        | 20                      | 10         | 25      | 60   | 28  |                           |
| Look     | Tempo di decisione      | 160                     | 180        | 195     | 220  | 255 | 123                       |
|          | Richieste pendenti      | 1, 2, 3, 4              | 1, 3, 4, 5 | 1, 3, 5 | 1, 3 | 1   |                           |
|          | Posizione della testina | 65                      | 85         | 100     | 75   | 40  |                           |
|          | Richiesta selezionata   | 2                       | 4          | 5       | 3    | 1   |                           |
|          | Tempi di ricerca        | 20                      | 15         | 25      | 35   | 28  |                           |
| C-Look   | Tempo di decisione      | 160                     | 180        | 195     | 283  | 311 | 186                       |
|          | Richieste pendenti      | 1, 2, 3, 4              | 1, 3, 4, 5 | 1, 3, 4 | 3, 5 | 5   |                           |
|          | Posizione della testina | 65                      | 85         | 100     | 12   | 40  |                           |
|          | Richiesta selezionata   | 2                       | 4          | 1       | 3    | 5   |                           |
|          | Tempi di ricerca        | 20                      | 15         | 88      | 28   | 35  |                           |



**Figura 14.16** Riepilogo dello scheduling del disco utilizzando le politiche FCFS, SSTF, Look e C-Look.

$$t_{\text{hm}} = t_{\text{const}} + |\text{track}_1 - \text{track}_2| \times t_{\text{pt}}$$

dove  $t_{\text{const}}$  è una costante,  $t_{\text{pt}}$  è il tempo necessario per spostare la testina da una traccia a un'altra e  $t_{\text{hm}}$  è il tempo totale di spostamento della testina. Assumiamo che la latenza rotazionale e i tempi di trasferimento dei dati siano trascurabili,  $t_{\text{const}} = 0$  ms e  $t_{\text{pt}} = 1$  ms. Un valore tipico per  $t_{\text{const}}$  è 2 ms. Inoltre, la formula per il calcolo di  $t_{\text{hm}}$  è non lineare.

La [Figura 14.16](#) mostra i seguenti dettagli per ogni decisione: tempo in cui è presa la decisione, richieste pendenti e posizione della testina in quel momento, l'operazione di I/O schedulata e il suo tempo di ricerca. L'ultima colonna mostra il tempo di ricerca totale per ogni politica. I grafici nella parte bassa della figura mostrano il movimento della testina del disco per ogni politica. Si noti che il tempo di ricerca totale nelle varie politiche di scheduling varia sensibilmente. SSTF è meglio di FCFS; tuttavia look ha il

tempo di ricerca più breve in questo esempio. È meglio di *C-Look* poiché può invertire la direzione della testina del disco dopo aver completato l'operazione di I/O sulla traccia 100 e servire le operazioni sulle tracce 75, 40 e 12, mentre *C-Look* avvia una nuova scansione con l'operazione sulla traccia 12.

Lo scheduling nel disco può essere più efficiente dello scheduling nel IOCS fisico poiché il disco utilizza un modello più preciso che considera il tempo di ricerca e anche la latenza rotazionale di un blocco. Dunque può distinguere meglio tra due comandi di I/O che apparirebbero equivalenti al IOCS fisico. Come esempio, si considerino i comandi di I/O che riguardano i blocchi che si trovano a  $+n$  e  $-n$  tracce rispetto alla posizione corrente delle testine del disco. Entrambi i comandi hanno lo stesso tempo di ricerca; il IOCS fisico dovrebbe effettuare una scelta casuale tra i due. Comunque, data la posizione rotazionale corrente dei piatti e la posizione del blocco o del settore richiesto, il disco potrebbe riscontrare che il blocco a  $+n$  tracce potrebbe essere già passato sotto le testine nel momento in cui le stesse fossero posizionate sulla traccia. Ciò significa che il blocco del disco può essere letto solo nella successiva rivoluzione del disco. Il blocco che si trova a  $-n$  tracce di distanza, dall'altra parte, potrebbe passare sotto le testine alcuni istanti dopo che le testine sono state posizionate sulla traccia. Dunque la latenza rotazionale sarebbe inferiore rispetto a quella della traccia a distanza  $+n$ . Tali sottili distinzioni contribuiscono a un throughput del disco più elevato.

### ***Scheduling nei dischi SCSI***

Un disco SCSI può accettare fino a 32 comandi concorrenti da parte del IOCS fisico. Il IOCS fisico associa un tag a ogni comando di I/O per indicare come vuole che il disco lo gestisca. Il disco memorizza i comandi in una tabella dei comandi e utilizza i rispettivi tag durante le decisioni di schedulazione. Questa funzionalità è chiamata *tagged command queuing*.

Il tag in un comando può essere di tre tipi - tag per coda semplice, tag per coda ordinata e tag per testa della coda. Un *tag per coda semplice* indica che il comando può essere riordinato per ottimizzare il throughput del disco. Un comando con un *tag per coda ordinata* indica che tutti i comandi che sono stati inseriti nella coda precedentemente devono essere schedulati prima che il comando venga schedulato. Tale comando dovrebbe essere eseguito periodicamente per garantire che le operazioni di I/O non incorrano in starvation, ovvero che non rimangano indefinitamente nella tabella dei comandi. Un comando con un *tag per testa della coda* deve essere eseguito immediatamente; ovvero deve essere eseguito prima di ogni altro comando. Questa caratteristica può essere utilizzata per garantire che i file dati siano scritti sul disco prima dei metadati (discussione sul file system journaling nel Paragrafo 13.12).

Lo scheduling nel disco ha anche i suoi svantaggi. Poiché il disco tratta tutte le operazioni di I/O uniformemente, potrebbe interferire con le ottimizzazioni a livello file eseguite dai moduli del metodo di accesso. Si consideri l'elaborazione di un file sequenziale mediante alcuni buffer, che discuteremo successivamente nel Paragrafo 14.9. Quando il file è aperto, il livello del metodo di accesso lancia i comandi per leggere i primi record del file e inserirli nei buffer. Per sfruttare i vantaggi del buffering, questi comandi di lettura dovrebbero essere effettuati nell'ordine in cui sono stati lanciati. Tuttavia, il disco potrebbe riordinarli sulla base delle loro latenze di ricerca e rotazionali. Dunque un record successivo del file può essere letto mentre un processo è in attesa di accedere a un record precedente! Quando si utilizza un disco sia per la paginazione che per i file degli utenti, il SO può voler eseguire le operazioni di paginazione con priorità più elevata. Lo scheduling nel disco può interferire con questa necessità.

Questi svantaggi dello scheduling del disco conducono all'ovvia domanda - lo scheduling del disco dovrebbe essere eseguito nel disco, nel IOCS fisico o in entrambi? L'uso di un modello più preciso per il calcolo delle latenze di ricerca e rotazionali indica che lo scheduling dovrebbe essere eseguito nel disco. Le esigenze di riordinamento dei comandi per supportare l'ottimizzazione dell'accesso a livello file implica che lo scheduling dovrebbe essere eseguito anche nel IOCS fisico. Un progettista di SO deve usare l'accodamento dei comandi con tag per garantire che questi scheduleri possano funzionare in maniera sinergica.

## **14.8 Tempo di trasferimento nello scheduling del disco**

L'esempio mostra il ruolo del tempo di trasferimento nel calcolo del tempo di accesso ai blocchi o settori del disco. Si supponga che nella coda delle richieste di una unità disco

composta da 200 tracce, si trovano, nell'ordine, le richieste dei dati ai seguenti blocchi:

39 700 - 304 - 115 - 2600 - 2120 - 270 - 321 - 0 - 760 - 20 000

tal che ogni blocco  $i$ -esimo sia memorizzato alla traccia  $i \bmod 200$ . La testina ha eseguito l'ultimo movimento portandosi dalla traccia 85 alla traccia 97. Si ipotizzi che lo spostamento da una traccia a un'altra richieda un tempo medio pari a  $40 \mu s$  per traccia, che l'inversione della direzione di movimento della testina richieda mediamente  $80 \mu s$  e la velocità di rotazione sia 7200 rpm. Si vuole determinare il tempo richiesto, complessivamente, per accedere alle tracce indicate per le politiche di scheduling: (a) Shortest Seek Time First; (b) C-SCAN e (D) LOOK. Si vuole, inoltre, determinare la velocità di trasferimento dati massima necessaria per trasferire blocchi da 4 K in un tempo pari al più al 60% del tempo necessario per blocchi di 8 K.

### 14.8.1 Soluzione

Affrontiamo la soluzione, ricordando come si calcola la latenza rotazionale che, nel nostro caso, risulta essere pari a:  $\frac{60}{2 * 7200} = 4.17 \text{ ms}$ .

Poiché ci viene indicato il blocco al quale avviene una richiesta, dobbiamo determinare la traccia alla quale si trova. Poiché il blocco  $i$ -esimo è posizionato alla traccia  $i \bmod 200$ , l'accesso deve avvenire:

blocchi: 39 700 304 115 2600 2120 270 321 0 760 20 000

tracce: 100 104 115 0 120 70 121 200 160 0

Vediamo ora il comportamento delle politiche di scheduling SSTF, S-SCAN e LOOK.

#### 1. SSTF:

la sequenza di scheduling risulta essere: 97  $\rightarrow$  100  $\rightarrow$  104  $\rightarrow$  115  $\rightarrow$  120  $\rightarrow$  121  $\rightarrow$  160  $\rightarrow$  200  $\rightarrow$  70  $\rightarrow$  0. Le distanze tra tracce della sequenza sono pertanto: 3 4 11 5 1 39 40 130 70. Il tempo di accesso risulta essere  $T_a = (303 * 40 \mu s) + 80 \mu s + (9 * 4.17 \text{ ms}) = 12.12 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 49.73 \text{ ms}$ .

#### 2. C-SCAN:

la sequenza di tracce attraversate dalla politica di scheduling risulta: 97  $\rightarrow$  100  $\rightarrow$  104  $\rightarrow$  115  $\rightarrow$  120  $\rightarrow$  121  $\rightarrow$  160  $\rightarrow$  200  $\rightarrow$  0  $\rightarrow$  70. Le distanze fra le tracce calcolate a partire dalla sequenza sono: 3 4 11 5 1 39 40 200 70. Il tempo di accesso risulta quindi  $T_a = (373 * 40 \mu s) + 80 \mu s + (9 * 4.17 \text{ ms}) = 14.92 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 52.53 \text{ ms}$ .

#### 3. LOOK:

la sequenza delle tracce attraversate dalla politica LOOK sono: 97  $\rightarrow$  100  $\rightarrow$  104  $\rightarrow$  115  $\rightarrow$  120  $\rightarrow$  121  $\rightarrow$  160  $\rightarrow$  200  $\rightarrow$  70  $\rightarrow$  0. Le distanze tra le tracce attraversate sono pertanto: 3 4 11 5 1 39 40 130 70. Il tempo di accesso risulta quindi  $T_a = (303 * 40 \mu s) + 80 \mu s + (9 * 4.17 \text{ ms}) = 12.12 \text{ ms} + 0.08 \text{ ms} + 37.53 \text{ ms} = 49.73 \text{ ms}$ .

La velocità di trasferimento non dipende dall'algoritmo di scheduling utilizzato, ma dalla velocità di rotazione nel seguente modo:

$$T_t = \frac{b}{r * N}$$

dove  $b$  = numero di blocchi da trasferire,  $N$  = numero di blocchi per traccia e  $r$  = velocità di rotazione in rps. Il valore di  $t_t$  per ogni politica di scheduling considerata in precedenza è, nel nostro caso, trascurabile. Per completare la soluzione dell'esempio, si richiede che:

$$\frac{4KB}{r * N} \leq 60\% \text{ di } \frac{8KB}{r * N}$$

ovvero:

$$\frac{4KB}{r * N} \leq \frac{60}{100} \frac{8KB}{r * N} \rightarrow 4KB \leq 0.6 * 8KB \rightarrow 4 \leq 4.8$$

Questa diseguaglianza è sempre verificata per cui, qualunque sia la velocità di trasferimento di 8 KB, quella necessaria per trasferire 4 KB sarà sempre al più pari al 60% di quella per 8 KB.

## 14.9 Buffering dei record

Per elaborare i record in un file sequenziale utilizzando il IOCS fisico, un processo avvia un'operazione di lettura su un record richiamando il modulo `start-io` e richiama immediatamente il modulo `await-io` per verificare se l'operazione di lettura è stata completata. Il modulo `await-io` blocca il processo finché non viene completata l'operazione (Paragrafo 14.5.4). In questo modo il processo deve attendere un certo periodo di tempo per ogni blocco, facendo degradare le prestazioni. Un metodo di accesso per i file sequenziali riduce i tempi di attesa di un processo mediante la tecnica del *buffering dei record*, che tenta di sovrapporre le attività di I/O e di CPU di un processo. Questo obiettivo è perseguito in due modi:

- *prefetching* di un record di input in un buffer di I/O;
- *postwriting* di un record di output da un buffer di I/O

dove un *buffer di I/O*, o semplicemente *buffer*, è un'area di memoria temporaneamente usata per memorizzare i dati coinvolti in un'operazione di I/O.

Nel prefetching, l'operazione di I/O per leggere il prossimo record e memorizzarlo in un buffer è avviata poco prima che il record sia richiesto dal processo - può essere avviata mentre il processo è impegnato nell'elaborazione del record precedente. Questa organizzazione sovrappone una parte del tempo impiegato per la lettura del prossimo record con l'elaborazione del record precedente, riducendo il tempo di attesa per il record successivo. Nel postwriting, il record che deve essere scritto viene semplicemente copiato in un buffer quando il processo esegue un'operazione di scrittura in modo che l'esecuzione del processo possa proseguire. L'effettiva scrittura è eseguita dal buffer in un momento successivo. Può sovrapporsi con (una parte del) l'elaborazione del record successivo.

Utilizziamo la seguente notazione durante la discussione della tecnica del buffering:

- $t_{io}$  tempo di I/O di un record (Equazione 14.1);
- $t_c$  tempo di copia di un record (ovvero la quantità di tempo di CPU necessaria per copiare un record da un'area di memoria a un'altra);
- $t_p$  tempo di elaborazione di un record (ovvero la quantità di tempo di CPU necessaria per elaborare un record);
- $t_w$  tempo di attesa di un record (ovvero la quantità di tempo che il processo deve attendere prima che il prossimo record sia disponibile per l'elaborazione);
- $t_{ee}$  tempo effettivo trascorso per ogni record (ovvero l'intervallo tra il momento in cui un processo vuole iniziare l'elaborazione di un record e il momento in cui l'elaborazione del record è completata).

$t_w$  e  $t_{ee}$  sono definiti in modo analogo per un file di output.

Si consideri un programma che legge ed elabora 100 record da un file sequenziale F. Consideriamo tre versioni del programma chiamate *Unbuf\_P*, *Single\_buf\_P* e *Multi\_buf\_P* che usano, rispettivamente, nessun buffer, 1 buffer ed  $n$  buffer con  $n > 1$ . Assumiamo che  $t_{io} = 75$  ms,  $t_p = 50$  ms e  $t_c = 5$  ms.

La [Figura 14.17](#) illustra il funzionamento e le prestazioni dei processi che rappresentano l'esecuzione di *Unbuf\_P*, *Single\_buf\_P* e *Multi\_buf\_P*. Per convenienza, assumiamo che un processo abbia lo stesso nome del programma che esegue. Ogni colonna della figura mostra il codice di un programma, illustra i passi necessari per la lettura e l'elaborazione di un record e mostra un grafico dei tempi relativo alle prestazioni del processo. Le istruzioni "avvia un'operazione di I/O" e "attendi il completamento dell'I/O" sono tradotte in chiamate ai moduli del IOCS fisico `start-io` e `await-io` con gli operandi appropriati. L'istruzione per cominciare l'I/O legge il prossimo record di F, se possibile, e lo inserisce in un'area di memoria. Se non ci sono più record

in F, la condizione *end\_of\_file* è impostata quando si esegue l'istruzione di attesa dell'I/O. *Unbuf\_P* utilizza una singola area di memoria chiamata *Rec\_area* per leggere ed elaborare un record del file F [Figura 14.17(a)]. Lancia un'operazione di lettura e attende il completamento prima di elaborare il record in *Rec\_area*. Il diagramma dei tempi mostra che l'I/O è eseguito su *Rec\_area* da 0 a 75 ms, mentre l'elaborazione del record contenuto in *Rec\_area* avviene tra 75 ms e 125 ms. Quindi  $t_w = t_{io}$  e  $t_{ee} = t_{io} + t_p$ . Questa sequenza di operazioni si ripete 100 volte, per cui il tempo necessario al processo è  $100 \times (75 + 50) \text{ ms} = 12.5 \text{ secondi}$ .

#### Programmi

##### Programma *Unbuf\_P*

```
avvia un'operazione di I/O per
  read (F, Rec_area);
  attende il completamento dell'I/O;
while (not end_of_file(F))
begin
  elabora Rec_area;
  avvia un'operazione di I/O per
    read (F, Rec_area);
  attende il completamento dell'I/O;
end
```

##### Programma *Single\_buf\_P*

```
avvia un'operazione di I/O per
  read (F, Buffer);
  attende il completamento dell'I/O;
while (not end_of_file(F))
begin
  copia Buffer in Rec_area;
  avvia un'operazione di I/O per
    read (F, Buffer);
  elabora Rec_area;
  attende il completamento dell'I/O;
```

##### Programma *Multi\_buf\_P*

```
for i := 1 to n
  avvia un'operazione di I/O
  per read (F, Bufi);
  attende il completamento dell'I/O su
  Bufi;
k := 1;
while (not end_of_file(F))
  copia Bufi in Rec_area;
  avvia un'operazione di I/O per
    read (F, Bufk);
  elabora Rec_area;
k := (k mod n) + 1;
  attende il completamento dell'I/O
  su Bufk;
end
```

#### Attività di I/O, copia ed elaborazione (UP:*Unbuf\_P*, SP:*Single\_buf\_P*, MP:*Multi\_buf\_P*)

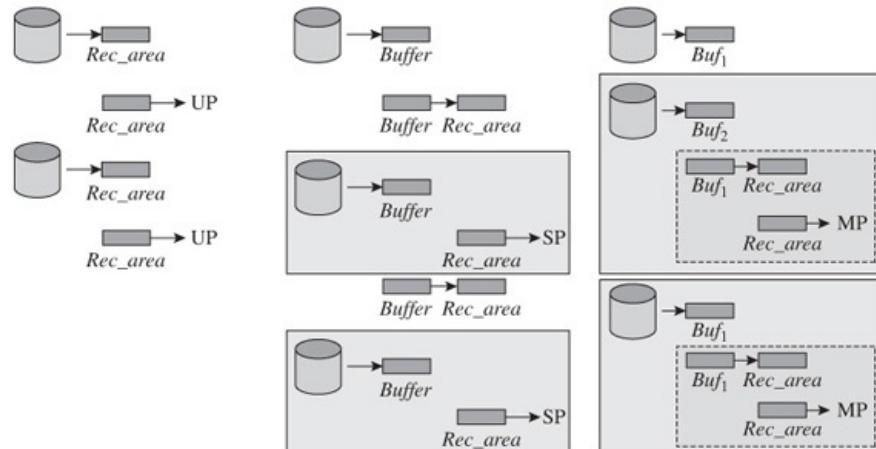
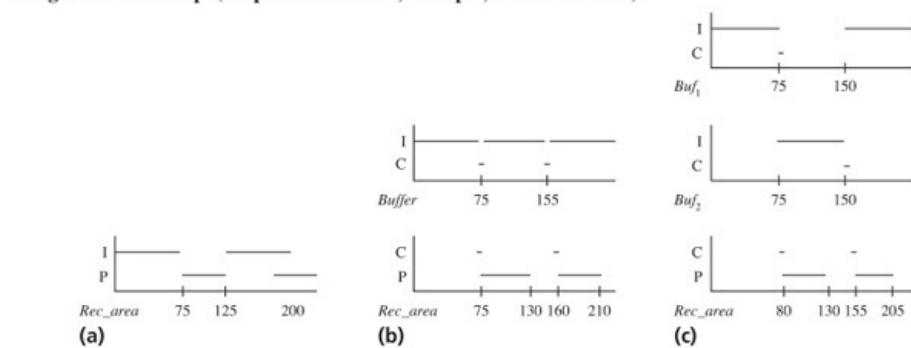


Diagramma dei tempi (I: operazione di I/O, C: copia, P: elaborazione)



**Figura 14.17** Elaborazione di un file non bufferizzato e bufferizzato. (Nota: la condizione *end\_of\_file* è impostata quando viene eseguita l'istruzione attende il completamento dell'I/O per un'operazione che tenta di leggere oltre la fine di un file.)

La Figura 14.17(b) illustra il funzionamento di *Single\_buf\_P*, che utilizza un singolo buffer chiamato *Buffer*. Il processo esegue un'operazione per leggere il primo record, memorizzarlo in *Buffer* e attendere il completamento. A questo punto entra nel ciclo principale del programma, che ripete i seguenti quattro passi 99 volte.

1. Copia il record da *Buffer* a *Rec\_area*.
2. Avvia un'operazione I/O su *Buffer*.
3. Elabora il record memorizzato in *Rec\_area*.
4. Attendi la fine dell'operazione di I/O su *Buffer*.

Come mostrato nel diagramma dei tempi di [Figura 14.17\(b\)](#), il processo al passo 1 deve attendere che venga completata l'operazione di lettura su *Buffer*. Quindi esegue i passi 2-4. Dopo aver copiato il record in *Rec\_area*, avvia un'operazione di lettura per il secondo record e inizia l'elaborazione del primo record. Questo parallelismo viene rappresentato disegnando un rettangolo che racchiude queste due azioni nella parte delle attività di [Figura 14.17\(b\)](#). Il passo 1, ovvero la copia del prossimo record da *Buffer* a *Rec\_area*, è eseguito solo dopo che entrambe le operazioni sono terminate. Completata questa operazione, si procede con l'elaborazione di un record e la lettura del record successivo in parallelo. Dunque, il tempo di attesa prima dell'elaborazione di ognuno dei record 2-99 è

$$\begin{aligned} t_w &= (t_{io} - t_p) + t_c, \text{ se } t_{io} > t_p \\ &= t_c, \quad \text{se } t_{io} \leq t_p \end{aligned} \quad (14.3)$$

per cui il buffering è più efficiente quando  $t_{io} \leq t_p$ .

Per i record 2-99, l'effettivo tempo trascorso per ogni record ( $t_{ee}$ ) è dato da

$$\begin{aligned} t_{ee} &= t_w + t_p \\ &= t_c + \max(t_{io}, t_p) \end{aligned} \quad (14.4)$$

Dunque il processo passa attraverso tre fasi distinte - la fase di avvio in cui viene letto il primo record, la fase centrale in cui un record è copiato ed elaborato mentre il record successivo è letto in parallelo e la fase finale in cui l'ultimo record viene copiato ed elaborato. Quindi, il tempo totale del processo è dato da:

$$\text{tempo totale trascorso} = t_{io} + (\text{numero di record} - 1) \times t_{ee} + (t_c + t_p) \quad (14.5)$$

Dall'Equazione 14.4 e 14.5,  $t_{ee}$  è 80 ms e il tempo totale del processo è  $75 + 99 \times 80 + 55$  ms = 8.05 secondi. Se  $t_{io}$  fosse di 50 ms, il tempo totale del processo sarebbe 5.55 secondi.

La [Figura 14.17\(c\)](#) illustra l'esecuzione del processo *Multi\_buf\_P*, che utilizza i buffer chiamati  $Buf_1, Buf_2 \dots, Buf_n$ . All'inizio dell'elaborazione del file, *Multi\_buf\_P* avvia le operazioni di I/O su tutti gli  $n$  buffer. All'interno del ciclo di elaborazione dei file, utilizza i buffer a turno, seguendo i quattro passi del ciclo del programma per elaborare un record presente in un buffer. L'istruzione  $k := (k \bmod n) + 1$ ; garantisce che i buffer siano utilizzati in maniera ciclica. Il processo attende il completamento dell'I/O sul buffer successivo, copia il record dal buffer in *Rec\_area*, richiama *start-io* per leggere il record successivo e inserirlo nel buffer e infine elabora il record presente in *Rec\_area*.

La presenza di più buffer causa una differenza significativa tra le operazioni di *Multi\_buf\_P* e quelle di *Single\_buf\_P*. Si consideri l'elaborazione dei primi due record da parte di *Multi\_buf\_P* [[Figura 14.17\(c\)](#)]. Quando viene completato l'I/O su  $Buf_1$ , *Multi\_buf\_P* copia il primo record da  $Buf_1$  in *Rec\_area* e inizia l'elaborazione.

Un'operazione di lettura su  $Buf_1$  sarebbe stata richiesta prima, per cui il IOCS fisico avrebbe iniziato questa operazione di lettura al momento del completamento dell'I/O su  $Buf_1$ . Quindi questa operazione sarebbe stata sovrapposta con la copia del primo record da  $Buf_1$ . In [Figura 14.17\(c\)](#), mostriamo questo parallelismo come segue: il rettangolo tratteggiato intorno alla copia e all'elaborazione del record contenuto in  $Buf_1$  indica che queste azioni sono eseguite sequenzialmente. Il rettangolo che racchiude il primo rettangolo e l'operazione di I/O su  $Buf_2$  indica che queste due attività sono eseguite in parallelo.

Di conseguenza, il tempo effettivo necessario per ogni record è dato da:

$$t_w = \begin{cases} t_{io} - t_p & \text{se } t_{io} > t_c + t_p \\ t_c, & \text{se } t_{io} \leq t_c + t_p \end{cases} \quad (14.6)$$

$$t_{ee} = \max (t_{io}, t_c + t_p) \quad (14.7)$$

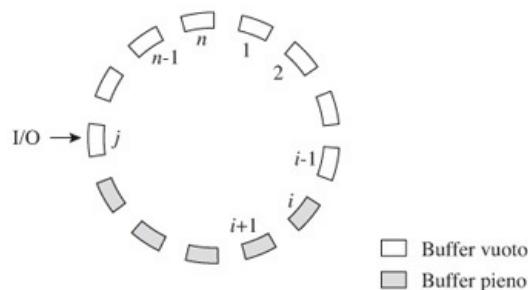
Dall'Equazione 14.7,  $t_{ee} = 75$  ms. Il tempo totale trascorso, che dipende dall'Equazione (14.5), è  $75 + 99 \times 75 + 55$  ms = 7.555 secondi, che risulta leggermente migliore rispetto al tempo di 8.05 secondi relativo a *Single\_Buf\_P*.

Il rapporto tra il tempo trascorso relativo a *Unbuf-P* e quello relativo a *Multi\_buf-P* è il fattore di speedup che si ottiene utilizzando più buffer. Considerando lo stato stabile, il fattore di speedup è approssimativamente:

$$\frac{t_{io} + t_p}{\max (t_{io}, t_c + t_p)}$$

Dall'Equazione 14.7, si può vedere che il valore migliore è ottenuto quando  $t_{io} = t_c + t_p$ . Questo valore ha come limite superiore 2.

Si consideri l'esecuzione di *Multi\_Buf\_P* in cui vengono utilizzati più buffer. La Figura 14.18 illustra la situazione tipica durante l'esecuzione di *Multi\_Buf\_P*. La CPU ha appena copiato il record da  $Buf_{i-1}$  in *Rec\_area* ed ha avviato un'operazione di I/O su  $Buf_{i-1}$ . Dunque, le operazioni di I/O sono state iniziate su tutti gli  $n$  buffer. Alcune operazioni di I/O, in particolare, quelle su  $Buf_i, \dots, Buf_{j-1}$ , sono già completate. L'I/O è ancora in esecuzione per  $Buf_j$ , mentre  $Buf_{j+1}, \dots, Buf_n, Buf_1, \dots, Buf_{i-1}$  si trovano nella coda per l'inizio dell'I/O. Quindi  $(j - i)$  buffer sono al momento pieni, l'I/O è in esecuzione per un buffer e  $(n - j + i - 1)$  buffer sono nella coda per l'I/O. Il valore di  $(j - i)$  dipende dai valori di  $t_{io}$  e  $t_p$ . Se  $t_{io} < t_p$ , ovvero, se l'operazione di I/O per un record richiede meno tempo della sua elaborazione, allora i buffer  $Buf_{i+1}, \dots, Buf_n, Buf_1, \dots, Buf_{i-1}$  saranno pieni e  $Buf_{i-1}$  sarà impegnato da un'operazione di I/O o sarà pieno nel momento in cui la CPU sta elaborando il record copiato da  $Buf_{i-1}$ . Se  $t_{io} > t_p$ , la situazione di stato stabile si verificherà quando  $Buf_i$  è impiegato da un'operazione di I/O nel momento in cui la CPU sta elaborando il record copiato da  $Buf_{i-1}$  e i buffer  $Buf_{i+1}, \dots, Buf_{i+1}, \dots, Buf_n, \dots, Buf_{i-1}$  sono vuoti.



**Figura 14.18** Uso dei buffer in *Buf\_P*.

L'uso di più buffer è irrilevante se un processo elabora ogni record singolarmente. Tuttavia, crea una differenza sostanziale se un processo elabora molti record insieme. L'utilizzo di  $n$  buffer in questo caso è di aiuto poiché molti buffer possono essere pieni quando il processo necessita di alcuni record insieme. Il prossimo esempio illustra questo punto.

#### Esempio 14.4 - Uso di più buffer

Ogni riga di codice di un programma scritto nel linguaggio L è memorizzata in un record del file F. Il compilatore di L utilizzato per compilare questo programma deve leggere un'intera istruzione prima di iniziare l'elaborazione. Un'istruzione può contenere fino a 1 righe. L'attesa dell'I/O da parte del compilatore può essere eliminata solo se si verificano due condizioni:

1.  $t_{io} \leq t_{pl}$ , e
2.  $l \leq n$

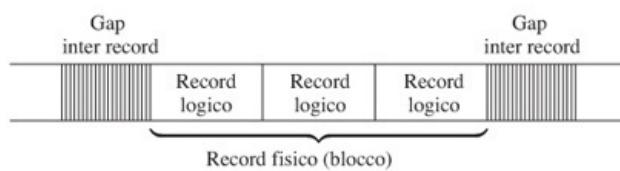
dove  $t_{pl}$  è il tempo di elaborazione medio per ogni riga di un'istruzione. La condizione 1 garantisce che, nello stato stabile, tutti i buffer saranno pieni nel momento in cui il compilatore avrà terminato l'elaborazione di un'istruzione. La condizione 2 garantisce che almeno  $l$  buffer saranno pieni quando il compilatore avrà terminato l'elaborazione di un'istruzione. Quindi il compilatore non dovrà attendere. Dovrà invece attendere se  $l > n$ , per esempio, se  $l = 3$  e utilizza 2 buffer.

## 14.10 Blocking dei record

Nell'elaborazione non bufferizzata di un file, il tempo speso nell'esecuzione delle operazioni di I/O può costituire una parte consistente del tempo di esecuzione del processo. Anche nell'elaborazione bufferizzata di un file,  $t_w > 0$  se  $t_{io} > t_p$  o  $t_p > t_c + t_p$  (Equazioni 14.3 e 14.6). In questo modo sia l'elaborazione dei file non bufferizzata che bufferizzata beneficiano di una riduzione del  $t_{io}$ . La tecnica del *blocking dei record* riduce il tempo effettivo dell'I/O, per ogni record, leggendo o scrivendo molti record in una singola operazione di I/O. Dall'Equazione (14.1),  $t_{io} = t_a + t_x$ . Dunque, un programma che elabora due record contenuti in un file che non adotta il blocking avrebbe bisogno di un tempo totale di I/O di  $2 \times (t_a + t_x)$ . Se viene utilizzato il blocking e un processo legge o scrive due record in una singola operazione di I/O, il tempo totale di I/O viene ridotto a  $t_a + 2 \times t_x$ .

### Record logici e fisici

Quando più record vengono letti o scritti insieme, è necessario differenziare l'accesso e l'elaborazione dei dati e come vengono scritti sul dispositivo di I/O. Un *record logico* è l'unità di dati utilizzata in un processo per l'accesso e l'elaborazione. Un *record fisico*, anche chiamato *blocco*, è l'unità dei dati utilizzata per il trasferimento da e per un dispositivo di I/O. Il *fattore di blocking* di un file è il numero di record logici in un record fisico. Si dice che un file adotta il *blocking* dei record se il fattore di blocking è maggiore di 1. La [Figura 14.19](#) mostra un file che utilizza un fattore di blocking pari a 3. Si noti che quando si utilizza il blocking, i gap inter record sul supporto di I/O separano i record fisici, ovvero i blocchi, piuttosto che i record logici.



**Figura 14.19** Un file con fattore di blocking = 3.

### Azioni di deblocking

Un'operazione di lettura su un file che utilizza il blocking dei record trasferisce  $m$  record logici alla memoria, dove  $m$  è il fattore di blocking. Le azioni necessarie per l'estrazione di un record logico da un blocco, in modo che possa essere utilizzato in un processo, sono collettivamente chiamate *azioni di deblocking*.

La [Figura 14.20](#) mostra un programma che elabora un file che utilizza il blocking dei record in maniera non bufferizzata. Il ciclo principale del programma legge un record fisico in ogni iterazione. Questo contiene un ciclo interno che estrae i record logici da un record fisico e li elabora. In questo modo, un'operazione di I/O è avviata solo dopo l'elaborazione di  $m$  record. Una logica simile può essere adottata nei programmi delle Figure 17(b) e (c) per ottenere l'elaborazione bufferizzata di un file che utilizza il blocking dei record.

---

```

avvio di un'operazione di I/O per leggere (F, Rec.area);
attesa del completamento dell'I/O;
while (not end_of_file(F))
    for i := 1 to m
        { estrazione dell'i-esimo record in Rec.area ed elaborazione }
        avvio di un'operazione di I/O per leggere (F, Rec.area);
        attesa del completamento dell'I/O;
    end

```

---

**Figura 14.20** Elaborazione di un file che utilizza il blocking dei record in maniera non bufferizzata.

### Scelta del fattore di blocking

Generalizzando la discussione precedente, possiamo dire che se  $s_{lr}$  e  $s_{pr}$  rappresentano la dimensione, rispettivamente, di un record logico e fisico,  $s_{pr} = m \times s_{lr}$ . Il tempo di I/O per ogni record fisico,  $(t_{io})_{pr}$  e il tempo di I/O per ogni record logico,  $(t_{io})_{lr}$ , sono dati da

$$(t_{io})_{pr} = t_a + m \times t_x \quad (14.8)$$

$$(t_{io})_{lr} = \frac{t_a}{m} + t_x \quad (14.9)$$

Dunque il blocking riduce il tempo di I/O effettivo per ogni record logico, di cui beneficiano le elaborazioni bufferizzata e non bufferizzata. Se  $t_x < t_p$ , con una scelta appropriata di  $m$  è possibile ridurre  $(t_{io})_{lr}$  in modo tale che  $(t_{io})_{lr} \leq t_p$ . Ottenuto ciò, dalle Equazioni 14.3 e 14.6 segue che il buffering può essere utilizzato per ridurre il tempo di attesa per ogni record a  $t_c$ . Il prossimo esempio illustra come  $(t_{io})_{lr}$  varia in base al fattore di blocking.

### Esempio 14.5 - Blocking dei record

La [Tabella 14.5](#) mostra la variazione di  $(t_{io})_{lr}$  in base a  $m$  per un disco con  $t_a = 10\text{ms}$ , transfer rate di 800 kB/s, dove 1 kB/s = 1000 byte al secondo  $s_{lr} = 200$  byte.  $t_x$ , tempo di trasferimento per ogni record logico, è  $\frac{200}{800}$  ms, ovvero 0.25 ms.  $(t_{io})_{pr}$  e  $(t_{io})_{lr}$  sono calcolati secondo le Equazioni 14.8 e 14.9. Se  $t_p = 3$  ms,  $m \geq 4$  implica  $(t_{io})_{lr} < t_p$ .

| Fattore di blocking ( $m$ ) | Dimensione del blocco | $t_a$ ms | $m \times t_x$ ms | $(t_{io})_{pr}$ ms | $(t_{io})_{lr}$ ms |
|-----------------------------|-----------------------|----------|-------------------|--------------------|--------------------|
| 1                           | 200                   | 10       | 0.25              | 10.25              | 10.25              |
| 2                           | 400                   | 10       | 0.50              | 10.50              | 5.25               |
| 3                           | 600                   | 10       | 0.75              | 10.75              | 3.58               |
| 4                           | 800                   | 10       | 1.00              | 11.00              | 2.75               |

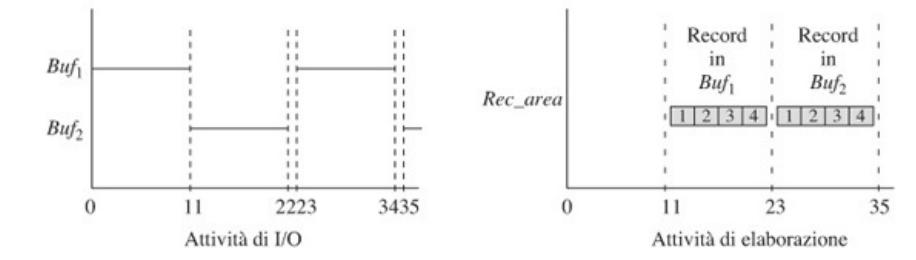
**Tabella 14.5** Variazione di  $(t_{io})_{lr}$  in base al fattore di blocking.

Il valore di  $m$  è limitato inferiormente dall'esigenza di rendere  $(t_{io})_{lr} \leq t_p$ . Superiormente è limitato dalla quantità di memoria utilizzata per i buffer e dalla dimensione di una traccia o settore del disco. Un valore pratico del fattore di blocking è il più piccolo valore di  $m$  che rende  $(t_{io})_{lr} \leq t_p$ . Il prossimo esempio illustra l'elaborazione di un file che adotta sia il blocking che il buffering dei record.

### Esempio 14.6 - Elaborazione bufferizzata di un file che utilizza il blocking dei record

La [Figura 14.21](#) mostra il grafico dei tempi di elaborazione del file dell'Esempio 14.5 con fattore di blocking pari a 4, utilizzando due buffer  $Buf_1$  e  $Buf_2$ . Assumiamo che  $t_c$  sia trascurabile. Quando si apre il file al tempo 0 secondi, le operazioni di lettura sono avviate su  $Buf_1$  e  $Buf_2$ . L'operazione su  $Buf_1$  viene completata a  $t = 11$  ms. Il processo estrae un record logico da  $Buf_1$  e lo elabora.  $t_p = 3$  ms, per cui l'elaborazione dei quattro record di  $Buf_1$  necessita di 12 ms. Questa si sovrappone con l'operazione di

lettura su  $Buf_2$ , che necessita di 11 ms. Quindi il successivo record logico del file viene letto e inserito in  $Buf_2$  prima che venga completata l'elaborazione dei record in  $Buf_1$ . Il processo avvia l'elaborazione dei record logici copiati da  $Buf_2$  a  $t = 23$  ms. Quindi, non ci saranno periodi di attesa dopo la fase di avvio.



**Figura 14.21** Elaborazione bufferizzata di un file che utilizza il blocking dei record con fattore di blocking = 4 e 2 buffer.

## 14.11 Metodi di accesso

Come menzionato nel Paragrafo 13.3.4, un metodo di accesso fornisce supporto per l'elaborazione efficiente di una classe di file che usa un'organizzazione specifica. Per le configurazioni fondamentali dei file discusse nel Paragrafo 13.3, il IOCS può fornire metodi di accesso per i seguenti tipi di elaborazione.

- Elaborazione non bufferizzata dei file ad accesso sequenziale.
- Elaborazione bufferizzata dei file ad accesso sequenziale.
- Elaborazione dei file ad accesso diretto.
- Elaborazione non bufferizzata dei file sequenziali indicizzati.
- Elaborazione bufferizzata dei file sequenziali indicizzati.

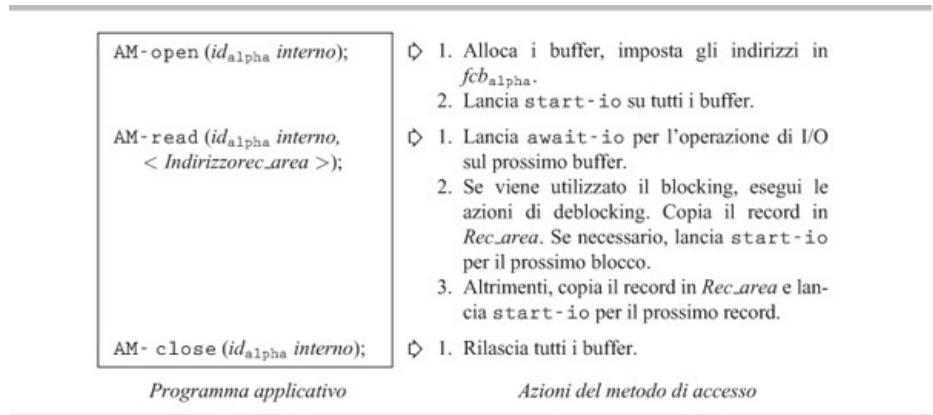
I metodi di accesso per l'elaborazione bufferizzata dei file ad accesso sequenziale e ad accesso sequenziale indicizzato incorporano la tecnica del buffering illustrata in [Figura 14.17\(c\)](#). Questi metodi di accesso possono eseguire anche il blocking dei record, utilizzando la tecnica mostrata in [Figura 14.20](#).

Assumiamo che ogni modulo che implementa un metodo di accesso fornisca tre accessi con i seguenti parametri:

1. AM-open (<id\_interno>)
2. AM-close (<id\_interno>)
3. AM-read/write (<id\_interno>, <record\_info>, <I/O\_area addr>)

I moduli del file system e del IOCS richiamano queste funzionalità per implementare l'elaborazione di un file. AM-open viene richiamata dopo la creazione di un file control block, utilizzando le informazioni contenute nell'elemento della directory relativo al file. Analogamente, AM-close è richiamata da iocs-close. AM-read/write sono richiamate da un modulo del file system; il punto di accesso AM-read è di fatto il punto di inizio del modulo di libreria seq-read di [Figura 14.14](#).

La [Figura 14.22](#) mostra le azioni del metodo di accesso per l'elaborazione bufferizzata di un file ad accesso sequenziale alpha. AM-open lancia le operazioni di lettura su tutti i buffer. AM-read utilizza le informazioni in  $fcb_{\alpha}$ , incluso  $fmt_{\alpha}$ , per creare una coppia (*id del record*, *id del byte*) nei passi 2 e 3 per il successivo record fisico nel file. Alcune azioni sarebbero differenti se alpha fosse un file di output. AM-write sarebbe richiamata per eseguire le operazioni di scrittura. Nei passi 2 e 3 richiamerebbe un modulo del file system per allocare più spazio sul disco ad alpha ed inserirebbe l'indirizzo in  $fmt_{\alpha}$ .



Programma applicativo

Azioni del metodo di accesso

**Figura 14.22** Azioni di un metodo di accesso per la lettura bufferizzata di un file.

## 14.12 Cache del disco e dei file

Una tecnica generale per velocizzare l'accesso ai dati consiste nell'utilizzare una gerarchia di memoria composta da una parte della memoria e dai file memorizzati sul disco. Si ricordi dai principi delle gerarchie di memoria discussi nel Paragrafo 2.2.3 che la memoria contiene alcune parti dei file dati memorizzati sul disco; le altre parti vengono caricate in memoria quando richieste. In pratica, la memoria funziona come una *cache* tra i file sul disco e i processi. Sia il IOCS fisico che i metodi di accesso utilizzano questo principio. Il IOCS fisico usa una *cache del disco*, che gestisce in modo uniforme tutti i file memorizzati su un disco e in ogni istante mantiene in memoria una *parte* dei dati di *alcuni* file. Un metodo di accesso, invece, utilizza una *cache dei file*, che mantiene in memoria alcune parti dei dati di un file. Il metodo di accesso usa una cache di file separata per ogni file.

I dati memorizzati in una cache del disco o del file sono tipicamente costituiti di pochi blocchi consecutivi; per semplicità assumiamo che tali dati siano in un solo blocco. Chiameremo *buffer* l'area di memoria utilizzata per memorizzare un'unità di dati. La cache è quindi un insieme di buffer gestiti dal software. Ogni buffer ha due parti - la parte *header* indica quali dati sono contenuti e la parte dati che contiene i dati. L'header contiene le seguenti informazioni:

- l'indirizzo dei blocchi sul disco da cui i dati sono stati caricati nel buffer;
- un *dirty* flag;
- le informazioni necessarie per eseguire il rimpiazzamento dei dati nel buffer, come per esempio il tempo dell'ultimo riferimento eseguito.

Quando un processo lancia un'operazione di lettura, specifica l'offset all'interno del file. Il IOCS determina l'indirizzo del blocco che contiene i dati richiesti e controlla se nella cache è presente il contenuto del blocco richiesto. In caso affermativo, i dati vengono copiati dal buffer nello spazio di indirizzamento del processo. Altrimenti, viene avviata un'operazione di I/O per caricare i dati dal blocco sul disco in un buffer nella cache e successivamente, al termine dell'operazione di I/O, nello spazio di indirizzamento del processo. Quando un processo esegue un'operazione di scrittura, il IOCS controlla se il contenuto del blocco contenente i vecchi valori dei dati è presente in un buffer. In caso affermativo, copia i valori da scrivere nel buffer dallo spazio di indirizzamento del processo e imposta il *dirty* flag del buffer a *true*. Altrimenti, copia l'indirizzo del blocco del disco e i valori dei dati da scrivere in un nuovo buffer e imposta il suo *dirty* flag a *true*. In ogni caso, il contenuto del buffer viene scritto sul disco dalla procedura descritta di seguito.

Per velocizzare le ricerche in cache, gli header dei buffer sono memorizzati in una struttura dati efficiente come una hash table. Per esempio, la *hash-with-chaining* utilizzata nella tabella delle pagine invertite dal gestore della memoria virtuale potrebbe essere adattata all'utilizzo nella cache (Figura 12.10 nel Paragrafo 12.2.3). In questa configurazione, l'indirizzo di un blocco i cui dati sono contenuti nella cache è passato alla funzione di hashing per ottenere un elemento della hash table. Tutti i buffer che contengono i blocchi che corrispondono allo stesso elemento della hash table vengono inseriti in una lista linkata, chiamata *chain* e l'elemento della tabella di hash viene fatto

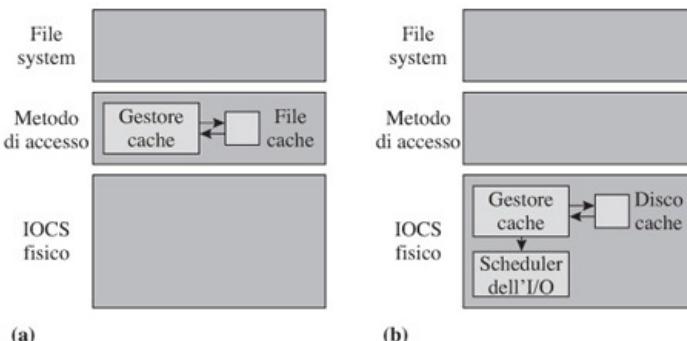
puntare alla lista. Per verificare se i dati di un blocco sono presenti nella cache, si passa l'indirizzo del blocco alla funzione di hashing per ottenere il numero di un elemento della tabella di hash e si scorre la lista puntata da questo elemento per verificare la presenza del blocco in uno dei buffer. Se il blocco non è presente nella cache, viene caricato in un buffer libero nella cache e il buffer viene aggiunto alla lista. Se la cache è piena, si utilizza una politica di rimpiazzo, come la LRU, per decidere quale buffer deve essere usato per caricare i dati richiesti. Se il *dirty* flag del buffer è *true*, il suo contenuto viene scritto nel blocco sul disco all'indirizzo indicato nell'header prima che vengano caricati i nuovi dati nel buffer. Questa organizzazione usata nella *buffer cache* di Unix sarà descritta successivamente nel Paragrafo 14.13.1.

Il caricamento di blocchi interi, con dimensione di pochi KB, nella cache sfrutta la *località spaziale* poiché i dati vicini ai dati cui si è avuto accesso di recente si troverebbero già nella cache. Questo effetto è analogo al *blocking* dei record discusso nel Paragrafo 14.10. Gli studi menzionati nel Paragrafo 14.13.1 indicano quali valori di hit ratio nella cache del disco pari o maggiori di 0.9 possono essere ottenuti riservando una piccola quantità di memoria alla cache del disco. Una cache di file può ulteriormente sfruttare la *località temporale* precaricando nella cache alcuni blocchi successivi di un file sequenziale, che corrisponde al *buffering* dei record discusso nel Paragrafo 14.9.

L'uso di una cache ha anche alcuni svantaggi. Un'operazione di I/O necessita di due operazioni di copia, una tra il disco e la cache e l'altra tra la cache e lo spazio di indirizzamento del processo che ha avviato l'operazione di I/O. L'uso della cache inoltre porta a una scarsa affidabilità del file system poiché i dati modificati sono presenti in un buffer della cache finché non sono scritti sul disco. Questi dati possono essere persi in caso di crash del sistema.

### ***Cache di file***

Una cache di file è implementata in un metodo di accesso ed ha lo scopo di fornire un accesso efficiente ai dati memorizzati in un file. Come mostrato in [Figura 14.23\(a\)](#), il metodo di accesso richiama il gestore della cache, che verifica se i dati richiesti sono disponibili nella cache. Il metodo di accesso richiama il IOCS fisico solo se la cache non contiene già i dati richiesti. Il vantaggio chiave della cache di file rispetto alla cache del disco è che il gestore della cache può utilizzare le tecniche a livello file per velocizzare gli accessi ai dati. Questa tecnica sfrutta le proprietà dell'organizzazione di un file per velocizzare gli accessi ai dati; per esempio, può eseguire il prefetching dei dati nei file ad accesso sequenziale. Tuttavia, uno svantaggio chiave è che deve essere implementata una cache di file separata per ogni file, per cui il IOCS deve decidere quanta memoria riservare a ciascuna cache.



**Figura 14.23** (a) Cache di file; (b) cache del disco.

### ***Cache del disco***

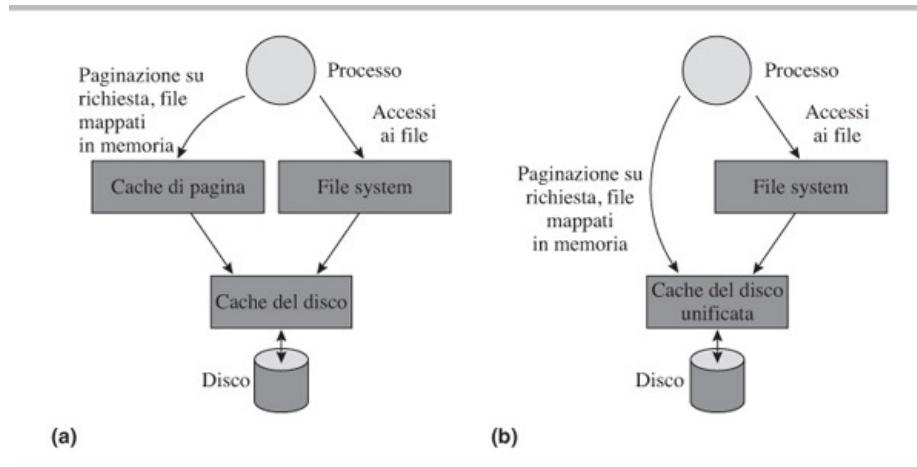
La cache del disco è implementata nel IOCS fisico o nel driver di dispositivo di un disco. Il suo scopo è di velocizzare l'accesso ai dati memorizzati sul disco. Come mostrato nella [Figura 14.23\(b\)](#), la richiesta per un'operazione di I/O viene passata allo scheduler dell'I/O solo se i dati richiesti non sono presenti nella cache del disco. Il vantaggio chiave di una cache del disco rispetto alla cache di file è che non discrimina tra i file memorizzati su un disco, per cui del suo utilizzo beneficiano tutte le elaborazioni su file. Inoltre non deve determinare la dimensione della cache individualmente per ogni file.

Tuttavia, l'hit ratio nella cache del disco è sensibile a come i processi accedono ai file. Per esempio, se un processo legge un grande file sequenziale rapidamente, i suoi dati potrebbero occupare la maggior parte dei buffer nella cache, cosa che influirebbe negativamente sull'accesso agli altri file. Le cache del disco implementate nei moderni sistemi operativi incorporano anche alcune caratteristiche delle cache di file per migliorare l'hit ratio. Dunque una cache del disco può effettuare il prefetch di alcuni blocchi in un file ad accesso sequenziale per ridurre i tempi di attesa del processo.

### 14.13 Cache del disco unificata

Oltre alla cache del disco e di file, il SO utilizza anche, implicitamente o esplicitamente, un'altra cache chiamata *cache delle pagine* nel gestore della memoria virtuale. L'uso di diverse cache può far aumentare le operazioni di copia da eseguire per accedere ai dati memorizzati sul disco. L'overhead di tempo e memoria introdotto dalle molte operazioni di copia motivano l'uso di una cache del disco unificata.

La [Figura 14.24\(a\)](#) è un diagramma che mostra l'uso della cache del disco e la cache delle pagine. La cache delle pagine contiene tutte le pagine di codice e dati dei processi presenti in memoria, incluse le pagine dei file mappati in memoria. Quando si verifica un page fault, una nuova pagina è caricata nella cache delle pagine. Poiché la dimensione della pagina è tipicamente di pochi blocchi, questa operazione implica la lettura di pochi blocchi da un programma o da un file di swap. Dunque i blocchi del disco vengono letti e copiati nella cache del disco e poi vengono copiati nella cache delle pagine. Quando una pagina modificata deve essere rimossa dalla memoria, viene prima copiata nella cache del disco. Da qui, la pagina è scritta sul disco in un momento successivo. In questo modo, sono necessarie due operazioni di copia per ogni operazione di page-in e page-out – un'operazione di copia tra il disco e la cache del disco e un'altra tra la cache del disco e la cache delle pagine. Dopo un'operazione di page-in, sono presenti due copie della pagina in memoria finché una delle copie non sarà sovrascritta. Le operazioni multiple di copia sulle pagine e le copie duplicate delle pagine causano problemi di prestazioni. Un'altra scelta difficile di progetto è la quantità di memoria da riservare per ogni cache; infatti può avere ripercussioni sulle prestazioni del sistema poiché una cache delle pagine sottodimensionata potrebbe comportare sia un ridotto livello di multiprogrammazione sia trashing, mentre una cache del disco sottodimensionata potrebbe rallentare le elaborazioni dei file a causa dei frequenti accessi al disco. L'unione delle due cache risolverebbe questi problemi: le copie e le operazioni multiple sarebbero eliminate e le porzioni di cache assegnate ai due usi potrebbero variare per adattarsi al carico del sistema.



**Figura 14.24** Caching del disco: (a) cache del disco e di file separate; (b) cache del disco unificate.

Una *cache del disco unificata* è una singola cache utilizzata sia per la paginazione che per l'I/O. La [Figura 14.24\(b\)](#) rappresenta un diagramma della cache del disco unificata. Il file system considera i file come oggetti paginati su disco. Scompon il byte di offset, passato come argomento in un'istruzione di lettura o scrittura, in un numero di pagina e un offset nella pagina. Passa il numero di pagina alla cache del disco unificata per

assicurarsi che la pagina sia caricata in memoria e utilizza l'offset nella pagina per copiare i dati tra la cache del disco unificata e lo spazio di indirizzamento di un processo. L'I/O delle pagine continua a essere gestito come nei sistemi convenzionali poiché la cache del disco unificata è di fatto una cache delle pagine.

La cache del disco unificata fu introdotta nel SO Sun 4.0. Successivamente fu implementata in Unix System 5 versione 4. Il kernel Linux 2.4 e le versioni successive utilizzano la cache del disco unificata.

## 14.14 Casi di studio

### 14.14.1 Unix

Unix supporta due tipi di dispositivi – dispositivi a *blocchi* e a *caratteri*. I dispositivi a blocchi sono dispositivi ad accesso casuale in grado di leggere o scrivere blocchi di dati, come i vari tipi di dischi, mentre i dispositivi a caratteri sono dispositivi ad accesso seriale come le tastiere, le stampanti e i mouse. Un dispositivo a blocchi può anche essere usato come un dispositivo seriale. I file in Unix sono semplicemente sequenze di caratteri, come anche i dispositivi di I/O, per cui Unix gestisce i dispositivi di I/O come file. In questo modo un dispositivo ha un nome di file, corrisponde a un elemento nella gerarchia delle directory e vi si accede utilizzando le stesse chiamate che si usano per i file, ovvero *open*, *close*, *read* e *write*.

Il IOCS di Unix si compone di due parti principali – i driver di dispositivo e un buffer cache. Questi due elementi sono descritti nei paragrafi seguenti.

#### **Driver di dispositivo**

Un driver di dispositivo in Unix è strutturato in due parti chiamate *top half* (parte superiore) e *bottom half* (parte inferiore). La parte superiore è costituita dalle routine che avviano le operazioni di I/O su un dispositivo come conseguenza di una chiamata *open*, *close*, *read* o *write* eseguita da un processo; la parte inferiore si compone del gestore degli interrupt per la classe di dispositivi gestita dal driver. In questo modo la parte superiore corrisponde ai moduli dello scheduler dell'I/O e dello starter dell'I/O di [Figura 14.13](#), mentre la parte inferiore corrisponde ai moduli del gestore del completamento dell'I/O e di gestione degli errori.

Un device driver ha un'interfaccia composta da un insieme di punti di ingresso predefiniti nelle routine del driver. Alcuni di questi sono:

1. `<ddname>_init`: routine di inizializzazione del device driver;
2. `<ddname>_read/write`: routine per leggere e scrivere un carattere;
3. `<ddname>_int`: routine di gestione degli interrupt.

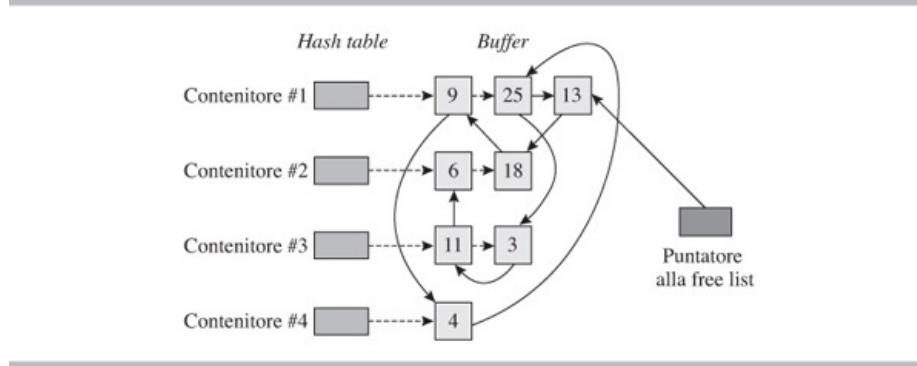
La routine `<ddname>_init` viene richiamata durante il boot del sistema. Inizializza i vari flag utilizzati dal driver. Inoltre verifica la presenza dei vari dispositivi, imposta i flag per indicarne la presenza e può allocare loro dei buffer. L'I/O dei caratteri è eseguito invocando le routine `<ddname>_read` e `<ddname>_write`. Il driver di dispositivo deve fornire una routine per il trasferimento dei blocchi di dati, che equivale approssimativamente allo scheduler dell'I/O mostrato in [Figura 14.13](#). Una chiamata a questa routine riceve come parametro l'indirizzo di un I/O control block. La routine aggiunge questo I/O control block a una IOQ e se possibile avvia l'operazione di I/O. Se l'avvio immediato non è possibile, l'operazione di I/O è fatta partire successivamente quando viene generato un interrupt di completamento dell'I/O.

#### **Buffer cache**

La buffer cache è una *cache del disco* come descritto nel Paragrafo 14.13. È organizzato come un pool di buffer, in cui ogni buffer ha la stessa dimensione di un blocco del disco. Ogni buffer ha un header che contiene tre elementi di informazione: una coppia *indirizzo del dispositivo*, *indirizzo del blocco del disco* fornisce l'indirizzo del blocco presente nel buffer, un *flag di stato* indica se l'I/O è in esecuzione per il buffer e un *flag busy* indica se qualche processo sta cercando di accedere al contenuto del buffer.

Una hash table è utilizzata per velocizzare la ricerca di un blocco del disco ([Figura 14.25](#)). La hash table si compone di un numero di contenitori e ogni contenitore punta a una lista di buffer. Quando un blocco del disco con indirizzo *aaa* viene caricato in un

buffer con indirizzo  $bbb$ ,  $aaa$  viene passato alla funzione di hashing  $h$  per calcolare un numero di contenitore  $e = h(aaa)$  nella hash table. Il buffer viene inserito nella lista dei buffer puntata dall' $e$ -esimo contenitore. In questo modo, la lista contiene tutti i buffer che contengono blocchi del disco i cui indirizzi corrispondono all' $e$ -esimo contenitore.



**Figura 14.25** Buffer cache in Unix.

La seguente procedura è utilizzata quando un processo  $P_i$  esegue un'operazione di lettura su un file  $\alpha$ .

1. Crea la coppia (*indirizzo del dispositivo, indirizzo del blocco del disco*) per il byte  $P_i$ .
2. Passa l'*indirizzo del blocco del disco* alla funzione di hashing per ottenere un numero di contenitore. Cerca tra i buffer del contenitore uno che abbia la stessa coppia nel proprio header.
3. Se questo buffer non esiste, alloca un buffer vuoto, inserisce la coppia (*indirizzo del dispositivo, indirizzo del blocco del disco*) nell'header, inserisce il buffer nella lista puntata dal contenitore appropriato, imposta il *flag di stato* a "I/O in esecuzione", accoda il buffer per l'I/O e sospende  $P_i$  in attesa del completamento dell'I/O.
4. Se esiste un buffer con la stessa coppia nell'header, restituisce il controllo a  $P_i$ , con l'indirizzo del buffer se i flag indicano che l'operazione di I/O sul buffer è completa e il buffer non è occupato. Altrimenti, mette  $P_i$  in attesa del completamento di un'operazione di lettura sul buffer o sulla condizione "non occupato" del buffer.
5. Se sono presenti buffer liberi, controlla se il prossimo blocco del disco allocato ad  $\alpha$  è già presente in un buffer. In caso contrario, alloca un buffer libero e lo accoda per un'operazione di lettura.

Questa procedura non alloca buffer ad ogni processo, per cui i processi che accedono in maniera concorrente a un file possono condividere i dati presenti in un buffer. Questa organizzazione facilita la semantica della condivisione dei file in Unix (Paragrafo 13.14.1). Allo stesso tempo, il prefetching dei dati è eseguito per ogni processo avviando un I/O per il prossimo blocco del file (passo 5), implementando il buffering per ogni processo. I benefici del blocking dei record riguardano il fatto che un intero blocco viene letto/scritto quando si accede a un byte.

I buffer nel pool di buffer vengono riutilizzati secondo la politica LRU come segue: tutti i buffer sono inseriti in una free list. Un buffer viene spostato alla fine della lista quando il suo contenuto è referenziato. In questo modo i buffer utilizzati meno recentemente si spostano verso la testa della free list. Nel passo 3, il buffer in testa alla free list è allocato a meno che non contenga dati modificati che sono già stati scritti nel blocco. In questo caso, un'operazione di scrittura per il buffer è accodata e viene allocato il prossimo buffer nella lista.

#### Esempio 14.7 - Buffer cache in Unix

La [Figura 14.25](#) illustra il buffer cache in Unix. I blocchi 9, 25 e 13 corrispondono al primo elemento della hash table; quindi vengono inseriti nella lista linkata puntata da questo elemento. In modo simile 6, 18 e 11, 3 formano le liste linkate puntate dal secondo e terzo elemento della hash table. Tutti i buffer sono anche inseriti nella free list. Se un processo accede ai dati presenti nel blocco 18, il buffer contenente il blocco 18 viene spostato alla fine della free list. Se il processo accede ai dati nel blocco 21, allora il primo buffer nella free list, ovvero il buffer contenente il blocco 13, viene allocato se il suo contenuto non è stato modificato da quando è stato caricato. Il buffer

viene aggiunto alla lista puntata dall'elemento appropriato della hash table dopo che il blocco 21 è stato caricato al suo interno. Inoltre è anche spostato alla fine della free list.

L'efficacia del buffer cache di Unix è stata studiata in maniera estensiva. Uno studio del 1989 riportò che una cache di 60 MB su un sistema HP forniva un hit ratio di 0.99 e una cache di 16 MB su un altro sistema forniva un hit ratio di 0.9. Quindi una quantità di memoria relativamente piccola dedicata al buffer cache può fornire un elevato hit ratio.

#### 14.14.2 Linux

L'organizzazione del IOCS di Linux è analoga a quella del IOCS di Unix. I dispositivi di I/O a blocchi e a caratteri sono supportati da device driver separati, i dispositivi sono gestiti come file e una buffer cache è utilizzata per velocizzare l'elaborazione dei file. Tuttavia, molte specifiche del IOCS sono differenti. Ne elenchiamo alcune prima di affrontare i dettagli della schedulazione del disco di Linux 2.6.

1. I moduli del kernel di Linux - che includono i device driver - sono caricabili dinamicamente, per cui un driver di dispositivo deve essere registrato nel kernel quando viene caricato e cancellato prima di essere rimosso dalla memoria.
2. Per i dispositivi, la struttura dati *vnode* del file system virtuale (VFS) (Paragrafo 13.13) contiene i puntatori alle funzioni specifiche del dispositivo per le operazioni *open*, *close*, *read* e *write*.
3. Ogni buffer nella cache del disco ha un header allocato in uno slab dall'allocatore di slab (Paragrafo 11.11).
4. I buffer dirty (referenziati) presenti nella cache del disco sono scritti nella cache quando questa è troppo piena, quando un buffer è stato in cache per troppo tempo, o quando viene richiesta la scrittura dei buffer per ragioni di affidabilità.

Lo scheduling dell'I/O in Linux 2.6 utilizza alcune innovazioni per migliorare le prestazioni della schedulazione. Un'operazione di lettura deve essere eseguita sul disco quando un processo effettua una chiamata *read* e i dati richiesti non sono già presenti nel buffer cache. Il processo sarebbe bloccato fino al termine dell'operazione di I/O. D'altra parte, quando un processo effettua una chiamata *write*, i dati da scrivere sono copiati in un buffer e l'effettiva operazione di scrittura avviene successivamente. Quindi il processo che esegue una *write* non viene bloccato e può continuare a eseguire altre *write*. Inoltre, per fornire tempi di risposta migliori ai processi, il IOCS esegue le operazioni di lettura con una priorità maggiore rispetto alle operazioni di scrittura.

Lo scheduler dell'I/O mantiene una lista delle operazioni di I/O pendenti e schedula le operazioni presenti in questa lista. Quando un processo effettua una *read* o una *write*, il IOCS controlla se la stessa operazione su dati vicini è pendente. Se il controllo ha esito positivo, combina la nuova operazione con l'operazione pendente, riducendo il numero di operazioni sul disco e il movimento delle testine del disco e migliorando di conseguenza il throughput del disco.

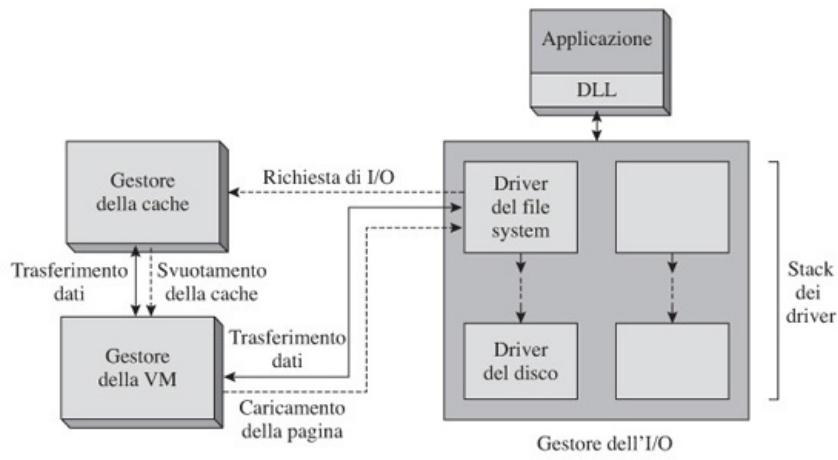
Linux 2.6 fornisce quattro scheduleri di I/O. L'amministratore di sistema può scegliere quello che meglio si adatta al carico in una specifica installazione. Lo scheduler *no-op* è semplicemente uno scheduler FCFS. Lo scheduler *deadline* utilizza lo scheduling Look come base ma incorpora anche una caratteristica per evitare lunghi ritardi. Implementa lo scheduling Look mantenendo una lista di schedulazione delle richieste ordinate per numero di traccia e selezionando una richiesta in base alla posizione corrente della testina. Tuttavia, lo scheduling Look presenta un problema quando un processo esegue un gran numero di scritture in una parte del disco - le operazioni di I/O nelle altre parti del disco sarebbero ritardate. Se un'operazione di lettura fosse ritardata, questo causerebbe ritardi considerevoli nel processo che l'aveva richiesta. Per prevenire questi ritardi, lo scheduler assegna una dedaline di 0.5 secondi per un'operazione di lettura e una deadline di 5 secondi per un'operazione di scrittura e mantiene due code - una per le richieste di lettura e una per quelle di scrittura - in base alle deadline. Normalmente schedula le richieste dalla lista di schedulazione; tuttavia, se scade la deadline di una richiesta in testa alla lista delle richieste di lettura o scrittura, schedula questa richiesta e un paio di altre richieste dalla stessa lista fuori dalla normale sequenza prima di riprendere la normale schedulazione. Lo scheduler con *accodamento* *completamente equo* mantiene una coda separata delle richieste di I/O per ogni processo ed esegue il round robin tra queste code. Questo approccio evita lunghe attese per i

processi.

Un processo che esegue l'I/O sincrono viene bloccato finché le operazioni di I/O non vengono completate. Questo processo esegue la prossima operazione di I/O immediatamente dopo essere stato risvegliato. Quando si utilizza lo scheduling Look, le testine del disco molto probabilmente sono già passate sulla traccia che contiene i dati necessari per la prossima operazione di I/O, per cui la prossima operazione di I/O del processo sarebbe servita solo durante la prossima scansione del disco. Questo causa ritardi nel processo e può causare un numero maggiore di movimenti delle testine del disco. Lo scheduler *anticipatore* affronta questo problema. Dopo aver completato un'operazione di I/O, attende pochi millisecondi prima di eseguire la prossima operazione di I/O. In questo modo, se il processo attivato al termine dell'operazione di I/O precedente eseguisse un'altra operazione di I/O nelle vicinanze dell'operazione precedente, questa operazione potrebbe essere servita durante la stessa scansione del disco.

### 14.14.3 Windows

Lo schema di [Figura 14.26](#) mostra l'organizzazione per l'elaborazione dei file utilizzata in Windows. Il *gestore della cache* esegue il *caching dei file*. Il *gestore dell'I/O* fornisce i servizi generici utilizzati per implementare le operazioni di I/O specifiche del subsystem mediante l'uso dei driver di dispositivo; inoltre esegue la gestione dei buffer di I/O. Come descritto nel Paragrafo 4.8.4, le DLL linkate alle applicazioni richiamano le funzioni nel gestore dell'I/O per ottenere uno specifico servizio del subsystem. Il *gestore della VM* è stato descritto nel Paragrafo 12.8.4.



**Figura 14.26** Elaborazione dei file in Windows.

La cache dei file è organizzata come un insieme di blocchi di cache, ognuno con dimensione di 256 KB. La parte di un file mantenuta in un blocco di cache è chiamata *vista*. Un *virtual address control block* (VACB) descrive ogni vista; contiene l'indirizzo virtuale associato alla vista, l'offset del suo primo byte nel file e il numero di operazioni di lettura e scrittura che stanno accedendo alla vista. La presenza dell'indirizzo virtuale e delle informazioni sull'offset del file nel VACB risulta utile nella semantica della condivisione dei file - assicura che i processi che eseguono accessi concorrenti a un file vedano il risultato dell'ultimo aggiornamento senza tener conto del fatto che il file sia mappato in memoria o vi si abbia accesso direttamente. Il gestore della cache imposta un array indice di VACB per un file al momento dell'apertura. Per un file sequenziale, l'array indice conterrebbe solo un puntatore al VACB relativo all'offset corrente nel file. Per un file ad accesso casuale, l'array index di VACB conterrebbe i puntatori ai VACB relativi ai recenti accessi al file.

Un'operazione di I/O è eseguita da un *device driver stratificato*. È rappresentato come un lista linkata di device driver chiamata *driver stack*. Quando un thread richiede un'operazione di I/O, il gestore dell'I/O costruisce un *pacchetto di richiesta dell'I/O* (*I/O request packet* - IRP) e lo passa al primo device driver nello stack driver appropriato. Il device driver può eseguire l'intera operazione di I/O, scrivere un codice di stato nell'IRP

e restituirlo al gestore dell'I/O. In alternativa, può decidere che siano necessarie operazioni addizionali per completare l'operazione di I/O, scrive le informazioni nell'IRP e passa l'IRP al prossimo device driver nello stack e così via, finché non viene implementata tutta l'operazione di I/O. Questo modello consente di aggiungere nuovi driver di dispositivo in modo da fornire nuove funzionalità nel sottosistema di I/O. Per esempio, un device driver potrebbe essere aggiunto tra il driver del file system, che affronteremo in seguito, e il driver del disco per realizzare il mirroring del disco. Tali driver sono chiamati *driver filtro*. Driver quali i driver del disco sono chiamati *driver funzione*. Contengono le funzionalità per l'inizializzazione, lo scheduling e l'avvio delle operazioni di I/O, la gestione degli interrupt, e l'aggiunta dinamica di nuovi dispositivi per facilitare la funzionalità plug-and-play.

Un file system è anche implementato come un *driver di file system* (FSD). Questo richiama altri driver che implementano le funzionalità del metodo di accesso e del driver di dispositivo. Questa organizzazione consente a molti file system di coesistere nel sistema host. Il gestore dell'I/O in questo modo fornisce le funzionalità di un file system virtuale (Paragrafo 13.13). Quando un sottosistema DLL linkato a un thread richiede un'operazione sul file, il gestore dell'I/O richiama un driver di file system appropriato per gestire la richiesta. La richiesta tipicamente contiene l'offset del byte del file coinvolto nell'operazione. Il driver del file system consulta la file map table del file, che è accessibile dal record base del file nella master file table (MFT) per convertire l'offset del byte nel file in un offset del byte in un blocco di dati su un dispositivo e richiama il relativo driver di dispositivo. Se il dispositivo è un RAID, il driver di dispositivo è un *volume manager*, che gestisce il RAID. Il driver converte l'offset del byte nel blocco in uno o più unità contenenti un numero di disco, un numero di settore e un offset di byte e passa la richiesta al driver del disco. Windows supporta i *volumi striped*, ovvero i sistemi RAID di livello 0, i *volumi mirrored*, ovvero i sistemi RAID di livello 1 e i sistemi RAID di livello 5. In maniera analoga supporta i *volumi spanned* descritti nel Paragrafo 13.14.4.

Quando un thread effettua una richiesta di lettura in un file, il gestore dell'I/O passa questa richiesta al driver del file system, che la passa al gestore della cache. Il gestore della cache consulta l'array indice del VACB per il file e determina se i byte richiesti del file sono presenti in qualche vista nella cache. In caso negativo, alloca un blocco di cache, crea una vista relativa ai byte richiesti del file nel blocco di cache e costruisce un VACB. Questa operazione coinvolge la lettura della parte rilevante del file nel blocco di cache. Il gestore della cache copia i dati richiesti dal blocco di cache nello spazio di indirizzamento del chiamante. Le azioni inverse vengono eseguite nel caso di una richiesta di scrittura. Se si verifica un page fault durante la copia dei dati dallo o nello spazio di indirizzamento del chiamante, il gestore della memoria virtuale richiama il driver del disco attraverso il file system per portare la pagina richiesta in memoria. Questa operazione non coinvolge la cache. In questo modo, un file system deve supportare l'I/O dei file sia con la cache che senza cache. Per facilitare la manipolazione efficiente dei metadati, il driver del file system utilizza le operazioni di lettura/scrittura a livello kernel, che accedono ai dati direttamente in cache invece di copiarli prima dallo (o nello) spazio di indirizzamento logico del driver del file system.

Il gestore della cache mantiene le informazioni sulle ultime richieste di I/O su un file. Se riesce a ricavarne un pattern, come l'accesso sequenziale al file, effettua il prefetching dei blocchi seguenti in base a questo pattern. Inoltre accetta *suggerimenti* dalle applicazioni riguardanti la natura delle attività di elaborazione dei file e li utilizza per lo stesso scopo. I dati da scrivere in un file rispecchiano la vista del file mantenuto nel gestore della cache. Una volta al secondo, il *lazy writer*, un thread di sistema creato dal gestore della cache, accoda un quarto delle pagine modificate presenti nella cache per la scrittura su disco e ordina al gestore della memoria virtuale di scrivere i dati.

Si ricordi che un SO tipicamente rileva i dispositivi connessi al momento del boot e costruisce le strutture dati di conseguenza. Questa organizzazione è restrittiva, poiché richiede il riavvio del sistema quando si connette o si disconnette un dispositivo dal sistema. Windows supporta la funzionalità plug-and-play (PnP) che consente di connettere e disconnettere i dispositivi dinamicamente. Questa funzionalità è realizzata coordinando il funzionamento dell'hardware di I/O, il sistema operativo e i device driver. L'hardware coopera con il software di boot per costruire l'elenco iniziale dei dispositivi connessi al sistema e inoltre si coordina con il gestore del PnP quando si connettono o disconnettono nuovi dispositivi. Il gestore del PnP, se necessario, carica un driver di dispositivo per un nuovo dispositivo, determina le risorse come i numeri di interrupt che possono essere richieste per il suo funzionamento e assicura l'assenza di conflitti assegnando o riassegnando le risorse richieste. A questo punto inizializza il device driver

per il nuovo dispositivo e reinizializza ogni altro dispositivo coinvolto.

Windows Vista ha una nuova caratteristica che risolve un insieme di problemi dello scheduling del disco: lo scheduling del disco tratta tutte le operazioni di I/O in maniera uniforme cercando di migliorare il throughput. In questo modo, occasionalmente le operazioni di I/O dei processi a bassa priorità possono essere favorite rispetto alle altre operazioni di I/O, cosa che causerebbe ritardi nei processi ad alta priorità e degraderebbe la risposta delle applicazioni che li hanno creati. La funzionalità di Vista chiamata *I/O a priorità* fornisce un metodo per ottenere il corretto bilanciamento tra throughput e risposta. Utilizzando questa funzionalità, un'applicazione può specificare una priorità per le proprie operazioni di I/O. Può effettuare una chiamata di sistema per abbassare la propria priorità a *background* in modo che le operazioni di I/O non abbiano una priorità superiore a quella dei processi non in *background* e può ritornare alla priorità originaria utilizzando un'altra chiamata di sistema quando desiderato. Un device driver controlla la priorità del processo che ha lanciato l'operazione di I/O e imposta i flag in un IRP per indicare se l'operazione di I/O debba essere trattata con una priorità inferiore.

## Riepilogo

Durante l'attività di elaborazione di un file, il file system implementa la condivisione e la protezione dei file, mentre l'*input output control system* (IOCS) di fatto implementa le operazioni sul file. Il IOCS è strutturato in due livelli chiamati *metodi di accesso* e *IOCS fisico* che garantiscono, rispettivamente, le buone prestazioni dell'attività di elaborazione di un file e un buon throughput dei dispositivi di I/O. In questo capitolo, abbiamo studiato le tecniche adottate dai metodi di accesso e dal IOCS fisico.

Buoni valori di throughput di un dispositivo di I/O sono ottenuti mediante azioni congiunte del dispositivo di I/O e del IOCS. Il dispositivo di I/O è progettato in modo affidabile e le operazioni di I/O hanno *tempi di accesso* brevi – ovvero il tempo necessario per posizionare il supporto di I/O o le testine di lettura/scrittura prima del tempo di trasferimento – e ottengono alti tassi di trasferimento. Le tecniche di *riorganizzazione dei dati*, *disk attachment technologies* e *redundant arrays of inexpensive disks* (RAID) sono rilevanti in questo contesto.

Anche con accessi veloci e tassi di trasferimento elevati dei dispositivi di I/O, un processo che esegue un'operazione di I/O incorre in considerevoli *tempi di attesa* finché non viene completata l'operazione di I/O. Il IOCS fisico fornisce due funzionalità di base per migliorare le prestazioni del sistema: blocca un processo finché non viene completata l'operazione di I/O, consentendo al kernel di commutare la CPU a un altro processo. Utilizza lo *scheduling del disco* per eseguire le operazioni di I/O dirette al disco in un ordine tale da ridurre il movimento delle testine di lettura/scrittura del disco e incrementare di conseguenza il throughput del disco e ridurre il tempo medio di attesa delle operazioni di I/O.

Un *metodo di accesso* migliora le prestazioni dell'attività di elaborazione di un file all'interno di un processo mediante tecniche di *buffering* e *blocking* dei record. Un *buffer* è un'area di memoria utilizzata per mantenere temporaneamente i dati che devono essere letti da un dispositivo o che devono essere scritti su di esso. Per un file di input, la tecnica del buffering cerca di effettuare il prefetch dei dati in modo che siano disponibili per un processo senza dover effettuare un'operazione di I/O, cosa che riduce o elimina il tempo di attesa. Per un file di output, copia i dati nel buffer e consente al processo di continuare la sua esecuzione; l'effettiva scrittura viene eseguita successivamente. La tecnica del blocking legge più dati da un dispositivo in una singola operazione di I/O rispetto a quelli richiesti da un processo; riduce il numero di operazioni di I/O da eseguire.

*Caching* è la tecnica di mantenere alcuni file in memoria, in modo che vi si possa accedere senza dover eseguire un'operazione di I/O. Il caching riduce il numero di operazioni di I/O necessarie per accedere ai dati memorizzati in un file, migliorando le prestazioni delle attività di elaborazione dei file nei processi e migliorando inoltre le prestazioni del sistema. Il IOCS fisico implementa una *cache del disco* per ridurre il numero di operazioni di I/O per accedere ai file memorizzati sul disco. Un metodo di accesso implementa una *cache di file* per ridurre il numero di operazioni di I/O eseguite durante l'elaborazione di un file.

Il gestore della memoria virtuale utilizza anche una cache chiamata *cache delle*

*pagine*, che contiene le pagine dei processi, per migliorare le prestazioni della memoria virtuale. Tuttavia, dal momento che le aree di swap dei processi sono implementate su un disco, l'uso della cache delle pagine e della cache del disco implica la copia delle pagine tra due cache, cosa che consuma tempo di CPU e consuma memoria a causa delle copie multiple di una pagina. I sistemi operativi quindi utilizzano una *cache del disco unificata* per ridurre le copie ed eliminare la necessità delle copie multiple delle pagine.

## Domande

- 14.1. Classificare ognuna delle seguenti affermazioni come vera o falsa.
- Quando sono utilizzati i bit di parità per memorizzare/leggere in maniera affidabile i dati, un codice di correzione degli errori richiede più bit di parità rispetto a un codice di rilevamento degli errori.
  - Ridurre l'allocazione dello spazio su disco per un file a un *gruppo di cilindri* – ovvero un gruppo di cilindri consecutivi su un disco – riduce i movimenti della testina del disco durante l'elaborazione di un file.
  - Il RAID livello 4, che utilizza la parità a blocchi alternati, fornisce parallelismo tra brevi operazioni di I/O.
  - Il blocking dei record velocizza l'elaborazione dei file sequenziali.
  - Il buffering dei record velocizza l'elaborazione dei file ad accesso diretto.
  - La politica di schedulazione del disco SCAN genera starvation.
  - Il IOCS fisico fornisce un metodo per evitare la condizione di attesa attiva mentre un processo attende il completamento dell'operazione di I/O.
  - Se  $t_x < t_p$ , è possibile ridurre  $t_w$  a  $t_c$  mediante appropriate azioni di buffering e blocking.
  - L'utilizzo di un fattore di blocking  $m$  riduce il tempo effettivo di I/O per record logico di un fattore  $m$ .
- 14.2. Selezionare l'alternativa corretta in ognuna delle seguenti domande.
- Una cache del disco è utilizzata per:
    - ridurre il numero di operazioni di I/O su un disco;
    - incrementare la capacità di un disco;
    - nessuna delle precedenti.
  - Le tecniche di riorganizzazione dei dati sono usate per:
    - ridurre il numero di operazioni sul disco durante l'elaborazione di un file;
    - ridurre i movimenti della testina del disco tra i blocchi con indirizzi adiacenti;
    - ridurre i ritardi rotazionali durante l'accesso ai blocchi con indirizzi adiacenti;
    - migliorare l'efficacia del buffering e del blocking dei record.
  - Scheduling del disco:
    - riduce il numero di operazioni di I/O eseguite su un disco;
    - riduce il movimento medio della testina del disco per ogni operazione di I/O;
    - mira a velocizzare l'elaborazione di un file.
  - Un programma esegue un'istruzione di lettura su un file *alpha* 100 volte durante la sua esecuzione; tuttavia solo 50 operazioni di I/O sono effettivamente eseguite per leggere dati dal file *alpha*. Questo è possibile se:
    - il metodo di accesso utilizzato per il file *alpha* usa il buffering senza il blocking;
    - il metodo di accesso non adotta il blocking e il IOCS fisico non adotta una cache del disco;
    - o il metodo di accesso utilizza il blocking o il IOCS fisico utilizza una cache del disco.

## Problemi

- 14.1. Spiegare come (e se) il buffering e il blocking dei record ha dei benefici per i seguenti tipi di file.
- Un file ad accesso sequenziale.
  - Un file ad accesso sequenziale indicizzato.
  - Un file ad accesso diretto.
- 14.2. Un file di *aggiornamento* è un file letto e modificato durante l'elaborazione – un programma legge un record, lo modifica in memoria e lo riscrive nel file.

- a. Quale dispositivo di I/O si adatta meglio per memorizzare un file di aggiornamento?  
 b. Il buffering e il blocking dei record sono utili per un file di aggiornamento?  
 Giustificare le risposte.
- 14.3. Descrivere come il throughput di un disco può essere ottimizzato in un file system che adotta l'allocazione non contigua dei blocchi dei file. (*Suggerimento:* considerare l'organizzazione dei blocchi nella free list, nella riorganizzazione dei dati e nei gruppi di cilindri.)
- 14.4. Un disco ha le seguenti caratteristiche:
- |                        |             |
|------------------------|-------------|
| tempo di 1 rivoluzione | = 8 ms      |
| $t_{\text{sect}}$      | = 1 ms      |
| $t_{\text{st}}$        | = 3 ms      |
| dimensione del settore | = 1024 byte |
- Mostrare il grafico del throughput di picco del disco rispetto al fattore di interposizione dei settori ( $F_{\text{int}}$ ).
- 14.5. Commentare l'efficacia di (a) cache del disco e (b) un RAM disk per velocizzare l'elaborazione dei file ad accesso sequenziale e ad accesso diretto.
- 14.6. Le richieste di operazioni di I/O sulle seguenti tracce sono pendenti al tempo  $t = 160$  ms.
- 7, 45, 98, 70, 68, 180
- Se le richieste sono effettuate in quest'ordine, costruire una tabella analoga alla [Figura 14.16](#) per il disco dell'Esempio 14.3.
- 14.7. Un *disco biased* è un ipotetico disco il cui tempo di ricerca per la traccia  $n$  è una funzione lineare in  $n$  (per esempio, tempo di ricerca =  $0.1 \times n$ ).  $\{\text{seg}_i\}$  è l'insieme delle operazioni di I/O richieste in un certo periodo di tempo. L'ordine in cui vengono schedulate le operazioni di I/O su un disco biased dall'algoritmo SSTF è identico all'ordine in cui sarebbero schedulate le stesse operazioni di I/O dall'algoritmo SCAN sul disco convenzionale dell'Esempio 14.3?
- 14.8. Un processo elabora un file ad accesso sequenziale. I tempi di I/O e di elaborazione per ogni record nel file sono i seguenti:
- |                                        |                              |
|----------------------------------------|------------------------------|
| tempo di accesso del dispositivo       | = 10 ms                      |
| tempo di trasferimento per ogni record | = 6ms                        |
| massimo numero di record               | = 5 record richiesti insieme |
| tempo di elaborazione per ogni record  | = 10 ms                      |
- a. Se sono utilizzati due buffer, trovare il valore del più piccolo fattore di blocking che può minimizzare il tempo di attesa per ogni record.  
 b. Se sono utilizzati due buffer e un fattore di blocking pari a 5, qual è il minimo numero di record presenti in memoria in ogni istante? (Assumere che un processo inizi un'operazione di I/O su un buffer dopo aver elaborato l'ultimo record al suo interno - [Figura 14.20](#).)
- 14.9. Un file sequenziale è memorizzato usando il blocking. Un processo lo elabora utilizzando due buffer. I tempi di I/O ed elaborazione sono i seguenti:
- |                                        |            |
|----------------------------------------|------------|
| tempo di accesso (medio)               | = 20 ms    |
| tempo di trasferimento per ogni record | = 5 ms     |
| massimo numero di record               | = 5 record |
| richiesti insieme                      |            |
| tempo di elaborazione per ogni record  | = 10 ms    |
- Determinare i valori ottimi del fattore di blocking e del numero di buffer. Quali modifiche, se ve ne sono, potrebbero essere effettuate se il massimo numero di record, che il processo verosimilmente richiederebbe insieme, contemporaneamente (i) 3 record, (ii) 8 record? (*Suggerimento:* Esempio 14.4.)
- 14.10. Un buffer è utilizzato nell'elaborazione del file *info* del Problema 13.6. Calcolare il tempo di esecuzione del processo se il tempo di copia per ogni record è di 0.5 ms. Spiegare il calcolo.
- 14.11. Classificare la seguente affermazione come vera o falsa: "Con un'attenta scelta del

fattore di blocking e del numero di buffer, è sempre possibile ridurre il tempo di attesa  $t_c$ ."

- 14.12. Un processo apre un file prima di accedervi. Se tenta di accedere al file senza aprirlo, il file system effettua una open prima di effettuare l'accesso. Un manuale per programmati di sistema avverte tutti i programmati di aprire il file prima di accedervi o pagare una penalizzazione delle prestazioni. Spiegare la natura e la causa della penalizzazione delle prestazioni.
- 14.13. Come i differenti algoritmi di scheduling del disco influenzano l'efficacia del buffering dell'I/O?
- 14.14. Un processo elabora un file di input utilizzando molti buffer. Quali delle seguenti affermazioni sono accurate? Spiegare il ragionamento.
  - a. "Di tutti gli algoritmi di scheduling, FCFS è quello che più verosimilmente fornisce le migliori prestazioni di tempo trascorso per il processo."
  - b. "La riorganizzazione dei dati è efficiente solo durante la lettura dei primi record nel file; non è efficiente durante la lettura degli altri record del file."
- 14.15. Un nastro magnetico ha una densità di memorizzazione di 80 bit/cm lungo una traccia. Il nastro si muove alla velocità di 2 metri al secondo durante la lettura/scrittura dei dati. Il gap inter record è di 0.5 cm e il tempo di accesso del nastro è di 5 ms. Un file sequenziale contenente 5000 record, ognuno con dimensione di 400 byte, è memorizzato su questo nastro magnetico. Calcolare la lunghezza del nastro magnetico occupata dal file e il tempo totale di I/O richiesto per leggere il file se il file è memorizzato (a) senza blocking e (b) con fattore di blocking pari a 4.
- 14.16. Un processo utilizza molti buffer durante l'elaborazione di un file contenente record a blocchi. Si verifica un malfunzionamento del sistema durante il funzionamento. È possibile riprendere l'esecuzione del processo dal punto del malfunzionamento?
- 14.17. Il *fattore di speedup* risultante dall'utilizzo di una speciale tecnica di I/O è il rapporto del tempo trascorso di un processo senza blocking o buffering dei record rispetto al tempo trascorso dallo stesso processo con la speciale tecnica di I/O. Nel Paragrafo 14.9, si mostrava che il fattore di speedup dovuto al buffering aveva un limite superiore di 2. Sviluppare una formula per il fattore di speedup quando un processo non utilizza buffer durante l'elaborazione di un file contenente record a blocchi. Il valore del fattore di speedup può eccedere il valore 2? In caso affermativo, fornire un esempio.
- 14.18. Sviluppare una formula per calcolare il *fattore di speedup* quando un processo utilizza due buffer per l'elaborazione di un file contenente blocchi di record e  $t_p \geq t_x$ .
- 14.19. Descrivere le implicazioni sull'affidabilità del file system di una cache dei file o del disco. Unix supporta una system call *flush()* che forza il kernel a scrivere l'output bufferizzato sul disco. Un programmatore può utilizzare la chiamata *flush()* per migliorare l'affidabilità dei propri file?
- 14.20. Il comando Unix *lseek* indica l'offset del prossimo byte da leggere o scrivere in un file ad accesso sequenziale. Quando un processo vuole eseguire un'operazione di lettura o scrittura, esegue un comando *lseek*. Questo comando è seguito da un comando di lettura o scrittura.
  - a. Quali sono i vantaggi nell'utilizzare il comando *lseek*?
  - b. Qual è la sequenza di azioni che il file system o il IOCS dovrebbero eseguire quando un processo lancia un comando *lseek*?
- 14.21. Mostrare che la divisione di un polinomio binario costituito da  $n_d + n_c$  bit in un record, dove  $n_d$  è il numero di bit di dati e  $n_c$  è il numero di bit del CRC, per il polinomio CRC ha resto 0. (*Suggerimento*: un termine  $x^i$ ,  $i = 1, \dots, n_d - 1$ , nel polinomio per  $n_d$  bit di dati è il termine  $x^{i+n_c}$  nel polinomio per gli  $n_d + n_c$  bit nel record. Inoltre si noti che l'addizione e la sottrazione modulo 2 producono lo stesso risultato.)

## Problemi avanzati

- 14.1. Sia dato un disco con 200 tracce, ognuna di 100 blocchi, e velocità di ricerca pari a una traccia per ms. All'istante  $t = 0$  il sistema operativo sta servendo una richiesta sulla traccia 35 e in coda ci sono richieste di lettura/scrittura di blocchi  $b$  per le tracce 15 ( $b = 3$ ), 58 ( $b = 1$ ), 15 ( $b = 2$ ), 32 ( $b = 4$ ), 78 ( $b = 5$ ) e 55 ( $b = 1$ ).

Successivamente arrivano altre richieste all'istante  $t = 7$  per la traccia 89 ( $b = 2$ ) e all'istante  $t = 23$  per la traccia 48 ( $b = 4$ ). Si calcoli il tempo medio di accesso per servire tutte le richieste nel caso in cui il tempo di latenza sia 2 ms per le politiche a) SSTF, b) SCAN, c) C-LOOK.

## Note bibliografiche

Tanenbaum (1990) descrive le componenti hardware di I/O. Ruemmler e Wilkes (1994) presentano un modello di driver del disco che può essere usato per l'analisi delle prestazioni e per il tuning. Teorey e Pinkerton (1972) e Hofri (1980) confrontano vari algoritmi di scheduling del disco, mentre Worthington et al. (1994) discutono di scheduling del disco per i dischi moderni. Lumb et al. (2000) discutono di come le attività in background come la riorganizzazione del disco possa essere eseguita durante il posizionamento meccanico delle testine del disco per il servizio delle attività di foreground e dell'effetto degli algoritmi di scheduling del disco sull'affidabilità di questo approccio.

Chen e Patterson (1990) e Chen et al. (1994) descrivono l'organizzazione RAID, mentre Wilkes et al. (1996) e Yu et al. (2000) discutono dei miglioramenti ai sistemi RAID. Alvarez et al. (1996) discutono di come i malfunzionamenti multipli possano essere gestiti in un'architettura RAID, mentre Chau e Fu (2000) descrivono un nuovo metodo per distribuire equamente le informazioni di parità per i RAID declusterizzati. Gibson et al. (1997) discutono dei file server per i dischi di rete. Nagle et al. (1999) trattano l'integrazione delle reti a livello utente con lo storage di rete (NAS). Curtis Preston (2002) discute del NAS e delle storage area network (SAN), mentre Clark (2003) si dedica alla tecnologia SAN. Toigo (2000) descrive i dischi moderni e le tecnologie future di memorizzazione.

Il caching del disco è trattato da Smith (1985). Braunstein et al. (1989) spiegano come velocizzare gli accessi ai file quando si utilizza l'hardware della memoria virtuale per la ricerca nella buffer cache dei file.

McKusick et al. (1996) descrivono il Berkeley fast file system per Unix 4.4BSD. Bach (1986) e Vahalia (1996) descrivono altri file system per Unix. Ruemmler e Wilkes (1993) presentano degli studi sulle prestazioni riguardanti varie caratteristiche degli accessi al disco fatti nel file system di Unix. Beck et al. (2002) e Bovet e Cesati (2005) trattano gli scheduler dell'I/O di Linux. Love (2004, 2005) descrive lo scheduler dell'I/O in Linux 2.6. Custer (1994) descrive il file system di Windows NT, mentre Russinovich e Solomon (2005) descrivono il file system NTFS per Windows.

1. Alvarez, G.A., W.A. Burkhard, F. Cristian (1996): "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering," *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 62-72.
2. Bach, M.J. (1986): *The design of the Unix operating system*, Prentice-Hall, Englewood Cliffs, N.J.
3. Beck, M., H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroter, and D. Verworner (2002): *Linux Kernel Programming*, Pearson Education, New York.
4. Bovet, D.P., and M. Cesati (2005): *Understanding the Linux Kernel*, 3rd ed., O'Reilly, Sebastopol, Calif.
5. Braunstein, A., M. Riley, and J. Wilkes (1989): "Improving the efficiency of Unix buffer caches," *ACM Symposium on SO Principles*, 71-82.
6. Chau, A., and A.W. Fu (2000): "A gracefully degradable declustered RAID architecture with near optimal maximal read and write parallelism," *Cluster Computing*, 5 (1), 97-105.
7. Chen, P.M., and D. Patterson (1990): "Maximizing performance in a striped disk array," *Proceedings of 17th Annual International Symposium on Computer Architecture, May 1990*.
8. Chen, P.M., E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson (1994): "RAID - high performance, reliable secondary storage," *Computing Surveys*, 26 (2), 145-186.
9. Clark, T. (2003): *Designing Storage Area Networks: A Practical Reference for Implementing Fibre Channel and IP SANS*, 2nd ed., Addison Wesley Professional.
10. Curtis Preston, W. (2002): *Using SANs and NAS*, O'Reilly, Sebastopol, Calif.
11. Custer, H. (1994): *Inside the Windows NT File System*, Microsoft Press, Redmond, Wash.

12. Gibson, G.A., D. Nagle, K. Amiri, F.W. Chang, E.M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka (1997): "File server scaling with network-attached secure disks," *Measurement and Modeling of Computer Systems*, 272-284.
13. Hofri, M. (1980): "Disk scheduling: FCFS vs. SSTF revisited," *Communications of the ACM*, **23** (11), 645-53.
14. Iyer, S., and P. Druschel (2001): "Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O," *Proceedings of the 18th ACM Symposium on Operating Systems Principles*.
15. Lampson, B. (1981): "Atomic transactions," in *Distributed Systems - Architecture and Implementation: An Advanced Course*, Goos, G. and J. Hartmanis (eds.), Springer Verlag, Berlin, 246-265.
16. Love, R. (2004): "I/O schedulers," *Linux Journal*, **118**
17. Love, R. (2005): *Linux Kernel Development*, 2nd ed., Novell Press.
18. Lumb, C.R., J. Schindler, G.R. Ganger, and D.F. Nagle (2000): "Towards higher disk head utilization: extracting free bandwidth from busy disk drives," *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*.
19. McKusick, M.K., K. Bostic, M.J. Karels, and J.S. Quarterman (1996): *The Design and Implementation of the 4.4BSD Operating System*, Addison Wesley, Reading, Mass.
20. Nagle D., G. Ganger, J. Butler, G. Gibson, and C. Sabol (1999): "Network support for network-attached storage," *Proceedings of Hot Interconnects*.
21. Ruemmler, C., and J. Wilkes (1993): "Unix disk access patterns," *Proceedings of the Winter 1993 USENIX Conference*, 405-420.
22. Ruemmler, C., and J. Wilkes (1994): "An introduction to disk drive modeling," *IEEE Computer*, **27** (3), 17-29.
23. Russinovich, M. E., and D. A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
24. Smith, A.J. (1985): "Disk cache-miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, **3** (3), 161-203.
25. Tanenbaum, A.S. (1990): *Structured Computer Organization*, 3rd ed., Prentice Hall, Englewood Cliffs, N.J.
26. Teorey, T.J., and T.B. Pinkerton (1972): "A comparative analysis of disk scheduling policies," *Communications of the ACM*, **15** (3), 177-184.
27. Toigo, J. (2000): "Avoiding a data crunch," *Scientific American*, **282** (5), 58-74.
28. Vahalia, U. (1996): *Unix Internals - The New Frontiers*, Prentice Hall, Englewood Cliffs, N.J.
29. Wilkes, J., R. Golding, C. Staelin, and T. Sullivan (1996): "The HP autoRAID hierarchical storage system," *ACM Transactions on Computer Systems*, **14 (1)**, 108-136.
30. Worthington, B.L., G.R. Ganger, and Y.N. Patt (1994): "Scheduling algorithms for modern disk drives," *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 241-251.
31. Yu, X., B. Gum, Y. Chen, R.Y. Wang, K. Li, A. Krishnamurthy, and T.E. Anderson (2000): "Trading capacity for performance in a disk array," *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*, 243-258.

---

# CAPITOLO 15

## Sicurezza e protezione

---

### Obiettivi di apprendimento

- Problematiche di sicurezza e protezione
- Minacce alla sicurezza
- Tecniche di cifratura
- Modelli di protezione: access control matrix, access control list, capability list, dominio di protezione
- Sicurezza e protezione dei sistemi operativi: MULTICS, Unix, Linux e Windows

I sistemi operativi adottano misure di sicurezza e protezione per evitare che un utente non autorizzato possa utilizzare, illegalmente, le risorse di un sistema di elaborazione, o interferire con esse in qualsiasi modo. Queste misure assicurano che solo gli utenti autorizzati possano accedere ai dati e ai programmi e soltanto nel modo consentito e che dati e programmi non siano né alterati, né negati agli utenti autorizzati. Le misure di *sicurezza* riguardano le minacce provenienti dall'esterno di un sistema di elaborazione, mentre le misure di *protezione* si occupano delle minacce interne.

Il principale strumento di sicurezza sono le password. La richiesta di password ostacola i tentativi, da parte di utenti non autorizzati, di fingersi l'entità legittima. La confidenzialità delle password è mantenuta tramite la *cifratura*.

Gli utenti dei sistemi di elaborazione hanno bisogno di condividere con i propri collaboratori dati e programmi memorizzati nei file; è in questo ambito che operano le misure di *protezione* del sistema operativo. Il proprietario di un file comunica al SO gli specifici *privilegi di accesso* da concedere agli altri utenti - se e come potranno accedere al file. La funzione di protezione del sistema operativo assicura, allora, che tutti gli accessi al file avverranno, rigorosamente, secondo i privilegi di accesso specificati.

Cominciamo discutendo quali tipi di violazione della sicurezza sono compiuti da *cavalli di Troia*, *virus*, *worm* e *buffer overflow*. La descrizione sarà seguita da una dissertazione sulle tecniche di cifratura. Descriviamo, poi, le tre strutture di protezione più note come *liste di controllo degli accessi* (*access control lists*), *capability lists*, e i *domini di protezione* (*protection domains*); successivamente esamineremo il grado di controllo da esse fornito nella condivisione dei file. In ultimo, spiegheremo come le classificazioni di sicurezza dei sistemi di elaborazione riflettono il grado di resistenza di un sistema alle minacce alla sicurezza e alla protezione.

### 15.1 Sicurezza e protezione: introduzione

Assicurare la non interferenza con le elaborazioni e le risorse degli utenti è uno dei tre obiettivi fondamentali di un SO menzionati nel Paragrafo 1.2. Una risorsa potrebbe essere: una risorsa hardware come un dispositivo di I/O, una risorsa software, come un programma o dati memorizzati in un file, oppure un servizio offerto dal SO. Si possono verificare diversi tipi di interferenza durante il funzionamento di un sistema di elaborazione; chiameremo ognuna di esse *minaccia*. Alcune minacce dipendono dalla natura delle specifiche risorse o servizi e dal modo in cui sono utilizzate, mentre altre sono di natura generica.

Un accesso non autorizzato alle risorse è un'ovvia minaccia al SO. Utenti non registrati nel sistema possono tentare di accedere alle sue risorse, oppure utenti registrati possono provare ad accedere a risorse per le quali non hanno l'autorizzazione. Tali utenti possono, maliziosamente, corrompere o distruggere una risorsa. Questa è una grave minaccia per i programmi e per i dati memorizzati nei file. Una minaccia meno ovvia consiste nell'impedire l'accesso legittimo, da parte degli utenti, alle risorse o ai servizi. L'obiettivo è quello di interrompere le attività degli utenti impedendo l'uso delle risorse e dei servizi messi a disposizione dal SO. Questo tipo di minaccia è detta *negazione del*

*servizio (denial of service)*. In questo capitolo, discuteremo come un SO affronta le minacce generiche e quelle riguardanti i programmi e i dati memorizzati nei file.

Le tecniche implementate nei SO per rispondere alle minacce ai dati e ai programmi possono essere divise in due categorie:

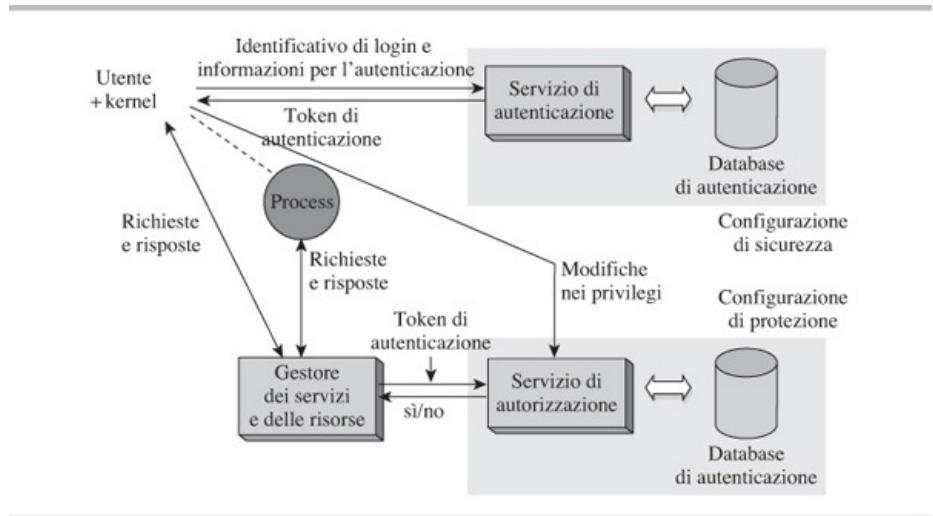
- le misure di *sicurezza* proteggono i dati e i programmi di un utente dall'interferenza da parte di persone o programmi al di fuori del sistema operativo; ci riferiamo, in senso lato, a tali persone e ai loro programmi come *non utenti*;
- le misure di *protezione* proteggono i dati e i programmi di un utente dall'interferenza da parte degli altri utenti del sistema.

La **Tabella 15.1** descrive due metodi usati dai sistemi operativi per l'implementazione della sicurezza e della protezione. L'autenticazione, che ha come obiettivo la sicurezza, consiste nel verificare l'identità di una persona. L'autenticazione messa in atto dal sistema si basa su due tipi di assunzioni. Un'assunzione banale è la seguente: una persona è l'utente che dice di essere se sa qualcosa che ci si aspetta che solo il SO e l'utente sappiano, per esempio una password. In questo caso si parla di *autenticazione per conoscenza*. L'altro metodo di autenticazione si basa su cose che si assume che solo l'utente abbia. Per esempio, l'autenticazione *biometrica* è basata su qualche caratteristica biologica unica e inalterabile come le impronte digitali, la retina o l'iride. L'autorizzazione è il metodo chiave di implementazione della protezione. Consiste (1) nell'assegnare a un utente un *privilegio di accesso* a una risorsa, ovvero un diritto di accesso alla risorsa nel modo specificato ([Capitolo 13](#)), e (2) nel determinare se un utente ha il diritto di accesso alla risorsa con le modalità specifiche.

| Termino        | Descrizione                                                                                                                                                                                                                                                                                                                                                                                              |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Autenticazione | L'autenticazione è la fase di verifica dell'identità dell'utente. I sistemi operativi, molto spesso, eseguono l'autenticazione <i>per conoscenza</i> . Ad una persona, che afferma di essere l'utente X, viene richiesto di provare qualche conoscenza condivisa solo tra il SO e l'utente X, come, per esempio una password.                                                                            |
| Autorizzazione | L'autorizzazione ha due aspetti: <ol style="list-style-type: none"> <li>1. assegnare un insieme di privilegi di accesso a un utente; per esempio, a un utente sono stati concessi i privilegi di lettura e scrittura su un file, mentre ad altri sono stati concessi quelli di sola lettura;</li> <li>2. verificare un diritto di accesso dell'utente a una risorsa con determinate modalità.</li> </ol> |

**Tabella 15.1** Terminologia usata nell'ambito della sicurezza e della protezione delle informazioni.

La [Figura 15.1](#) mostra uno schema generale di implementazione della sicurezza e della protezione in un sistema operativo. La configurazione di sicurezza è mostrata nel riquadro ombreggiato nella parte superiore della figura. Si compone del *servizio di autenticazione* e del *database di autenticazione*. Il database di autenticazione contiene una coppia del tipo (login id, informazioni di validazione) per ogni utente registrato del sistema operativo, dove il campo informazioni di validazione tipicamente è una forma cifrata della password utente. Per accedere al sistema, una persona invia al kernel la propria login id e la propria password. Il kernel passa queste informazioni al servizio di autenticazione, che cifra la password e la confronta con il contenuto del campo informazioni di validazione dell'utente memorizzato nel database di autenticazione. Se tale controllo viene superato, il servizio di autenticazione genera un *token (gettone) di autenticazione* per l'utente e lo restituisce al kernel. Il token di autenticazione, tipicamente, è la user id assegnata all'utente. Ogni volta che l'utente o un processo iniziato dall'utente effettua una richiesta di accesso a una risorsa, il kernel aggiunge alla richiesta il token di autenticazione dell'utente per facilitare i controlli di protezione.



**Figura 15.1** Configurazione generale della sicurezza e della protezione in un sistema operativo.

La configurazione di protezione è mostrata nel riquadro ombreggiato nella parte inferiore della [Figura 15.1](#). È composta dal *servizio di autorizzazione* e dal *database di autorizzazione*. Il *database di autorizzazione* contiene triple del tipo (token di autenticazione, id della risorsa, privilegi). Quando un utente vuole attribuire ad altri utenti del sistema i privilegi di accesso a un suo file, o revocare qualche privilegio di accesso precedentemente attribuito su un file, effettua una richiesta al kernel. Come mostrato in [Figura 15.1](#), il kernel inoltra la richiesta al servizio di autorizzazione unitamente al token di autenticazione dell'utente. Il servizio di autorizzazione effettua, poi, le modifiche appropriate nel database di autorizzazione. Per accedere a una risorsa, un utente o il suo processo invia una richiesta al *gestore dei servizi e delle risorse*. La richiesta contiene l'id della risorsa, il tipo di accesso desiderato e il token di autenticazione dell'utente. Il gestore dei servizi e delle risorse inoltra la richiesta al servizio di autorizzazione, che determina se l'utente possiede i privilegi per utilizzare la risorsa nel modo desiderato e invia una risposta positiva o negativa al gestore dei servizi e delle risorse. La risposta determina se la richiesta dell'utente sarà soddisfatta.

Non tutti i sistemi operativi contemplano tutti gli elementi mostrati in [Figura 15.1](#) nelle loro configurazioni di sicurezza e protezione. Per esempio, nella maggior parte dei sistemi operativi moderni, le informazioni riguardanti l'autorizzazione sono, generalmente, conservate e utilizzate dal file system, quindi il sistema operativo non ha il database di autorizzazione e non esegue l'autorizzazione.

La distinzione tra sicurezza e protezione fornisce una netta separazione per il SO. In un sistema operativo convenzionale, ciò che riguarda la sicurezza è limitato ad assicurare che solo gli utenti registrati possano usare il sistema. Quando una persona tenta di accedere al sistema, viene eseguito un controllo di sicurezza. Il sistema decide se la persona è un utente del SO e identifica la sua user id. Dopo questo controllo, tutte le minacce alle informazioni memorizzate nel sistema riguardano la protezione; il SO utilizza la user id di una persona per determinare se può accedere a uno specifico file nel SO. Tuttavia, in un sistema distribuito, le questioni legate alla sicurezza sono più complesse a causa della presenza della componente di rete (Capitolo 21). Limitiamo la discussione in questo capitolo solo ai sistemi operativi convenzionali.

### **Meccanismi e politiche**

La [Tabella 15.2](#) descrive i meccanismi e le politiche in ambito sicurezza e protezione. Le politiche di sicurezza specificano se a un utente sarà consentito utilizzare un sistema. Le politiche di protezione specificano se a un utente sarà consentito l'accesso a un determinato file. Entrambe queste politiche sono applicate al di fuori del dominio del SO, pertanto un amministratore di sistema decide se una persona potrà diventare un utente del sistema e quest'ultimo specificherà quali utenti potranno accedere ai suoi file. I meccanismi di sicurezza e protezione implementano queste politiche utilizzando i contenuti dei database di autenticazione e autorizzazione per effettuare specifici

controlli durante le operazioni di sistema.

|            |                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Sicurezza  | <ul style="list-style-type: none"><li><i>Politica</i>: una persona può o non può diventare un utente del sistema. L'amministratore di sistema impiega una politica nel registrare nuovi utenti.</li><li><i>Meccanismi</i>: aggiungere o cancellare utenti, verificare se una persona è un utente registrato (per esempio, eseguire l'autenticazione), utilizzare la cifratura per assicurare la confidentialità delle password.</li></ul> |
| Protezione | <ul style="list-style-type: none"><li><i>Politica</i>: il proprietario del file specifica la politica di autorizzazione per un file. Decide quali utenti possono accedere a un file e in quale modo.</li><li><i>Meccanismi</i>: impostare o modificare le informazioni di autorizzazione per un file. Controllare se una richiesta di elaborazione file è compatibile con i privilegi utente.</li></ul>                                   |

**Tabella 15.2** Politiche e meccanismi in ambito sicurezza e protezione.

### 15.1.1 Obiettivi di sicurezza e protezione

La **Tabella 15.3** descrive i quattro obiettivi di sicurezza e protezione, cioè *segretezza*, *privatezza*, *autenticità* e *integrità* dell'informazione.

| Obiettivo   | Descrizione                                                                                                                                                                 |
|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Segretezza  | Solo gli utenti autorizzati hanno accesso alle informazioni. Questo obiettivo è anche detto <i>confidenzialità</i> .                                                        |
| Privatezza  | Le informazioni sono usate solo per gli scopi per i quali erano state intese e condivise.                                                                                   |
| Autenticità | È possibile appurare la sorgente o il mittente dell'informazione e verificare inoltre che l'informazione è stata conservata nella forma in cui era stata creata od inviata. |
| Integrità   | Non è possibile distruggere o alterare le informazioni, per esempio cancellando un disco.                                                                                   |

**Tabella 15.3** Obiettivi della sicurezza e protezione informatica.

Dei quattro obiettivi, solo la *privatezza* è un elemento esclusivamente legato alla protezione. Un SO gestisce la privatezza tramite il servizio di autorizzazione e il gestore dei servizi e delle risorse ([Figura 15.1](#)). Il servizio di autorizzazione verifica se un utente possiede i privilegi per accedere a una risorsa nel modo specificato, e il gestore dei servizi e delle risorse respinge le richieste che non sono compatibili con i privilegi utente. È compito degli utenti assicurare la privatezza delle loro informazioni usando questa configurazione. Un utente che vuole condividere i suoi dati e i suoi programmi con altri utenti dovrebbe impostare l'autorizzazione alle sue informazioni secondo il ben noto principio *need-to-know*: dovrebbero essere autorizzate ad accedere solo quelle persone che hanno bisogno di utilizzare le informazioni per una funzione legittima.

*Segretezza*, *autenticità* e *integrità* riguardano sia la protezione che la sicurezza. Come elemento di protezione, la segretezza, l'autenticità e l'integrità sono facili da soddisfare poiché l'identità di un utente dovrebbe già essere stata verificata e il gestore dei servizi e delle risorse userà le informazioni di autorizzazione, ovvero una parte della configurazione di protezione mostrata in [Figura 15.1](#). Tuttavia, è necessario elaborare soluzioni per soddisfare segretezza, autenticità e integrità come elemento di sicurezza. Queste saranno discusse nel [Capitolo 20](#).

### 15.1.2 Minacce alla sicurezza e alla protezione

Vediamo come e quando si verificano in un SO le minacce alla sicurezza e alla protezione; innanzitutto, consideriamo un SO convenzionale. Le sue procedure di autenticazione assicurano che solo gli utenti registrati possono collegarsi al sistema e attivare processi. Di conseguenza, il SO sa quale utente ha iniziato uno specifico

processo e, quindi, può prontamente controllare se, al processo, è consentito utilizzare una specifica risorsa. Quando i processi comunicano con altri processi, anche le azioni del SO relative alla comunicazione sono confinate sullo stesso sistema di elaborazione. Di conseguenza, un accesso illegale a una risorsa o a un servizio da parte di un processo e un tentativo di alterare i messaggi sono entrambi minacce alla protezione piuttosto che alla sicurezza.

La situazione è diversa quando un sistema ha una connessione Internet e un utente scarica dati o programmi da Internet. Alcune persone o programmi esterni al SO possono essere in grado di alterare i dati e i programmi di cui si sta effettuando il download. Le minacce sollevate da tali dati e programmi sono, per definizione, minacce alla sicurezza.

Nei sistemi distribuiti, il verificarsi di minacce alla sicurezza è molto più facile. Un messaggio tra processi può superare il dominio di un nodo quando si sposta tra un mittente e un destinatario. La comunicazione tra nodi ha luogo tra link di comunicazione, inclusi i link pubblici. Di conseguenza, è possibile che una entità esterna alteri i messaggi. Discuteremo le misure per contrastare tali minacce nel [Capitolo 20](#).

## 15.2 Attacchi alla sicurezza

I tentativi di infrangere la sicurezza di un sistema sono detti *attacchi alla sicurezza*, e la persona o il programma che effettua l'attacco è detto *attaccante* o *intruso*. Gli attacchi alla sicurezza più comuni sono due:

- *impersonificazione (masquerading)*: assumere l'identità di un utente registrato del sistema attraverso strumenti illegittimi;
- *negazione del servizio (denial of service)*: impedire che gli utenti registrati del sistema accedano alle risorse per cui posseggono i privilegi d'accesso.

In un attacco di tipo *masquerading*, concluso con successo, l'intruso ottiene l'accesso alle risorse alle quali è autorizzato l'utente che sta impersonando, di conseguenza può modificare o distruggere programmi e dati appartenenti a esso. Il modo più semplice di lanciare un attacco di tipo masquerading è scoprire la password di un utente e, con questa, superare il test di autenticazione al momento del login. Un altro approccio è eseguire il masquerading in un modo più subdolo tramite programmi importati in ambiente software. Discuteremo questo approccio nel Paragrafo 15.2.1.

Un attacco di tipo *denial-of-service*, anche detto attacco DoS, è lanciato sfruttando alcuni punti deboli nella progettazione o nel funzionamento di un SO. Si può lanciare un attacco DoS tramite diversi mezzi; alcuni di questi possono essere impiegati solo dagli utenti del sistema, mentre altri possono essere utilizzati da intrusi situati in altri sistemi. Molti di questi strumenti sono legittimi, il che rende semplice lanciare attacchi DoS e difficile per un SO individuarli ed evitarli. Per esempio, un attacco DoS può essere lanciato sovraccaricando una risorsa tramite mezzi fantasma in modo che ne risulti negato l'uso agli utenti legittimi. Se il kernel di un SO limita il numero totale di processi che possono essere creati in modo da controllare il carico sulle strutture dati del kernel, un utente può creare un numero di processi tale da bloccare il sistema. Analogamente, l'uso dei network socket può essere negato aprendo un elevato numero di socket. Un attacco DoS può, inoltre, essere lanciato modificando un programma che offre un servizio, o distruggendo qualche informazione di configurazione nel kernel, per esempio l'uso di un dispositivo di I/O può essere negato cambiando la sua entry nella tabella dei device fisici del kernel (Paragrafo 14.5.2).

Un *attacco DoS di rete* può, inoltre, essere lanciato inondando la rete con messaggi diretti a un particolare server, in tal modo la banda di rete non risulta disponibile ai messaggi "genuini" ed, inoltre, il server è talmente impegnato nella ricezione dei messaggi che non riesce a rispondere ad alcun messaggio. Un *attacco DoS distribuito* è quello lanciato da alcuni intrusi localizzati in host diversi della rete; è più difficile da individuare ed evitare rispetto a quelli non distribuiti.

Molti altri attacchi alla sicurezza sono lanciati attraverso il sistema di comunicazione dei messaggi. La lettura non autorizzata di messaggi, anche detta *eavesdropping*, e l'alterazione di messaggi sono due attacchi di questo tipo. Questi attacchi principalmente si verificano nei sistemi operativi distribuiti, quindi ne discuteremo nel [Capitolo 20](#).

### 15.2.1 Cavalli di Troia, virus e worm

*I cavalli di Troia, i virus e i worm* sono programmi contenenti codice in grado di lanciare

un attacco alla sicurezza quando sono attivati. La [Tabella 15.4](#) sintetizza le loro caratteristiche. Un cavallo di Troia o un virus accede a un sistema quando un utente insospettabile scarica programmi da Internet o da un disco. Diversamente, un worm, esistente in un sistema, si auto-propaga ad altri sistemi.

| Minaccia         | Descrizione                                                                                                                                                                                                                                              |
|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cavallo di Troia | Un programma che esegue una funzione autorizzata, nota a un SO o ai suoi utenti e che, inoltre, ha una componente nascosta che può essere usata successivamente per scopi illegali come attacchi alla sicurezza dei messaggi o attacchi di masquerading. |
| Virus            | Un pezzo di codice che si unisce ad altri programmi del sistema e si propaga ad altri sistemi quando tali programmi vengono copiati o trasferiti.                                                                                                        |
| Worm             | Un programma che si propaga ad altri sistemi sfruttando i punti deboli della sicurezza nel SO come la debolezza negli strumenti per la creazione di processi remoti.                                                                                     |

**Tabella 15.4** Minacce alla sicurezza attraverso cavalli di Troia, virus e worm.

Un *cavallo di Troia* è un programma che ha una componente nascosta progettata per causare un danno al sistema. Per esempio, può cancellare un hard disk in un computer, quindi una violazione della integrità, carpire informazioni per il masquerading, o forzare un sistema per provocarne un crash o un rallentamento, comportando la negazione del servizio (denial of service). Un tipico esempio di cavallo di Troia è un falso programma di login, che visualizza un prompt di login fittizio per indurre un utente a rivelare la propria password, da utilizzare successivamente per il masquerading. Poiché un cavallo di Troia è caricato esplicitamente da un utente insospettabile, non è difficile tracciare l'autore o l'origine.

Un *virus* è un pezzo di codice che “infetta” altri programmi e si propaga ad altri sistemi quando i programmi infetti vengono copiati o trasferiti. Un virus chiamato *virus eseguibile* o file virus agisce in questo modo: ispeziona il disco, seleziona un programma da infettare e aggiunge il suo codice, che chiameremo *codice virale*, al codice del programma. Esso modifica, inoltre, il codice del programma in modo tale che, quando è eseguito il programma, venga attivato il codice virale. Un semplice modo di fare ciò consiste nel modificare la prima istruzione nel codice del programma, ossia, l’istruzione contenuta nell’indirizzo di inizio esecuzione del programma (Paragrafo 11.3), per trasferire il controllo al codice virale. Quando il codice virale viene attivato, ispeziona il disco cercando altri programmi da infettare. Dopo aver infettato questi programmi, passa il controllo al vero codice del programma. Un utente non ha modo di sapere se un programma è stato infettato sia perché i passi per infettare altri programmi non impiegano molto tempo di CPU sia perché il funzionamento dei programmi infettati non viene alterato. Il modo in cui un virus si unisce a un altro programma lo rende molto più difficile da tracciare rispetto a un cavallo di Troia.

Un virus tipicamente configura una *back door* che può essere sfruttata per scopi distruttivi successivamente. Per esempio, è possibile configurare un demone dormiente attivabile da un trigger, che potrebbe essere una data o un’ora particolare oppure un messaggio; una volta attivato, il virus può eseguire un’azione distruttiva. Altre categorie di virus infettano e si replicano in maniera diversa. Oltre ai file virus descritti precedentemente, il *boot-sector virus* si stabilisce nel settore di boot di un hard disk o di un floppy disk. Un virus di questo tipo ha l’opportunità di essere eseguito durante il boot di sistema e ha l’opportunità di replicarsi quando viene creato un nuovo disco di boot.

I virus eseguibili e boot-sector proliferavano quando i programmi erano caricati tramite i floppy. L’uso di CD, che non possono essere modificati, ha limitato tale minaccia. Tuttavia, i nuovi virus utilizzano tecnologie più sofisticate per infrangere le difese dei sistemi. Un *e-mail virus* accede a un sistema tramite una e-mail e invia falsi messaggi a tutti gli utenti i cui indirizzi sono presenti nella rubrica. Il virus Melissa del 1999 usò un codice virale costituito da un documento Word inviato a un newsgroup Internet. Il virus era attivato all’apertura di una copia scaricata del documento; esso stesso inviava il documento a 50 persone il cui indirizzo e-mail era presente nella rubrica dell’utente attaccato. La back door in questo caso era un piccolo frammento di codice, associato al

documento Word, scritto in linguaggio Visual Basic for Application (VBA). Esso era attivato tramite la proprietà di autoesecuzione di Microsoft Word, che automaticamente esegue il programma associato a un documento Word al momento della sua apertura. Il virus I LOVE YOU dell'anno 2000 era un e-mail virus il cui codice virale era allegato a una e-mail. Questo codice era eseguito quando l'utente cercava di visualizzare l'allegato. Esso inviava e-mail contenenti copie di se stesso ad altri utenti e, inoltre, modificava i file sul disco dell'host sul quale veniva eseguito. Sia Melissa che I LOVE YOU furono virus così potenti da costringere grandi compagnie a fermare i loro server e-mail finché non riuscirono a controllarli.

I virus usano varie tecniche per eludere l'individuazione da parte dei software antivirus. Queste tecniche comprendono: cambiare la loro forma, comprimere o cifrare il loro codice e i loro dati, nascondersi in parti del SO, ecc.

Un *worm* è un programma che si autoreplica in altri sistemi sfruttando fallo nella loro configurazione di sicurezza. È molto più difficile tracciarlo rispetto a un virus proprio a causa della sua natura autoreplicante. I worm sono noti per il tasso inimmaginabilmente alto di replicazione, sovraccaricando, così, la rete e consumando tempo di CPU durante la replicazione. Il worm Code Red del 2001 si diffuse a un quarto di milione di host in 9 ore, effettuando un attacco di buffer overflow. Il worm Morris del 1988 si diffuse a migliaia di host tramite tre punti deboli del sistema Unix.

- Il comando remote login *rsh* di Unix abilitava un utente a impostare una configurazione tramite la quale poter accedere a un host remoto senza dover fornire una password. Il worm cercava i file che memorizzavano i nomi degli host remoti ai quali si poteva accedere tramite *rsh* e utilizzava questi file per spostarsi su tali host remoti.
- La tecnica del *buffer overflow*, descritta nel Paragrafo 15.2.2, forza un demone su un server non protetto ad accettare e eseguire un pezzo di codice. Il worm Morris utilizzava questo attacco sul demone *finger* di un host remoto Unix per mandare il suo codice all'host remoto ed effettuare la sua esecuzione su quell'host.
- La facility debug venne usata nel programma *sendmail* di Unix per spedire una copia del suo codice a un altro host ed eseguirlo su di esso.

Gli attacchi alla sicurezza lanciati tramite cavalli di Troia, virus o worm possono essere contrastati attraverso le seguenti misure:

- caricando con prudenza nuovi programmi in un sistema;
- usando programmi antivirus;
- arginando i punti deboli nel sistema di sicurezza appena scoperti o riportati.

Il caricamento di programmi dai dischi originali, usati dai fornitori per la distribuzione, può eliminare una sorgente primaria di cavalli di Troia o virus. Questo approccio è diventato particolarmente operativo con la tecnologia dei compact disk (CD). Poiché tali dischi non possono essere modificati, il programma originale non può essere sostituito da un cavallo di Troia, o un disco distribuito dal fornitore non può essere stato infettato da un virus.

I programmi antivirus analizzano ogni programma su un disco per vedere se contiene qualche caratteristica analoga a quelle dei virus noti. L'elemento principale, che essi verificano, è l'eventuale modifica dell'indirizzo di inizio esecuzione del programma oppure controllano se i primi byte del programma eseguono azioni simili alla replicazione, per esempio, se attaccano codice a qualche programma su disco.

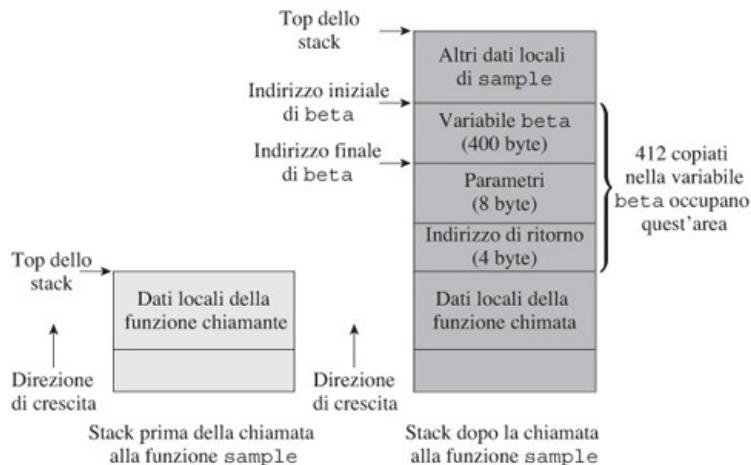
I fornitori di SO pubblicano informazioni circa i punti deboli della sicurezza dei loro sistemi operativi sui loro siti web periodicamente e forniscono security patch che risolvono questi problemi. Un amministratore di sistema dovrebbe controllare tali informazioni e installare le patch regolarmente; ciò contrasterebbe gli attacchi alla sicurezza lanciati tramite worm.

## 15.2.2 Tecnica del buffer overflow

La tecnica del *buffer overflow* può essere impiegata per forzare un programma a eseguire un codice prodotto da un intruso per ottenere l'accesso privilegiato a un sistema host. Questa tecnica è stata usata con effetti devastanti in mail server e in altri server Web. L'idea di base di questa tecnica è semplice: la maggior parte dei sistemi ha un elemento di vulnerabilità fondamentale; per esempio alcuni programmi non validano

le lunghezze degli input che ricevono dagli utenti o da altri programmi. A causa di ciò, la buffer area, in cui tali input vengono memorizzati, può andare in overflow e sovrascrivere i contenuti di aree di memoria adiacenti. Sulle piattaforme hardware che usano stack che crescono verso indirizzi bassi di memoria, per esempio, l'architettura Intel 80 × 86, tali overflow consentono l'esecuzione di pezzi di codice sotto forma di dati memorizzati nel buffer. Un codice di questo tipo potrebbe lanciare una varietà di attacchi alla sicurezza.

La [Figura 15.2](#) illustra come un intruso può lanciare un attacco alla sicurezza tramite la tecnica di buffer overflow. Un Web server è in funzione. Quando una delle sue funzioni chiama una funzione *sample* con due parametri, lo stack è utilizzato per due scopi – il primo è memorizzare un indirizzo di ritorno che sarà usato per ripristinare l'esecuzione della funzione chiamante quando *sample* termina la sua esecuzione; il secondo è passare i parametri a *sample*. Supponiamo che l'indirizzo di ritorno e ciascuno dei parametri occupino 4 byte, e che lo stack cresca all'indietro in memoria, cioè verso indirizzi più piccoli. Durante l'esecuzione, *sample* alloca spazio per le sue variabili locali sullo stack. In questo modo una variabile *beta*, con una dimensione di 400 byte, è adiacente ai parametri sullo stack. Quando invocata, la *sample* accetta un messaggio contenente una richiesta per il Web server e la copia in *beta*; non effettua, però, controlli per assicurare che la richiesta non sia più lunga di 400 byte. Un intruso sfrutta questo punto debole inviando un messaggio lungo 412 byte, in cui i primi 408 byte contengono un codice e gli ultimi quattro byte contengono l'indirizzo di partenza di questo codice. Quando questo messaggio è copiato in *beta*, i suoi ultimi quattro byte sovrascriveranno la entry nello stack che contiene l'indirizzo di ritorno. Quando l'esecuzione di *sample* termina, il controllo viene passato all'indirizzo memorizzato nello stack. Così, verrà attivato il codice dell'intruso ed eseguito con i privilegi del Web server. Questo codice potrebbe danneggiare il codice e i file del Web server in modo tale che non riesca a rispondere ad alcuna richiesta, causando così un denial of service, oppure potrebbe spedire false mail. Come può un intruso sapere quale Web server è vulnerabile alla tecnica del buffer overflow e quanti byte sono necessari per sovrascrivere l'indirizzo di ritorno? Entrambe le risposte possono essere dedotte tramite una prova.



#### Come può essere utilizzato un buffer overflow per lanciare un attacco alla sicurezza

1. Lo stack cresce all'indietro, cioè verso indirizzi più piccoli di memoria. Lo stack appare, come mostrato sulla sinistra, prima che la funzione, attualmente in esecuzione, chiama la funzione `sample`.
2. Il codice della funzione chiamante effettua il push sullo stack dell'indirizzo di ritorno e di due parametri di `sample`. Ciascuno di questi occupa quattro byte.
3. Il codice di `sample` aloca la variabile `beta` e altre variabili sullo stack. Lo stack appare ora come mostrato sulla destra. Notare che l'indirizzo iniziale di `beta` è al margine inferiore della memoria allocata a esso. L'indirizzo finale di `beta` è adiacente all'ultimo byte dei parametri.
4. La funzione `sample` copia 412 byte nella variabile `beta`. I primi 408 byte contengono codice la cui esecuzione causerà una violazione di sicurezza. I byte 409-412 contengono l'indirizzo di partenza di questo codice. Questi quattro byte sovrascrivono l'indirizzo di ritorno nello stack.
5. La funzione `sample` esegue un'istruzione di `return`. Il controllo è trasferito all'indirizzo trovato nella entry dello stack che doveva contenere l'indirizzo di ritorno. In realtà, invece, viene invocato il codice nella variabile `beta`. Esso viene eseguito con i privilegi della funzione chiamante.

**Figura 15.2** Lanciare un attacco alla sicurezza tramite la tecnica di buffer overflow.

### 15.3 Aspetti formali di sicurezza

Per dimostrare formalmente che un sistema può resistere a tutte le forme di attacchi alla sicurezza, abbiamo bisogno di un modello di sicurezza comprendente politiche e meccanismi di sicurezza, una lista di minacce, una lista di attacchi fondamentali e una metodologia di prova. La lista di attacchi deve essere provatamente completa nel senso che dovrebbe essere possibile produrre qualsiasi minaccia della lista delle minacce tramite una combinazione degli attacchi fondamentali. La metodologia di prova dovrebbe essere in grado di assodare se il modello di sicurezza può resistere a certi tipi di attacchi.

I primi lavori in ambito sicurezza seguivano queste direttive. Nel *modello takegrant* di sicurezza dei computer [Landwehr (1981)], ai processi venivano attribuiti i privilegi sugli oggetti e sugli altri processi. Un privilegio su un oggetto attribuiva all'holder il privilegio di accedere all'oggetto nel modo specificato. Un privilegio per un altro processo attribuiva all'holder il privilegio di prendere un privilegio di accesso posseduto da un altro processo (un'operazione di *take*), o di trasferire uno dei suoi privilegi di accesso a un altro processo (un'operazione di *grant*). La prova consiste nell'accertare se un determinato processo può ottenere un particolare privilegio di accesso per un determinato oggetto tramite una serie di operazioni di *take* e *grant*. Nell'esempio seguente, discutiamo come si può individuare un punto debole nella sicurezza tramite l'approccio formale.

#### Esempio 15.1 - Individuazione di una falla di sicurezza

In un'organizzazione che utilizza sicurezza di tipo militare, tutti i documenti sono classificati secondo tre livelli di sicurezza: non classificato, riservato e segreto. Alle persone che lavorano in un'organizzazione vengono attribuite autorizzazioni di sicurezza dette U (unclassified = non classificato), C (confidential = riservato) e S (secret = segreto) con la condizione che una persona può accedere a tutti i documenti del suo livello di classificazione di sicurezza e dei livelli più bassi della classificazione. Quindi, una persona con una classificazione C può accedere a documenti confidenziali

e non classificati, ma è vietato l'accesso ai documenti segreti.

L'organizzazione usa un sistema Unix e le persone nell'organizzazione usano comandi/proprietà Unix per accedere ai file contenenti i documenti. In questo modo, ci aspettiamo che un programma eseguito da un utente può accedere a un documento di uno specifico livello di sicurezza solo se l'utente possiede le appropriate autorizzazioni di sicurezza. Per controllare se la sicurezza dei documenti è a prova di errore, tutte le operazioni nel sistema sono modellate e viene effettuato un controllo per vedere se una persona può accedere a un documento che è a un livello più alto di classificazione rispetto alla sua autorizzazione di sicurezza. Si è visto che una combinazione di assegnazioni indiscriminate di privilegi di "esecuzione" per i programmi agli utenti e l'uso della proprietà Unix *setuid* può consentire a un utente di accedere a un documento proibito. Questo può accadere perché la funzione *setuid* permette a un utente di eseguire un programma con i privilegi del proprietario del programma (Paragrafo 15.9.2). Quindi se un utente può eseguire un programma il cui proprietario possiede un'autorizzazione di sicurezza più alta, egli può "assumere (take)" l'autorizzazione di sicurezza del proprietario del programma.

Questa anomalia nella sicurezza può essere eliminata o vietando l'uso della funzione *setuid* oppure limitando il privilegio di "esecuzione" per un programma solo agli utenti la cui autorizzazione di sicurezza non è più bassa di quella del proprietario di programma.

Il punto debole nell'Esempio 15.1 poteva essere anche individuato tramite procedure manuali; le procedure manuali, però, diventano meno affidabili quanto più i sistemi crescono in complessità. I metodi formali costruiscono sequenze possibili di operazioni e deducono o verificano le loro proprietà. In questo modo, essi possono scoprire sequenze di operazioni che hanno conseguenze disastrose, oppure asserire che tali sequenze non esistono.

Anche l'approccio formale possiede alcuni svantaggi. Al crescere della dimensione del sistema da analizzare, i requisiti di elaborazione e memorizzazione dei metodi formali superano le capacità dei sistemi di elaborazione attuali. L'approccio formale è, inoltre, difficile da applicare perché richiederebbe una specifica completa del sistema e una lista esaurente degli attacchi fondamentali che non è possibile sviluppare per i moderni sistemi operativi. Tale approccio richiede, inoltre, una chiara definizione delle politiche di sicurezza. Questo requisito è forte perché la maggior parte delle politiche di sicurezza si compone di regole informali affinché chiunque in un'organizzazione possa comprenderle. Tuttavia, proprio in questo ambito l'approccio formale contribuisce sostanzialmente a migliorare la sicurezza, enfatizzando la necessità di specifiche precise.

## 15.4 Cifratura

La cifratura è l'applicazione di una trasformazione algoritmica ai dati. Quando un dato è memorizzato nella sua forma cifrata, soltanto un utente o un suo processo, che conosce come rispristinare la forma originale del dato, lo può utilizzare. Questa caratteristica aiuta a mantenere la *riservatezza* del dato. I meccanismi di protezione e sicurezza utilizzano la cifratura per proteggere le informazioni relative agli utenti e alle loro risorse; tuttavia, potrebbe essere anche usata per proteggere le informazioni appartenenti agli utenti. La *crittografia* è quella branca delle scienze che si occupa di tecniche di cifratura.

La [Tabella 15.5](#) schematizza i termini chiave e le definizioni usate in crittografia. La forma originale dei dati è detta *testo in chiaro (plaintext)* e la forma trasformata è detta *cifrata o testo cifrato (ciphertext)*. Usiamo la seguente notazione:

$P_d$  forma in chiaro del dato  $d$

$C_d$  forma cifrata del dato  $d$

| Termino   | Descrizione                                                                                                                                                                                                                                                                                                                                       |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Cifratura | La cifratura è l'applicazione di una trasformazione algoritmica $E_k$ ai dati, dove $E$ è un <i>algoritmo di cifratura</i> e $k$ è una <i>chiave di cifratura</i> . È usato per proteggere la riservatezza dei dati. Il dato originale è ricostruito applicando una trasformazione $D_k$ , dove $D$ è un <i>algoritmo di decifratura</i> e $k'$ è |

|                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                     | una <i>chiave di decifratura</i> . Uno schema che usa $k = k'$ è detto <i>cifratura simmetrica</i> , e uno che usa $k \neq k'$ è detto <i>cifratura asimmetrica</i> .                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Plaintext<br>(testo in chiaro)      | Il dato da cifrare.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| Ciphertext<br>(testo cifrato)       | La forma cifrata del testo in chiaro.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Confusione                          | Il principio di Shannon della confusione richiede che i cambiamenti causati in un testo cifrato dovuti a una modifica del testo in chiaro non saranno facili da individuare.                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Diffusione                          | Il principio di Shannon della diffusione richiede che l'effetto di una piccola sottostringa nel testo in chiaro sia ripartito su molte cifre nel testo cifrato.                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| Attacchi ai sistemi di crittografia | Un <i>attacco</i> è una serie di tentativi, da parte di un intruso, di trovare una funzione di decifratura $D_k$ . In un attacco <i>solo testo cifrato (ciphertext only)</i> , l'intruso può esaminare solo un insieme di testi cifrati per determinare $D_k$ . In un attacco <i>testo in chiaro conosciuto (known plaintext)</i> , l'intruso ha l'opportunità di esaminare la forma in chiaro e quella cifrata dello stesso dato, mentre in un attacco <i>testo in chiaro selezionato (chosen plaintext)</i> l'intruso può scegliere un testo in chiaro e ottenere la sua forma cifrata per eseguire l'attacco. |
| Funzione one-way                    | Una funzione, il calcolo della cui inversa è talmente oneroso da essere considerato impraticabile. Il suo utilizzo come funzione di cifratura rende difficili gli attacchi di crittografia.                                                                                                                                                                                                                                                                                                                                                                                                                      |
| Cifratura a blocchi                 | La tecnica della cifratura a blocchi (block cipher) consiste nel sostituire blocchi di misura fissa del testo in chiaro con blocchi del testo cifrato. Ciò introduce un po' di confusione, ma non introduce sufficiente diffusione.                                                                                                                                                                                                                                                                                                                                                                              |
| Cifratura a flusso                  | Sia il testo in chiaro che la chiave di cifratura sono considerati come flussi di bit (bit stream). I bit nel testo in chiaro vengono cifrati usando un ugual numero di bit della chiave di cifratura. Una cifratura a flusso (stream cipher) non introduce confusione e introduce una limitata diffusione; tuttavia, qualche sua variante può introdurre un elevato livello di diffusione.                                                                                                                                                                                                                      |
| DES                                 | Il Data Encryption Standard del National Bureau of Standards, adottato nel 1976, usa una tecnica basata sulla cifratura a blocchi e prevede, come opzione, la cifratura a blocchi in cascata. Esso effettua 16 iterazioni, che eseguono trasformazioni complesse sul testo in chiaro o sul testo cifrato intermedio.                                                                                                                                                                                                                                                                                             |
| AES                                 | L'Advanced Encryption Standard è il nuovo standard adottato dal National Institute of Standards and Technology (formalmente conosciuto come National Bureau of Standards) nel 2001. Esso esegue tra 10 e 14 cicli di operazioni, ognuna effettua solo sostituzioni e permutazioni, sui blocchi del testo in chiaro di 128, 192 o 256 bit.                                                                                                                                                                                                                                                                        |

**Tabella 15.5** Termini e definizioni di crittografia.

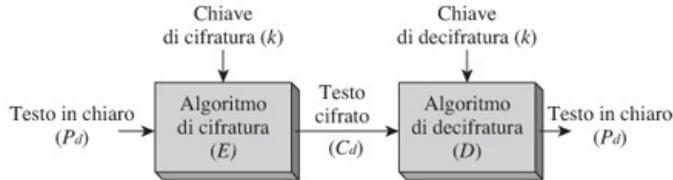
dove  $P_d \equiv d$ . La cifratura è eseguita applicando al dato un algoritmo di cifratura  $E$  con una specifica chiave di cifratura  $k$ . Il dato è ripristinato applicando l'algoritmo di decifratura  $D$  con una chiave  $k'$ . Nella forma più semplice di cifratura detta *cifratura simmetrica*, la decifratura è eseguita utilizzando la stessa chiave  $k$ . Nelle tecniche avanzate di cifratura, detta *cifratura asimmetrica*, viene usata una chiave differente  $k'$

per decifrare un testo cifrato.

La [Figura 15.3](#) illustra la cifratura simmetrica. Rappresentiamo la cifratura e la decifratura dei dati usando gli algoritmi  $E$  e  $D$  con chiave  $k$  come applicazione delle funzioni rispettivamente  $E_k$  e  $D_k$ . Così,

$$C_d = E_k(d)$$

$$P_d = D_k(C_d)$$



**Figura 15.3** Cifratura simmetrica del dato  $d$ .

Ovviamente le funzioni  $E_k$  e  $D_k$  devono soddisfare la relazione:

$$D_k(E_k(d)) = d, \text{ per ogni } d$$

quindi, un processo deve essere in grado di eseguire la trasformazione  $D_k$  allo scopo di ottenere il testo in chiaro (plaintext) dai dati cifrati.

In pratica, la cifratura è eseguita tramite gli algoritmi standard  $E$  e  $D$ . Di conseguenza, l'efficacia della cifratura dipende dal fatto che un intruso possa determinare la chiave di cifratura attraverso tentativi ed errori. Più avanti in questo paragrafo valuteremo le difficoltà per un intruso di scoprire la chiave di cifratura a causa del grande numero di tentativi necessari; tuttavia, teoricamente, non è impossibile farlo. Questa proprietà rende la cifratura efficace in senso probabilistico, sebbene non in senso assoluto. La riservatezza dei dati cifrati deriva da questa proprietà.

La riservatezza fornita tramite la cifratura aiuta, inoltre, a verificare l'integrità dei dati. Se la forma cifrata dei dati è alterata da un intruso, la sua decifratura, da parte di un processo che possiede il giusto algoritmo e la chiave di decifratura, produrrebbe dati incomprensibili, che rivelerebbero che qualcosa è stato alterato in modo non autorizzato. A causa di questa proprietà dei dati cifrati, usiamo il termine "decifratura errata" nel caso in cui la decifratura, tramite la chiave corretta, produca dati incomprensibili.

### 15.4.1 Attacchi ai sistemi di crittografia

Un attacco a un sistema di crittografia è composto da una serie di tentativi per individuare la funzione di decifratura  $D_k$ . Poiché  $D_k(E_k(d)) = d$ ,  $D_k$  è l'inversa di  $E_k$ . Di conseguenza, un attacco implica la ricerca dell'inversa di  $E_k$ . Se definiamo la *qualità* della cifratura in base all'abilità a resistere agli attacchi, lo scopo di una tecnica di cifratura è eseguire cifratura di alta qualità a basso costo. La qualità della cifratura è migliore se la funzione  $E_k$  è una *funzione one-way*, cioè se il calcolo della sua inversa tramite un attacco coinvolge una quantità non perseguitabile di tentativi e di tempo.

Un intruso, che può essere all'interno o al di fuori di un SO, può lanciare una varietà di attacchi a un sistema di crittografia. La natura dell'attacco dipende dal ruolo dell'intruso nel sistema. Se un intruso non può invocare la funzione di cifratura e può solo esaminare i dati in forma cifrata, si dovrà basare su prove. Questo è un approccio denominato prova-ed-errore (trial-and-error) in cui si prova, ripetutamente, la funzione  $D_k$  finché la sua applicazione al testo cifrato produce un output comprensibile. Questo attacco è detto *attacco esaustivo (exhaustive attack)* perché potrebbero essere provate tutte le possibilità per  $D_k$ .

Un attacco esaustivo prevede un elevato numero di tentativi. Per esempio, per infrangere uno schema di cifratura che utilizza una chiave a 56 bit tramite un attacco esaustivo potrebbero essere necessari  $2^{55}$  tentativi. Si pensava che un così grande numero rendesse uno schema di questo tipo computazionalmente sicuro, e che la qualità

della cifratura migliorasse all'aumentare del numero di bit nella chiave di cifratura. Tuttavia si possono impiegare potenti tecniche matematiche come l'analisi differenziale per trovare  $D_k$  in maniera molto più semplice rispetto a un attacco esaustivo. Inoltre, gli intrusi possono utilizzare gli attacchi descritti successivamente che prevedono meno tentativi di un attacco esaustivo. Vedremo alcuni esempi di questi attacchi quando discuteremo della sicurezza delle password nel Paragrafo 15.5.

Nell'*attacco solo testo cifrato*, un intruso ha accesso soltanto a una collezione di testi cifrati. Conseguentemente, per rendere l'attacco più efficace rispetto a quello esaustivo, l'intruso si basa sugli indizi estratti dall'analisi delle stringhe nei testi cifrati e dalle caratteristiche dei testi in chiaro, per esempio se sono formati solo da parole del dizionario. Nell'*attacco testo in chiaro conosciuto*, un intruso conosce il testo in chiaro corrispondente a un testo cifrato. Questo attacco è applicabile se un intruso riesce a ottenere una posizione all'interno del SO da cui può osservare sia il testo in chiaro che il corrispondente testo cifrato. Recuperare un numero sufficiente di coppie di testo in chiaro-testo cifrato fornisce indizi per determinare  $D_k$ . Nell'*attacco testo in chiaro scelto*, un intruso è in grado di fornire un testo in chiaro e osservare la sua forma cifrata o, equivalentemente, scegliere  $d$  e osservare  $E_k(d)$ . Ciò consente all'intruso di creare sistematicamente una collezione di coppie testo in chiaro-testo cifrato a supporto dei tentativi e del raffinamento dei tentativi durante l'attacco.

### 15.4.2 Tecniche di cifratura

Le tecniche di cifratura si differenziano nel modo in cui tentano di ostacolare i tentativi dell'intruso nel trovare  $D_k$ . L'approccio fondamentale è mascherare le caratteristiche di un testo in chiaro - per esempio, assicurare che un testo cifrato non riveli le caratteristiche del corrispondente testo in chiaro - senza, per questo, incorrere in un costo troppo alto di cifratura.

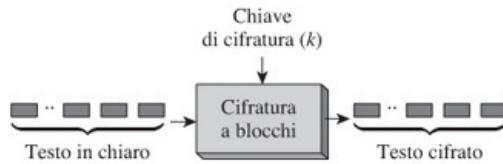
Consideriamo la tecnica di cifratura più semplice, la classica *sostituzione del blocco* (*substitution cipher*), che sostituisce ogni lettera in un testo in chiaro con un'altra lettera dell'alfabeto. Tale tecnica non maschera molto bene le caratteristiche di un testo in chiaro; quindi l'*analisi della frequenza* fornisce un metodo semplice per trovare  $D_k$ . La tecnica consiste nell'elencare le lettere dell'alfabeto in ordine di frequenza decrescente di utilizzo in una raccolta di testi cifrati, acquisire i dati standard sulla frequenza delle lettere nei testi in inglese, e organizzare le lettere in ordine di frequenza decrescente. Dunque, una buona scelta per  $D_k$  è una funzione che semplicemente associa una lettera della prima lista nella corrispondente lettera della seconda lista.

Come mascherare le caratteristiche di un testo in chiaro durante la cifratura? Shannon (1949) formulò due principi per la progettazione di tecniche di cifratura di alta qualità. Questi principi sono chiamati *confusione* e *diffusione*. Il principio di confusione enuncia: non dovrebbe essere facile trovare quali modifiche si verificheranno nel testo cifrato in seguito a un cambiamento nel testo in chiaro.

Il principio di diffusione enuncia: l'effetto di una piccola sottestringa nel testo in chiaro dovrebbe essere ripartito su molte cifre del testo cifrato. Questi principi assicurano che le caratteristiche di un testo in chiaro sono, effettivamente, mascherate perché parti individuali di un testo in chiaro e del relativo testo cifrato non avranno una forte correlazione tra loro. Nel seguito, descriviamo i quattro schemi di cifratura e discutiamo le loro proprietà di confusione e diffusione.

#### Cifratura a blocchi

La cifratura a blocchi è un'estensione della cifratura per sostituzione. Esegue una sostituzione di blocchi di dimensione fissa di un testo in chiaro con blocchi del testo cifrato di uguale dimensione. Per esempio, un blocco costituito da  $n$  bit è stato cifrato con una chiave  $k$  per ottenere un blocco di  $n$  bit del testo cifrato (Figura 15.4). Questi blocchi sono composti per ottenere il testo cifrato. La tecnica del cifrario a blocchi è semplice da implementare. Tuttavia, la confusione e la diffusione, da esso introdotte, sono limitate a un blocco nel testo cifrato. Di conseguenza, blocchi identici in un testo in chiaro producono blocchi identici nel testo cifrato. Questa caratteristica rende l'approccio vulnerabile a un attacco basato sull'analisi della frequenza e su testo in chiaro conosciuto oppure testo in chiaro scelto. Valori maggiori di  $n$  possono essere utilizzati per rendere tali attacchi meno praticabili.



**Figura 15.4** Cifratura a blocchi.

### Cifratura a flusso

Una cifratura a flusso considera che un testo in chiaro, e la chiave di cifratura siano un flusso di bit. La cifratura è eseguita usando una trasformazione che prende alcuni bit del testo in chiaro e un egual numero di bit della chiave di cifratura. Una scelta tipica per la trasformazione è una trasformazione bit-a-bit del testo in chiaro, ottenuta tipicamente eseguendo un'operazione di OR esclusivo tra un bit del testo in chiaro e un bit della chiave di cifratura.

Un cifrario a flusso è più veloce di un cifrario a blocchi. Non fornisce confusione o diffusione quando viene utilizzata una trasformazione bit-a-bit. Una variante di questa cifratura, chiamata *cifratura vernam*, utilizza un flusso random di bit come la sequenza chiave (key stream), la cui dimensione corrisponde esattamente alla dimensione del testo in chiaro. Di conseguenza, da sottostringhe identiche in un testo in chiaro non si arriva a sottostringhe identiche nel testo cifrato. Il *one-time pad*, noto per il suo utilizzo durante la seconda guerra mondiale, era, effettivamente, una cifratura vernam in cui era utilizzata una sequenza chiave per codificare solo un testo in chiaro. Ciò rese la cifratura "sicura".

Le varianti della cifratura a flusso sono state progettate per introdurre diffusione. Una cifratura di questo tipo opera come segue: viene usata una sequenza chiave a  $n$  bit per cifrare i primi  $n$  bit del testo in chiaro. I successivi  $n$  bit della sequenza chiave sono gli  $n$  bit del testo cifrato che sono già stati generati e, così via, finché non è cifrato l'intero testo in chiaro. Quindi, una sottostringa del testo in chiaro influenza la cifratura del resto del testo in chiaro; questo fornisce un alto livello di diffusione. Questa cifratura è detta cifratura *ciphertext autokey* (Figura 15.5). Se il generatore della sequenza chiave usa  $n$  bit del testo in chiaro appena cifrati, invece del suo testo cifrato, il cifrario è detto *self-synchronizing cipher*. La diffusione introdotta da esso è limitata solo ai successivi  $n$  bit del testo cifrato.



**Figura 15.5** Cifratura ciphertext autokey.

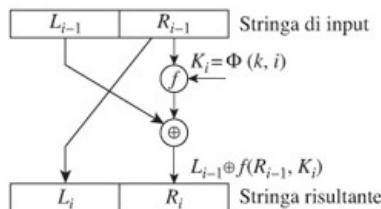
RC4 è una cifratura a flusso, ampiamente usata, che utilizza una sequenza chiave cioè un flusso di bit pseudorandom. Si usa un generatore di flussi pseudorandom che è inizializzato usando una chiave generata dall'algoritmo di key scheduling. È veloce, poiché richiede solo tra 8 e 16 operazioni macchina per generare 1 byte nella sequenza chiave. È usato nel protocollo *Wired Equivalent Privacy* (WEP) per la sicurezza delle reti wireless e nel suo successore, il protocollo *Wi-Fi Protected Access* (WAP), e nel protocollo *Secure Socket Layer* (SSL) per Internet. È stato mostrato che l'algoritmo di key scheduling di RC4 possiede punti deboli, che provocavano intrusioni nei protocolli WEP e WAP. Comunque, il suo uso nel protocollo SSL è considerato sicuro perché il protocollo SSL stesso genera la chiave usata per inizializzare il generatore di flussi pseudorandom.

### Data Encryption Standard (DES)

Il DES è stato sviluppato da IBM per il National Bureau of Standards. Esso usa una chiave a 56 bit per cifrare blocchi di dati a 64 bit ed è pertanto un cifrario a blocchi. Per superare il problema della scarsa diffusione, il DES fornisce una modalità di *cifratura a blocchi a cascata* (chipher block chaining – CBC). In tal modo, il primo blocco del testo in chiaro è combinato con un vettore iniziale tramite l'operazione di OR esclusivo e poi viene cifrato. Il testo cifrato risultante viene poi combinato con un secondo blocco del testo in chiaro usando l'operazione di OR esclusivo, poi viene cifrato, e così via.

Il DES consta di tre passi: il passo iniziale di permutazione, il passo di trasformazione e il passo finale di permutazione. Il passo di trasformazione è composto da 16 iterazioni. A ogni iterazione la stringa di input è sottoposta a una trasformazione complessa che comprende un'operazione di permutazione sulla stringa, che genera diffusione, e un'operazione di sostituzione tramite duplicazione e omissione di qualche bit, che genera confusione. La [Figura 15.6](#) illustra le operazioni eseguite a ogni iterazione. Alla prima iterazione, la stringa di input è il testo in chiaro. In tutte le altre iterazioni, la stringa di input è l'output della precedente iterazione. La stringa di input è divisa in due metà di 32 bit ciascuna. La metà di destra della stringa di input diventa la metà di sinistra della stringa risultato; viene, poi, eseguita una trasformazione complessa che coinvolge entrambe le metà (sinistra e destra) per ottenere la metà di destra della stringa risultato.

La trasformazione della metà di destra della stringa di input consta dei seguenti passi. Anzitutto è estesa a 48 bit permutando i suoi bit e duplicandone alcuni. Essa viene combinata con la chiave  $K_i$  usando un'operazione di OR esclusivo (funzione  $f$  in [Figura 15.6](#)), in cui la chiave  $K_i$  è derivata dalla permutazione della chiave di cifratura  $k$ , usando il passo di iterazione  $i$ . Il risultato di questa operazione è diviso in otto gruppi di 6 bit ciascuno. Ogni gruppo di 6 bit è l'input per una S-box che lo sostituisce con un gruppo di 4 bit. I risultati della sostituzione sono concatenati per ottenere una stringa a 32 bit che è permutata per ottenere un'altra stringa a 32 bit. Questa stringa è unita alla metà di sinistra della stringa di input, usando un'operazione di OR esclusivo per ottenere la metà di destra della stringa risultato. La S-box introduce confusione. La permutazione introduce diffusione, mentre l'operazione finale di OR esclusivo introduce confusione.



**Figura 15.6** Una iterazione in DES (che indica l'operazione di OR esclusivo).

Il DES effettua sia cifratura che decifratura utilizzando la stessa sequenza di passi, a parte il fatto che le chiavi sono utilizzate nell'ordine inverso durante la decifratura; cioè l'iterazione  $i$  usa la chiave  $K_{17-i}$  invece della chiave  $K_i$ . La chiave usata in DES di lunghezza 56 bit richiederà  $2^{55}$  tentativi in un attacco esaustivo, che, tra gli anni 1970 e 1980, era considerato un numero troppo grande per contrastare tali attacchi. D'altronde, l'uso di chiavi di lunghezza limitata rendono il DES vulnerabile agli attacchi che usano tecnologie moderne. Nel 1998, un messaggio cifrato con il DES fu decifrato in meno di 3 giorni tramite un elaboratore progettato per lo scopo. Nel 1999, un altro messaggio fu decifrato in meno di un giorno usando 100 000 PC su Internet. L'algoritmo *triple DES* fu, allora, pubblicato come standard ad interim in attesa che fosse adottato un nuovo standard. Esso effettuava tre iterazioni, ogni iterazione applicava l'algoritmo DES, usando una chiave diversa derivata dalla chiave di cifratura – la prima e la terza iterazione eseguivano la cifratura usando le loro chiavi, mentre la seconda iterazione eseguiva la decifratura usando la sua chiave. L'algoritmo poteva utilizzare in modo efficiente chiavi di lunghezza fino a 168 bit, il che lo rese sicuro per alcuni anni contro gli attacchi. Il nuovo standard chiamato *Advanced Encryption Standard* (AES) fu adottato nel 2001.

### Advanced Encryption Standard (AES)

AES è una variante del *Rijndael*, un algoritmo di cifratura compatto e veloce che utilizza

solo sostituzioni e permutazioni. AES usa, come dimensione del blocco, 128 bit e chiavi a 128, 192 o 256 bit, mentre Rijndael può usare qualsiasi dimensione per la chiave e per il blocco compresa nell'intervallo tra 128 e 256 bit tale che sia multiplo di 32 bit. Un blocco di testo in chiaro di 16 byte è trattato come una matrice  $4 \times 4$  byte detta *state*. È cifrata tramite molti cicli di operazioni, dove il numero di cicli dipende dalla lunghezza della chiave: pertanto sono necessari 10 cicli per chiavi a 128 bit, 12 cicli per chiavi a 192 bit, 14 cicli per chiavi a 256 bit. Ogni ciclo è composto dalle seguenti operazioni.

1. *Sostituzione di byte*: ogni byte della state è soggetto a una trasformazione non lineare applicata da una S-box.
2. *Shifting di righe*: le righe della state sono shiftate ciclicamente rispettivamente di 0, 1, 2 e 3.
3. *Mixing di colonne*: i 4 byte in una colonna sono sostituiti in modo tale che ogni byte risultante sia funzione di tutti i 4 byte nella colonna.
4. *Key addition*: una sottochiave, la cui dimensione è la stessa dimensione della state, è ricavata dalla chiave di cifratura usando un key schedule. La sottochiave e lo state sono viste come stringhe di bit e unite usando l'operazione di OR esclusivo. Se questo è l'ultimo ciclo, il risultato dell'operazione di OR esclusivo è un blocco del testo cifrato; altrimenti, è usata come state per il prossimo ciclo di cifratura.

Per far sì che la stessa sequenza di passi valga sia per la cifratura che per la decifratura, prima di cominciare il primo ciclo, viene eseguita una key addition e, nell'ultimo ciclo, viene saltato il passo mixing di colonne.

## 15.5 Autenticazione e sicurezza delle password

L'autenticazione è tipicamente eseguita tramite password, usando lo schema mostrato in [Figura 15.1](#). Per ogni utente registrato, il sistema memorizza una coppia del tipo (login id, *<info\_validazione>*) in una tabella delle password, dove *<info\_validazione>* =  $E_k$  (password). Per autenticare un utente, il sistema effettua la cifratura delle password dell'utente usando  $E_k$  e confronta il risultato con l'informazione di convalida memorizzata nella tabella delle password. Se corrispondono, l'utente si considera autentico.

Se un intruso ha accesso alla tabella delle password, potrà lanciare uno degli attacchi descritti precedentemente nel [Paragrafo 15.4.1](#) per determinare  $E_k$ . In alternativa, l'intruso può lanciare un attacco per risalire alla password di un utente. Nello schema descritto, se due utenti usano le stesse password, anche le forme cifrate delle loro password saranno identiche, il che faciliterà i tentativi di un intruso di individuare la password se la tabella delle password è visibile. Di conseguenza, la funzione di cifratura  $E$  richiede due parametri. Un parametro è la chiave di cifratura  $k$ , e l'altro parametro è una stringa derivata dalla login id dell'utente. In questo caso, password identiche hanno stringhe cifrate diverse.

Gli intrusi possono usare programmi di intercettazione delle password per scoprire le password degli utenti. Il loro compito è semplificato dalla tendenza degli utenti a usare password che non sono difficili da indovinare, come parole del dizionario o targhe di veicoli, o semplici sequenze da tastiera. Per account usati di rado, gli utenti spesso scelgono password semplici che sono facili da ricordare, in quanto non hanno file importanti in quell'account. Comunque, una password è proverbialmente il punto più debole nella catena di sicurezza. Ogni password intercettata fornisce a un intruso l'opportunità per lanciare ulteriori attacchi alla sicurezza. Di conseguenza, un gran numero di problemi legati alla sicurezza dipendono dall'uso di password semplici.

I sistemi operativi usano un insieme di tecniche per fronteggiare gli attacchi alle password ([Tabella 15.6](#)). L' *invecchiamento delle password* limita l'esposizione delle password agli intrusi, rendendo le password più sicure. Le password scelte dal sistema assicurano l'uso di password *robuste*, che non possono essere individuate da semplici tecniche come cercare nelle password, parti di nomi o parole del dizionario. Il loro uso obbligherà un intruso a utilizzare un attacco esaustivo per individuare una password, ovvero un attacco impraticabile.

| Tecnica                                                   | Descrizione                                                                                                                                                                                                                                                                                                                          |
|-----------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Invecchiamento delle password                             | Incoraggiare o costringere gli utenti a cambiare le loro password frequentemente, almeno una volta ogni 6 mesi. Ciò limita l'esposizione di una password agli attacchi degli intrusi.                                                                                                                                                |
| Password scelte dal sistema                               | L'amministratore di sistema utilizza una metodologia per generare e assegnare password <i>robuste</i> agli utenti. Non è consentito agli utenti cambiare tali password. Un intruso dovrà usare un attacco esaustivo per intercettare tali password.                                                                                  |
| Codifica delle password                                   | La forma cifrata delle password è memorizzata in un file di sistema; tuttavia, il testo cifrato delle password è visibile a tutti gli utenti del sistema. Un intruso può usare uno degli attacchi descritti nel Paragrafo 15.4.1 per trovare $E_k$ , oppure lanciare un attacco esaustivo per intercettare la password di un utente. |
| Codifica e occultamento delle informazioni sulle password | La forma cifrata delle password non è visibile ad alcuno all'interno o al di fuori del sistema. Di conseguenza, un intruso non può usare alcun attacco descritto nel Paragrafo 15.4.1.                                                                                                                                               |

**Tabella 15.6** Tecniche di SO per contrastare attacchi alle password.

Quando il file delle password cifrate è visibile all'interno e all'esterno del sistema, un utente registrato può usare un attacco del tipo testo in chiaro scelto per scoprire  $E_k$  cambiando la sua password ripetutamente e analizzando la sua forma cifrata. Tuttavia, un intruso dovrà usare un attacco del tipo solo testo cifrato per trovare  $E_k$  poiché non conosce alcuna informazione sulle password. Nascondendo il file delle password cifrate, un intruso, all'interno e all'esterno del sistema, non avrà l'opportunità di usare uno degli attacchi descritti nel Paragrafo 15.4.1, quindi dovrà affidarsi alla individuazione delle password singole, che ha scarsa possibilità di successo nel caso in cui vengono usate password robuste.

Il sistema operativo Unix utilizza il DES per la cifratura delle password. Linux utilizza un *message digest*, che è un valore di hash a 128 bit o 160 bit ottenuto applicando una funzione di hash one-way a una password. Questa tecnica ha come varianti MD2, MD4 e MD5. Linux usa MD5. Sia Unix che Linux prevedono l'opzione delle shadow password. Scegliendo tale opzione, il testo cifrato delle password è memorizzato in un file shadow accessibile solo da root.

## 15.6 Strutture di protezione

Una *struttura di protezione* è il classico nome per il database delle autorizzazioni di cui abbiamo discusso nel Paragrafo 15.1 e che abbiamo illustrato in [Figura 15.1](#). Esso contiene informazioni su quali utenti possono accedere, a quali file e in che maniera. Cominciamo descrivendo la natura delle informazioni contenute in una struttura di protezione e come tale informazione viene utilizzata per implementare la protezione. Successivamente in questo paragrafo, discuteremo gli elementi chiave nell'organizzazione delle strutture di protezione.

Ricordiamo dal Paragrafo 15.1 che un *privilegio di accesso* a un file è il diritto a effettuare una specifica forma di accesso al file, per esempio un accesso di lettura o di scrittura. Un utente può avere uno o più privilegi di accesso a un file, per esempio può essere autorizzato solo a leggere un file, oppure può leggere e scrivere il file ma non eseguirlo. Un *descrittore di accesso* è una rappresentazione di un insieme di privilegi di accesso a un file. Le *informazioni per il controllo di accesso* a un file sono una raccolta di descrittori di accesso; essa rappresenta i privilegi di accesso a un file di tutti gli utenti nel sistema.

Usiamo le notazioni  $r$ ,  $w$  e  $x$  per rappresentare i privilegi di accesso in lettura, scrittura ed esecuzione sui dati o programmi. Un descrittore di accesso può essere rappresentato come un insieme di privilegi di accesso, per esempio l'insieme  $\{r, w\}$  indica i privilegi sia

di lettura che di scrittura su file. Per semplicità in questo capitolo useremo l'insieme dei privilegi; tuttavia, un insieme di rappresentazione è costoso in termini di richiesta di memoria e di efficienza di accesso, quindi i sistemi operativi, in realtà, usano uno schema codificato a bit per i descrittori di accesso. In questo schema, un descrittore di accesso è una stringa di bit, in cui ogni bit indica la presenza o l'assenza di uno specifico privilegio di accesso. Per esempio, in un SO che utilizza solo tre privilegi di accesso *r*, *w* e *x*, il descrittore di accesso 110 potrebbe essere usato per indicare che sono presenti i privilegi di lettura e scrittura, mentre quello di esecuzione è assente.

Come detto nel Paragrafo 15.1, le informazioni per il controllo degli accessi per un file *alpha* vengono create e usate come segue.

1. Quando un utente A crea il file *alpha*, specifica le informazioni per il controllo degli accessi su di esso. Il file system lo memorizza nella struttura di protezione.
2. Quando un utente X effettua il login, è autenticato. Il servizio di autenticazione genera un token di autenticazione, tipicamente un user id. Quando un processo, iniziato da un utente X, ha la necessità di aprire o accedere a un file *alpha*, il suo token di autenticazione è passato al file system.
3. Il file system usa il token di autenticazione per trovare, nella struttura di protezione, i privilegi di accesso dell'utente X per il file *alpha*, e verifica se il tipo di accesso richiesto dal processo è compatibile con i privilegi di accesso.

L'organizzazione della struttura di protezione influenza due aspetti chiave della protezione: il livello di discrezionalità che può esercitare il proprietario di un file nel passo 1, specificando quali altri utenti possono accedere al file, e quanto efficacemente può essere implementato il controllo di protezione del passo 3. Discutiamo queste problematiche nei paragrafi successivi.

### 15.6.1 Granularità della protezione

La *granularità della protezione* rappresenta il grado di discrezionalità che il proprietario di un file può esercitare in relazione alla protezione dei file. Definiamo tre livelli di granularità nella [Tabella 15.7](#).

| Granularità               | Descrizione                                                                                                                                |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| Protezione a grana grossa | Privilegi di accesso a un file specificabili solo per gruppi di utenti. Ogni utente in un gruppo ha identici privilegi di accesso al file. |
| Protezione a gran media   | Privilegi di accesso a un file specificabili individualmente per ciascun utente nel sistema.                                               |
| Protezione a grana fine   | Privilegi di accesso a un file specificabili per un processo o per una fase dell'esecuzione di un processo.                                |

**Tabella 15.7** Granularità della protezione.

La *protezione a grana grossa* comporta che gli utenti siano riuniti in gruppi e che i privilegi di accesso siano specificati per un gruppo di utenti, mentre la *protezione a grana media* comporta che il proprietario di un file possa specificare i privilegi di accesso, individualmente, per ciascun utente del sistema. La *protezione a grana fine* consente di specificare i privilegi di accesso per un processo o per differenti fasi dell'esecuzione di un processo. In questo modo, processi diversi, creati dallo stesso utente, possono avere diversi privilegi di accesso a un file, oppure lo stesso processo può avere diversi privilegi di accesso a un file in tempi diversi. Ciò aiuta a garantire la riservatezza delle informazioni (Paragrafo 15.1.1).

Gli utenti richiedono protezione a grana fine o media. Tuttavia, una protezione di questo tipo comporta strutture di protezione di grandi dimensioni. Questo è il motivo per cui i sistemi operativi ricorrono alla protezione a grana grossa.

### 15.6.2 Matrice di controllo degli accessi

Una *matrice di controllo degli accessi* (ACM) è una struttura di protezione che fornisce accesso efficiente sia ai privilegi di accesso degli utenti ai vari file sia alle informazioni

per il controllo degli accessi ai file. Ogni elemento dell'ACM contiene i privilegi di accesso di un utente a un file. Ogni utente ha una riga nell'ACM, mentre ogni file ha una colonna. In questo modo, una riga nell'ACM descrive i privilegi di accesso di un utente per tutti i file nel sistema, e ogni colonna descrive le informazioni per il controllo degli accessi a un file. Quando un utente  $u_i$  vuole accedere a un file  $f_j$ , si può accedere all'elemento ACM( $u_i, f_j$ ), in maniera efficiente, per validare il tipo di accesso che  $u_i$  sta effettuando. La [Figura 15.7](#) mostra un'ACM. L'utente Jay ha i privilegi di accesso {lettura, scrittura} per beta ma solo il privilegio di {lettura} per alpha.

|          |        | File →  | alpha | beta | gamma |
|----------|--------|---------|-------|------|-------|
| Utenti ↓ | Jay    | {r}     | {r,w} |      |       |
|          | Anita  | {r,w,x} |       | {r}  |       |
|          | Sheila |         |       | {r}  |       |

↑  
Informazioni di controllo  
degli accessi alpha

← Privilegi di accesso  
di Anita

**Figura 15.7** Matrice di Controllo degli Accessi (ACM).

L'ACM fornisce protezione a grana media. È, tuttavia, di grandi dimensioni perché un SO ha un elevato numero di utenti e contiene un notevole numero di file. Di conseguenza, deve essere riservata un'ampia area di memoria per contenere l'ACM, o parti di essa, in memoria durante il funzionamento del sistema. I sistemi operativi usano due approcci per ridurre la dimensione delle informazioni per il controllo degli accessi. Nel primo approccio, il numero di righe viene ridotto assegnando privilegi di accesso a gruppi di utenti piuttosto che a utenti singoli. Questo approccio conserva il vantaggio di base dell'ACM, cioè accesso efficiente sia ai privilegi di accesso degli utenti che alle informazioni per il controllo degli accessi ai file. Esso comporta, comunque, protezione a grana grossa perché tutti gli utenti in un gruppo hanno gli stessi privilegi di accesso a un file.

Il secondo approccio per ridurre la dimensione della struttura di protezione sfrutta il fatto che un utente tipico possiede privilegi di accesso solo su pochi file. Quindi, la maggior parte degli elementi in un'ACM contengono elementi nulli; è possibile così risparmiare spazio organizzando le informazioni relative alla protezione sotto forma di liste contenenti solo i privilegi di accesso non nulli. Questo approccio non impatta sulla granularità della protezione; compromette, però, l'efficienza di accesso della struttura di protezione. Nei seguenti paragrafi presenteremo due strutture di protezione organizzate come liste.

### 15.6.3 Liste di controllo degli accessi (ACL)

La *lista di controllo degli accessi* (ACL) di un file è una rappresentazione delle informazioni per il controllo degli accessi; contiene gli elementi non nulli relativi alla colonna del file nell'ACM. È memorizzata come una lista di coppie del tipo (*id\_utente, privilegi di accesso*). La [Figura 15.8](#) mostra le liste di controllo degli accessi per i file alpha, beta e gamma di [Figura 15.7](#). L'ACL per alpha è `[(Jay, {read}), (Anita, {read, write, execute})]`, che indica che l'utente Jay può solo leggere il file alpha, mentre Anita può leggere, scrivere o eseguire il file. All'utente Sheila non è consentito alcun tipo di accesso al file alpha, poiché l'ACL di alpha non contiene un elemento per Sheila.

| Nome<br>del file | Lista di controllo<br>degli accessi (ACL)     |
|------------------|-----------------------------------------------|
| alpha            | <code>[(Jay, {r}), (Anita, {r, w, x})]</code> |
| beta             | <code>[(Jay, {r, w})]</code>                  |
| gamma            | <code>[(Anita, {r}), (Sheila, {r})]</code>    |

**Figura 15.8** Liste di controllo degli accessi (ACLs).

Anche se l'uso di un'ACL elimina la necessità di memorizzare i privilegi di accesso

nulli, la presenza di un gran numero di utenti in un sistema comporta ACL di grandi dimensioni, e pertanto uno spreco di spazio nel file system. Anche l'overhead di tempo è alto, poiché l'ACL deve essere scandita per la validazione di un accesso a un file. Utilizzando la protezione a grana grossa, si può risparmiare tempo sia di memoria che di CPU, specificando le informazioni di protezione per gruppi di utenti piuttosto che per utenti singoli. Per esempio, se gli utenti Jay e Anita appartenessero allo stesso gruppo di utenti, l'ACL del file alpha conterrebbe una singola coppia. Sarebbe quindi più facile determinare se Jay può accedere ad alpha; però, sia Jay che Anita avrebbero gli stessi privilegi di accesso. Un'ACL di questo tipo potrebbe essere piccola abbastanza per essere memorizzata nell'elemento della directory relativa a un file.

#### 15.6.4 Capability list (C-list)

Una *capability list* (C-list) rappresenta i privilegi di accesso di un utente a vari file nel sistema; essa contiene le entry, non nulle, che avrebbe contenuto la riga dell'utente nell'ACM. Ogni entry nella C-list è una *capability*, che rappresenta i privilegi di accesso a un file; è costituita da una coppia del tipo *(file\_id, privilegi\_accesso)*. La [Figura 15.9](#) mostra una C-list per l'utente Anita di [Figura 15.7](#). Anita può leggere, scrivere o eseguire il file alpha e può leggere il file gamma. Anita non ha privilegi di accesso sul file beta, poiché non esiste alcuna entry per beta nella C-list. Le C-list sono, di solito, di piccole dimensioni; questa caratteristica limita l'overhead di spazio e tempo quando si utilizzano per la protezione dei file. Discuteremo come le capability sono utilizzate in un elaboratore nel [Paragrafo 15.7](#).

---

|                    |
|--------------------|
| (alpha, {r, w, x}) |
| (gamma, {r})       |
|                    |

---

**Figura 15.9** Capability list per l'utente Anita.

#### 15.6.5 Dominio di protezione

La matrice di controllo degli accessi, la lista di controllo degli accessi o la capability list servono a conferire privilegi di accesso agli utenti. Questa soluzione è utile per la *segretezza*, obiettivo della sicurezza e della protezione in quanto solo gli utenti autorizzati possono accedere a un file. Tuttavia, la *riservatezza*, obiettivo della sicurezza e della protezione, richiede che l'informazione sia usata solo per gli scopi stabiliti ([Paragrafo 15.1.1](#)), e questo requisito potrebbe essere violato nel seguente modo: un privilegio di accesso su un file è stato concesso a un utente poiché *un* processo iniziato dall'utente lo richiede. Però, ogni altro processo, iniziato dall'utente, ottiene lo stesso privilegio di accesso sul file; uno di questi processi può accedere al file in modo non consentito, violando, così, il requisito di riservatezza. L'esempio seguente illustra come può essere pregiudicata la riservatezza dell'informazione.

##### Esempio 15.2 - Violazione della riservatezza

Un utente  $u_i$  ha un privilegio di esecuzione per il programma *invest* di proprietà di un altro utente  $u_j$ . Quando  $u_i$  esegue *invest*, *invest* opera come processo iniziato dall'utente  $u_i$ . Esso può accedere a qualsiasi file per il quale l'utente  $u_i$  ha un privilegio di accesso, inclusi i file che non hanno niente in comune con gli investimenti. Se  $u_j$  volesse, potrebbe codificare *invest* per ottenere la lista della directory corrente di  $u_i$  e copiare o modificare qualcuno dei file trovati in essa.

La violazione della riservatezza solleva un'importante questione di affidabilità: la correttezza dei dati dipende non solo dalle operazioni corrette da parte dei processi che notoriamente vi accedono, ma anche dall'innocuità degli accessi operati dai processi che non si presuppone vi accedano. Il concetto di *dominio di protezione* è usato per evitare violazioni della riservatezza. Possiamo pensare a un dominio di protezione come a un "ambiente di esecuzione" teorico: i privilegi di accesso sono attribuiti a un dominio di protezione piuttosto che a un utente o al suo processo. Un processo, che agisce "all'interno" di un dominio di protezione, può accedere ai file per i quali il dominio di

protezione possiede i privilegi di accesso. Questa soluzione facilita l'implementazione del principio need-to-know con una granularità a grana fine: a un processo viene consentito di agire all'interno di un dominio di protezione solo se ha bisogno di accedere ai file per i quali il dominio di protezione ha i privilegi di accesso. L'esempio seguente illustra come questo approccio assicura la riservatezza delle informazioni.

### Esempio 15.3 - Domini di protezione

La Figura 15.10 mostra tre domini di protezione. Il dominio  $D_1$  ha i privilegi di lettura e scrittura per i file `personal` e `finance`, mentre  $D_2$  ha solo il privilegio di lettura per `finance`. Il dominio  $D_3$  ha i privilegi lettura e scrittura per i file `memos` e `notes` e il privilegio di lettura per il file `project`. Quindi, i domini  $D_1$  e  $D_2$  si intersecano mentre il dominio  $D_3$  è disgiunto da entrambi.

| File →       |  | personal | finance    | memos      | notes      | project    |
|--------------|--|----------|------------|------------|------------|------------|
|              |  | $D_1$    | $\{r, w\}$ | $\{r, w\}$ |            |            |
| $\downarrow$ |  | $D_2$    |            | $\{r\}$    |            |            |
|              |  | $D_3$    |            |            | $\{r, w\}$ | $\{r, w\}$ |

**Figura 15.10** Domini di protezione.

L'utente  $u_i$  avvia i tre processi chiamati `self`, `invest` e `job_related` nei domini  $D_1$ ,  $D_2$  e  $D_3$ , rispettivamente. Quindi `invest` può accedere solo al file `finance`, e può solo leggerlo.

Se il SO non usasse i domini di protezione, l'utente  $u_i$  necessiterebbe dei privilegi di lettura e scrittura per i file `personal`, `finance`, `memos` e `notes` e il privilegio di lettura sul file `project`. Se l'utente  $u_i$  avesse eseguito il programma `invest` dell'Esempio 15.2, di proprietà dell'utente  $u_j$ , `invest` avrebbe potuto modificare i file `personal`, `finance`, `memos` e `notes` dell'utente  $u_i$ .

La riservatezza può essere migliorata consentendo a un processo l'accesso ad alcune risorse solo durante specifiche fasi della sua esecuzione. Questo meccanismo è facilitato consentendo a un processo, sotto certe condizioni, di cambiare il suo dominio di protezione durante l'elaborazione. Usando la proprietà di cambio del dominio, l'utente  $u_i$  dell'Esempio 15.3 dovrebbe essere in grado di utilizzare un solo processo per eseguire alcuni calcoli personali, prendere decisioni sugli investimenti usando il programma `invest` il cui proprietario è l'utente  $u_j$ , e scrivere qualche memo e note usando un pacchetto standard. Il processo sarà iniziato nel dominio  $D_1$ . Dopo l'esecuzione delle elaborazioni personali in questo dominio, il processo cambierà e passerà al suo dominio  $D_2$  e chiamerà il programma `invest`, in modo tale che `invest` possa vedere solo i dettagli finanziari di  $u_i$ , ma non modificare questi o qualche altra informazione personale di  $u_i$ . Il processo passerà successivamente al dominio  $D_3$  per scrivere memo e note, usando il pacchetto standard. Descriviamo le caratteristiche del cambio di dominio di protezione nei sistemi operativi Unix e Multics nel Paragrafo 15.9.

## 15.7 Capabilities

Dennis and Van Horn (1966) proposero il concetto di *capability* da utilizzare nella condivisione e nella protezione. Una capability è un token che rappresenta alcuni privilegi di accesso a un oggetto, dove un oggetto è qualsiasi entità hardware o software nel sistema, per esempio una stampante laser, una CPU, un file, un programma o una struttura dati di un programma. Un processo ha una capability. Questo possesso fornisce al processo il diritto di accedere all'oggetto in modo compatibile ai privilegi di accesso descritti nella capability.

La Figura 15.11 mostra il formato di una capability. È composta da due campi, *id dell'oggetto* e *privilegi di accesso*. Ogni oggetto ha un unico id dell'oggetto nel sistema. Il campo privilegi di accesso tipicamente contiene un descrittore di accesso codificato a bit. Un processo può avere più capability. Queste sono memorizzate nella *capability list* (C-list) discussa precedentemente nel Paragrafo 15.6.4.



**Figura 15.11** Formato di una capability.

Quando un processo  $P_i$  crea un oggetto  $O_i$ , il SO crea una capability per  $O_i$  che contiene l'intero insieme dei privilegi di accesso definiti nel sistema e passa tale capability a  $P_i$ . Usando questa capability,  $P_i$  può richiedere al SO di creare un *sottoinsieme di capability* per  $O_i$  che contiene alcuni privilegi di accesso. Esso può, inoltre, fare copie della capability per  $O_i$  che ha ricevuto dal SO. Cioè nel sistema possono esistere più capability per  $O_i$ . Il processo  $P_i$  può condividere l'oggetto  $O_i$  con altri processi trasferendo le capability per  $O_i$  agli altri processi. Quindi, ogni processo ha capability per gli oggetti di sua proprietà e delle capability passate da altri processi. Tutte queste capability sono ottenute in maniera lecita, cioè nessuna può essere sottratta o creata illecitamente da un processo. Questo è il motivo per cui la capability è spesso descritta come un token non riproducibile che conferisce privilegi di accesso sulle proprie proprietà.

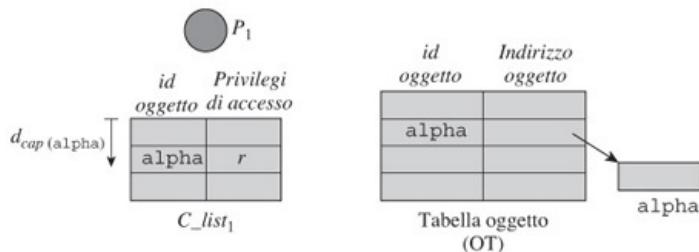
Usiamo la notazione  $Cap_k(obj_i)$  per riferirci a una capability per  $obj_i$ . Il pedice di  $Cap$  è usato semplicemente per distinguere tra le differenti capability per un oggetto. Non ha altro significato. Per semplicità, omettiamo il pedice in contesti in cui c'è una singola capability di un oggetto.

### 15.7.1 Sistemi basati su capability

Un sistema basato su capability implementa l'indirizzamento e la protezione per *tutti* gli oggetti nel sistema, andando dagli oggetti long-life come i file a quelli short-life come le strutture dati e le copie dei programmi in memoria. Molti sistemi basati sulle capability furono costruiti a scopo di ricerca; l'Intel iapx-432 era un sistema commerciale basato sulle capability.

La [Figura 15.12](#) è un diagramma dell'indirizzamento di oggetti basato sulla capability. Il sistema non associa esplicitamente la "memoria" ai processi; esso associa le C-list ai processi. Ogni oggetto ha un unico id. La *object table* (OT) è una tabella globale all'interno del sistema, che contiene le informazioni sulla posizione di tutti gli oggetti nel sistema. Il campo *object address* di una entry della OT indica l'indirizzo dell'oggetto in memoria primaria e secondaria del computer. L'accesso a un oggetto è implementato come segue: un processo  $P_1$  esegue un'operazione  $<op_i>$  su di un oggetto tramite un'istruzione del tipo:

$$<op_i> d_{Cap(obj_i)} \quad (15.1)$$



**Figura 15.12** Indirizzamento basato sulla capacità.

dove  $d_{Cap(obj_i)}$  è lo scostamento di  $Cap(obj_i)$  nella C-list di  $P_1$ . La CPU individua la capability nella C-list di  $P_1$  tramite lo scostamento e verifica che l'operazione  $<op_i>$  sia compatibile con i privilegi di accesso nella capability. L'id dell'oggetto nella capability, cioè  $\alpha$ , è utilizzato per localizzare la entry di  $\alpha$  nella OT e l'indirizzo dell'oggetto trovato è usato per implementare  $<op_i>$ . L'indirizzamento basato sulla capability può

essere reso più efficiente tramite l'uso di buffer analoghi a quelli per la traduzione degli indirizzi (Paragrafo 12.2.2) e speciali memorie cache per la traduzione degli indirizzi.

Le capability in una C-list possono essere usate per accedere agli oggetti presenti nel sistema, per esempio in memoria o su disco; la posizione di un oggetto è “irrilevante” per un processo. Questa caratteristica permette al SO di spostare gli oggetti all'interno della memoria per una migliore gestione della memoria stessa, oppure spostarli tra la memoria e il disco per ottenere prestazioni d'accesso più efficaci, senza influire sul modo in cui un programma accede agli oggetti. Quindi, si può rendere uniforme l'accesso agli oggetti persistenti, come i file, e a quelli non persistenti, come le strutture dati.

### **Operazioni sugli oggetti e capability**

Al momento della creazione di un processo, il SO può assegnargli alcune capability; il processo può inoltre ereditarne dal processo padre. Quando il processo esegue l'operazione “create a new object”, la CPU crea un nuovo oggetto e un nuovo elemento nella OT. Il valore dell'elemento è impostato con l'id e l'indirizzo del nuovo oggetto creato. Crea, dunque, una capability contenente l'intero insieme dei privilegi d'accesso per l'oggetto e la mette nella C-list di  $P_i$ . Memorizza, inoltre,  $d_{Cap}(obj_i)$  in un registro della CPU. Il processo  $P_i$  salva i contenuti di questo registro per accessi futuri ad  $obj_i$ .

Tutte le operazioni eseguite da un processo sono subordinate ai privilegi di accesso contenuti nella sua C-list. L'operazione di creazione di un oggetto può essa stessa essere subordinata a un privilegio di accesso; in tal caso, il SO conferirebbe questo privilegio di accesso tramite una delle capability di default attribuite a ciascun processo. La [Tabella 15.8](#) elenca le operazioni che un processo può eseguire sugli oggetti e le capability. Quindi, un processo può creare, modificare, distruggere, copiare o eseguire un oggetto se ha una capability con gli appropriati privilegi di accesso.

|                          |                                                                                                                                                                                                                                                                                                                                               |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Operazioni su oggetti    | <ul style="list-style-type: none"> <li>• Creare un oggetto</li> <li>• Leggere o modificare l'oggetto</li> <li>• Distruggere l'oggetto</li> <li>• Copiare l'oggetto</li> <li>• Eseguire l'oggetto</li> </ul>                                                                                                                                   |
| Operazioni su capability | <ul style="list-style-type: none"> <li>• Fare una copia della capability</li> <li>• Creare un “sottoinsieme” di capability</li> <li>• Usare la capability come parametro in una funzione/chiamata a procedura</li> <li>• Trasferire la capability per l'utilizzo da parte di un altro processo</li> <li>• Cancellare la capability</li> </ul> |

**Tabella 15.8** Operazioni consentite sugli oggetti e capability.

Le operazioni su una capability sono subordinate ai privilegi di accesso contenuti in essa. Per esempio, un processo può essere in grado di creare un sottoinsieme di capability di  $Cap(obj_i)$  solo se  $Cap(obj_i)$  contiene il privilegio di accesso “Creare un sottoinsieme di capability”. Questa caratteristica controlla le operazioni che i processi possono eseguire sulle capability. La condivisione di oggetti si verifica quando un processo trasferisce a un altro processo una capability per un oggetto. Il processo che riceve la capability la inserisce nella sua C-list. La condivisione è implicita nel fatto che entrambe le C-list contengono una capability per l'oggetto. La protezione è implicita nel fatto che queste capability possono conferire differenti privilegi di accesso sui processi.

### **Protezione delle capability**

I metodi di protezione che utilizzano le capability si basano sull'assunzione fondamentale che le capability non possono essere contraffatte o alterate. Questa assunzione non sarebbe valida se un processo potesse accedere alla sua C-list e modificare le capability presenti in essa. Per esempio, il processo  $P_1$  di [Figura 15.12](#) potrebbe alterare il campo dei privilegi di accesso della capability per  $\alpha$  al fine di acquisire un privilegio di accesso “write” e, quindi, usare la capability modificata per modificare l'oggetto  $\alpha$ . Un'alterazione delle capability di questo tipo può essere evitata assicurando che non possono essere eseguite operazioni arbitrarie sulle capability. Ciò è implementato usando due approcci: le architetture tagged e i segmenti di capability.

In un sistema con *architettura tagged*, la rappresentazione a run-time di un'entità è composta da due campi: un campo *etichetta (tag)* e un campo valore. Il campo tag descrive il tipo di entità. La CPU è progettata per eseguire, su un'entità, solo quelle operazioni compatibili con il suo tag. In questo modo, solo le sei operazioni menzionate in [Tabella 15.8](#) possono essere eseguite su una capability, il che assicura che una capability non può essere alterata. In un sistema che utilizza i *segmenti di capability*, gli oggetti dati e le loro capability sono memorizzati in differenti segmenti e le istruzioni nella CPU sono progettate per accedere ai loro operandi da un segmento opportuno. Solo le sei operazioni menzionate in [Tabella 15.8](#) prenderanno i loro operandi dal segmento di capability. In questo modo, non possono essere eseguite operazioni arbitrarie sulle capability.

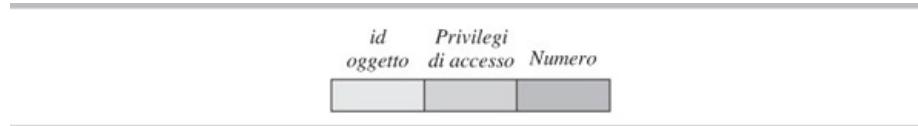
### 15.7.2 Capability software

Il SO per un sistema non basato sulle capability può implementare le capability software. La soluzione per gli oggetti e le capability può essere analoga a quella mostrata in [Figura 15.12](#). Tuttavia, la manipolazione e la protezione degli oggetti non può essere eseguita dalla CPU del sistema; ora, è eseguita da una componente del kernel chiamata *gestore di oggetti (object manager)* (OM). Un programma indica le sue richieste di manipolazione degli oggetti all'object manager effettuando una chiamata OM ( $<op_i>$ ,  $Cap(obj_i)$ ). Questa chiamata ha lo stesso effetto dell'Istruzione 15.1. L'object manager implementa  $<op_i>$  solo se  $Cap(obj_i)$  contiene i privilegi di accesso necessari per eseguirla.

Vanno analizzati due elementi importanti nelle capability software. In prima istanza, un processo può essere in grado di bypassare la soluzione di protezione basata sulla capability accedendo agli oggetti ed essere, così, in grado di alterare o creare capability. Inoltre, possiamo evitare che un processo manipoli oggetti senza passare tramite l'object manager? Un modo è nascondere gli oggetti dalla vista dei processi utente cifrando la tabella degli oggetti (object table). In questo modo, i processi non conosceranno la posizione degli oggetti. Di conseguenza, dovranno dipendere dall'object manager per manipolare gli oggetti. L'alterazione delle capability può essere evitata, anche, tramite la cifratura. Come esempio, descriviamo una versione semplificata dello schema di protezione delle capability usato nel sistema operativo distribuito Amoeba.

#### Capability in Amoeba

Nel momento in cui si crea un oggetto  $obj_i$  gli viene assegnata una chiave di cifratura  $key_i$ . La chiave di cifratura è memorizzata nell'elemento della object table relativo a  $obj_i$ . La capability di Amoeba ha il formato mostrato in [Figura 15.13](#). Il campo *numero* contiene le informazioni usate per proteggere la capability. Una capability per  $obj_i$  può essere creata usando la seguente procedura:



**Figura 15.13** Una capability sul modello Amoeba.

1. i campi *id dell'oggetto* e i *privilegi di accesso* della capability sono impostati in maniera appropriata;
2. la chiave di cifratura  $key_i$  è ottenuta dall'elemento dell'object table relativo a  $obj_i$ . I contenuti del campo *privilegi di accesso* sono concatenati con  $key_i$  e la stringa risultante è cifrata usando  $key_i$ . Denotiamo questa operazione con  $E_{key_i}(\text{privilegi di accesso. } key_i)$ , dove  $“.”$  rappresenta la concatenazione. Il risultato del passo di cifratura è memorizzato nel campo *numero* della capability.

Per manipolare  $obj_i$ , un processo deve inviare una capability per  $obj_i$  all'object manager. L'object manager verifica la validità di tale capability in questo modo:

1. la chiave di cifratura  $key_i$  è ottenuta dalla entry nella object table di  $obj_i$ ;
2. la stringa *privilegi di accesso* è ottenuta dalla capability e  $E_{key_i}(\text{privilegi di accesso. } key_i)$ .

$key_i$ ) è confrontata con il campo *numero* della capability.

Il confronto nel passo 2 fallisce se il campo *object id* oppure *privilegi di accesso* della capability è stato alterato, quindi l'object manager termina il processo se il confronto fallisce.

### **Confronto con sistemi basati su capability**

Il punto di forza delle capability software – la loro indipendenza dall'hardware sottostante – è anche la loro maggiore debolezza. Le operazioni come la creazione di sottoinsiemi di capability, che sono eseguite dall'hardware del sistema basato su capability, devono essere eseguite dal software. Ognuna di queste effettua una chiamata di sistema per invocare l'object manager. Inoltre, la prevenzione delle alterazioni richiede la validazione di una capability prima dell'uso. Queste richieste comportano un overhead di tempo sostanziale.

## **15.7.3 Aree critiche nell'uso delle capability**

L'uso delle capability pone tre problemi pratici:

- *necessità per il garbage collection*: quando può essere distrutto un oggetto?
- *confine delle capability*: come assicurare che i processi non trasferiscono le capability ad altri processi in modo indiscriminato?
- *revoca delle capability*: come revocare una capability o i privilegi di accesso da essa conferiti?

### **Garbage collection**

Il proprietario di un oggetto può preparare un sottoinsieme di capability per un oggetto e trasferirle ad altri processi in modo che essi possano accedere all'oggetto. Prima di distruggere un tale oggetto, il proprietario deve sapere che nessun processo lo sta attualmente usando. Si può ottenere tale informazione solo tramite la sincronizzazione tra il proprietario dell'oggetto e tutti gli utenti di un oggetto. Questo approccio è impraticabile quando c'è un'alta percentuale di creazione e utilizzo di oggetti, oppure quando vengono condivisi oggetti con tempi di vita elevati. Possono sorgere due problemi se vengono distrutti gli oggetti senza raccogliere tali informazioni. Possono rimanere puntatori "pendenti" – ossia, un oggetto può essere distrutto mentre esiste ancora qualche sua capability – oppure un oggetto può esistere anche dopo che le sue capability sono state distrutte. Evitare entrambe queste situazioni richiede l'uso di costose tecniche di garbage collection.

### **Confine delle capability**

Il confine comporta restrizioni sull'uso di una capability da parte di un insieme di processi. La mancanza di un confine implica la proliferazione delle capability in tutto il sistema dovuto al trasferimento indiscriminato di capability. Ciò complica la funzione di garbage collection e prolunga la vita di un oggetto. Può, inoltre, invalidare il meccanismo di protezione violando il principio *need-to-know*. Ciò può essere realizzato rendendo il passaggio di una capability un diritto di accesso: se il processo  $P_i$  elimina il diritto di accesso "pass" nella capability quando la trasferisce a  $P_j$ ,  $P_j$  non sarà in grado di passare la capability a nessun altro processo.

### **Revoca delle capability**

La revoca di tutte le capability per un oggetto è il problema più difficile in un sistema basato sulle capability, poiché non c'è modo di sapere quali processi hanno le capability per l'oggetto e non esiste un metodo per invalidare una capability. Tuttavia, la revoca è possibile in caso di capability software perché sono protette attraverso la cifratura. In Amoeba, tutte le capability esistenti di un oggetto saranno invalidate quando viene cambiata la chiave di cifratura assegnata all'oggetto. Per revocare alcune capability di un oggetto in maniera selettiva, il proprietario può invalidare tutte le capability dell'oggetto cambiando la chiave di cifratura e, poi, attribuire nuove capability soltanto ad altri processi. Tuttavia, questa è un'operazione costosa e intrusiva; infatti, nel caso in cui fosse necessario revocare una qualsiasi capability su un oggetto, tutti i processi in possesso di una capability nello stesso sarebbero coinvolti nell'operazione.

## 15.8 Classificazione della sicurezza informatica

Una politica di sicurezza specifica i ruoli delle entità – sia individui sia programmi – nell'assicurare che le risorse di un sistema siano utilizzate in maniera legittima. Nella terminologia di [Figura 15.1](#), una politica di sicurezza specifica i ruoli degli amministratori di sistema e i programmi utilizzati per amministrare i database di autenticazione e autorizzazione e i ruoli dei programmi del SO che costituiscono i servizi di autenticazione e autorizzazione.

Idealmente, sarebbe possibile provare che le politiche di sicurezza non possono essere compromesse. Tuttavia, tali prove sono difficili per le ragioni menzionate nel Paragrafo 15.3, quindi gli sviluppatori di sistemi devono usare altri mezzi per testare le misure di sicurezza dei sistemi. Questi mezzi tipicamente consistono nell'implementazione, da parte del sistema, del *controllo degli accessi* per assicurare che le risorse siano utilizzate in modo legittimo, e in una *auditing capability*, che mantiene informazioni su come un evento, relativo alla sicurezza, è stato gestito dall'entità opportunamente preposta.

Il Dipartimento della Difesa americano è avvalso della collaborazione del Trusted Computer System Evaluation Criteria (TCSEC) per determinare quanto un sistema sia conforme ai requisiti di sicurezza e protezione. Questi criteri classificano i sistemi in quattro divisioni e vari livelli all'interno di ciascuna divisione ([Tabella 15.9](#)). La classificazione considera che un sistema può essere partizionato in due parti: la *trusted computing base* (TCB) è quella parte del suo hardware, software e firmware che implementa le funzionalità legate alla sicurezza del sistema; il resto del sistema non implementa alcuna funzione relativa alla sicurezza. La classificazione di un sistema dipende dal fatto che la sua TCB soddisfa i requisiti di una specifica divisione nella classificazione e in tutte le divisioni inferiori. La divisione D è la classificazione di sicurezza più bassa; è conferita ai sistemi che non riescono a soddisfare i requisiti di nessun'altra divisione.

| Divisione                              | Descrizione e livelli                                                                                                                                                                                                                                                                                                                     |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Protezione verificata (divisione A)    | Un sistema deve implementare metodi formali di verifica della sicurezza.                                                                                                                                                                                                                                                                  |
| Protezione obbligatoria (divisione B)  | Un sistema deve associare sensitivity label ai dati e ai programmi e implementare regole di controllo degli accessi obbligatorie tramite un <i>reference monitor</i> (RM). <ul style="list-style-type: none"><li>• B1: protezione di sicurezza con label</li><li>• B2: protezione strutturata</li><li>• B3: domini di sicurezza</li></ul> |
| Protezione discrezionale (divisione C) | Un sistema deve implementare la protezione need-to-know e fornire capacità di verifica (audit) per la verifica dei soggetti e la determinazione delle loro azioni (accountability). <ul style="list-style-type: none"><li>• C1: protezione di sicurezza discrezionale</li><li>• C2: protezione degli accessi controllata</li></ul>        |
| Protezione minimale (divisione D)      | Un sistema che fallisce i requisiti per una divisione più alta.                                                                                                                                                                                                                                                                           |

**Tabella 15.9** Criteri di valutazione dei sistemi sicuri (trusted).

La TCB di un sistema di elaborazione di divisione C ha tre capability chiave. Primo, permette a un utente di specificare quali altri utenti possono accedere ai dati o ai programmi di sua proprietà; esegue l'autenticazione degli utenti per fornire tale capability. Secondo, facilita l'auditing di eventi relativi alla sicurezza nel sistema mantenendo un record di eventi come i tentativi di autenticazione, apertura/chiusura file, azioni di amministratori di sistema, ecc. Terzo, fornisce la *object reuse protection* per assicurare che un altro utente non possa accedere accidentalmente a un dato utente. Ciò è implementato liberando la memoria allocata per un data object prima di restituirlo al pool della TCB degli oggetti liberi o della memoria libera. I livelli C1 e C2 della divisione

C corrispondono alle diverse granularità di protezione. Un sistema soddisfa il livello C2 della classificazione se un utente può identificare ogni singolo utente che può accedere ai file di sua proprietà; altrimenti il sistema soddisfa il livello C1. Quindi, un sistema che implementa la protezione a grana grossa otterrebbe un livello di classificazione C1 (Paragrafo 15.6.1).

Per ottenere una classificazione di tipo divisione B, un sistema deve assegnare una sensitivity label a tutti i dati e programmi che rispecchiano i loro livelli di sicurezza e protezione, e devono usare queste label per la validazione di ciascun accesso a un dato o programma, che è detta *controllo degli accessi obbligatorio*. Esso deve, inoltre, controllare la propagazione dei diritti di accesso. Lo sviluppatore di sistema deve fornire un modello della politica di sicurezza su cui si basa il TCB. Questo modello deve impiegare un *reference monitor* (RM) per validare ciascuna referenza a un dato o programma da parte di un utente o del suo processo. Il reference monitor dovrebbe essere a prova di alterazione e abbastanza piccolo da poter essere analizzato e testato nella sua completezza.

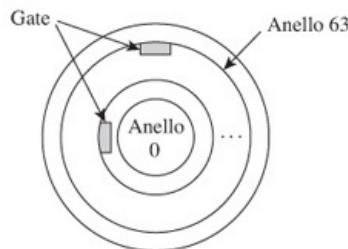
La divisione B consiste di tre livelli, che si differenziano nell'estensione della *protezione obbligatoria*, la resistenza all'intrusione, il supporto per la gestione delle trusted facility e la strutturazione della TCB in elementi critici per la protezione e non. Nel livello B1 deve esistere il controllo degli accessi obbligatorio, e l'amministratore di sistema dovrebbe essere in grado di verificare le azioni di utenti selezionati o azioni relative a programmi o data object selezionati. Nel livello B2 il controllo degli accessi obbligatorio dovrebbe essere esteso a tutti gli utenti e a tutti i dati e oggetti. Il sistema dovrebbe essere in grado di resistere all'intrusione e dovrebbe fornire supporto all'amministratore di sistema. Il sistema dovrebbe, inoltre, fornire un *trusted path* tra un utente e il TCB. Questo path è tipicamente usato quando un utente effettua il login. Il suo uso elimina gli attacchi di masquerading tramite un programma cavallo di Troia (Paragrafo 15.2). Nel livello B3 il sistema deve essere altamente resistente alle intrusioni e deve supportare l'amministratore di sistema nella raccolta di informazioni su imminenti attacchi alla sicurezza e nella terminazione degli eventi che potrebbero essere parte di tali attacchi.

Per la divisione A, un sistema deve avere le capability di livello B3, e il suo sviluppatore deve fornire una prova formale della sua politica di sicurezza.

## 15.9 Casi di studio

### 15.9.1 MULTICS

MULTICS fornisce 64 domini di protezione organizzati ad anelli concentrici. Gli anelli sono numerati dal più interno al più esterno (Figura 15.14). I privilegi di accesso di un dominio includono i privilegi di accesso di tutti i domini con numerazione più alta. Inoltre, il dominio può avere qualche ulteriore privilegio di accesso rispetto ai propri. Ogni procedura di un programma viene assegnata a un dominio di protezione e può essere eseguita solo da un processo che si trova nello stesso dominio di protezione.



**Figura 15.14** Anelli di protezione in MULTICS.

Il codice della componente di un processo può essere composto da procedure in diversi domini di protezione. Viene sollevato un interrupt quando un processo in esecuzione nel dominio di protezione  $D_i$  invoca una procedura che è assegnata a un dominio di protezione  $D_j$ , dove  $D_j \neq D_i$ . Per eseguire la procedura, il dominio di protezione del processo dovrebbe essere cambiato in  $D_j$ . Il kernel controlla se ciò è consentito secondo

la regola per il cambiamento del dominio di protezione. Una versione semplificata di questa regola è la seguente: il cambiamento del dominio di protezione è consentito se un processo eseguito in qualche dominio  $D_i$  invoca una procedura presente in un dominio a numerazione più alta. Tuttavia, un processo, per accedere a un dominio a numerazione più bassa, deve invocare una procedura progettata appositamente, chiamata *gate*. Un tentativo di invocare qualsiasi altra procedura in un livello con numerazione più bassa fallisce e il processo è terminato. Se una chiamata a procedura soddisfa questa regola, il dominio di protezione del processo è temporaneamente modificato nel dominio in cui si trova la procedura invocata. La procedura invocata viene eseguita in questo dominio di protezione e accede alle risorse secondo i privilegi di accesso del dominio. Al ritorno, il dominio di protezione del processo è riportato al suo precedente valore, ossia, a  $D_i$ .

La struttura di protezione di MULTICS è complessa e comporta sostanziali overhead di esecuzione dovuti ai controlli fatti con una chiamata a procedura. A causa del requisito per cui i privilegi di accesso di un dominio di protezione includono i privilegi di accesso di tutti i domini a numerazione più alta, non è possibile utilizzare i domini i cui privilegi di accesso sono disgiunti. Per esempio, i domini  $D_1$ ,  $D_2$  e  $D_3$  della [Figura 15.10](#) non possono essere implementati in MULTICS poiché il dominio  $D_3$  è disgiunto dai domini  $D_1$  e  $D_2$ . Questa caratteristica limita la libertà degli utenti nella specifica dei requisiti di protezione.

### 15.9.2 Unix

Come detto nel [Paragrafo 15.5](#), Unix impiega la cifratura per la sicurezza delle password. È possibile utilizzare il meccanismo delle shadow password, accessibile solo da root, che forza un intruso a utilizzare un attacco esaustivo per individuare le password. Ogni utente Unix ha un id unico nel sistema. L'amministratore di sistema crea gruppi di utenti disgiunti e assegna un group id univoco a ogni gruppo. Le credenziali di un utente sono composte dalla sua user id e il group id. Sono memorizzate nella tabella delle password e diventano il token di autenticazione dell'utente dopo che l'utente è stato autenticato.

Unix definisce tre classi di utente – proprietario del file, user group e altri utenti – e fornisce solo tre diritti di accesso, *r*, *w* e *x*, che rappresentano, rispettivamente, lettura (read), scrittura (write) ed esecuzione (execute). Per ogni classe di utente viene utilizzato un descrittore di accesso bit-encoded a 3 bit e l'access control list (ACL) contiene i descrittori di accesso per le tre classi di utente nella sequenza proprietario del file, user group e altri utenti. In questo modo, l'ACL richiede solo 9 bit ed è memorizzata nell'inode di un file ([Paragrafo 13.14.1](#)). L'identità del proprietario del file è memorizzata in un altro campo dell'inode del file. La [Figura 15.15](#) mostra le ACL di Unix riportate sotto forma di lista di directory. Qualsiasi utente nel sistema può leggere il file *sigma*, ma solo il suo proprietario può scrivere in esso. *delta* è un file di sola lettura per tutte le classi di utenti, mentre *phi* ha i privilegi di lettura, scrittura ed esecuzione solo per il suo proprietario.

---

|                          |               |
|--------------------------|---------------|
| <i>rw-r--r--</i>         | <i>sigma</i>  |
| <i>r--r--r--</i>         | <i>delta</i>  |
| <i>rwx----</i>           | <i>phi</i>    |
| proprietario<br>del file | user<br>group |

---

**Figura 15.15** Access control list di Unix.

I privilegi di accesso di un processo Unix sono determinati in base alla sua uid. Quando il kernel crea un processo, associa l'uid del processo all'id dell'utente che lo ha creato. Così, il processo funziona nel dominio di protezione determinato dall'id dell'utente che l'ha creato. Unix cambia il dominio di protezione di un processo sotto due condizioni – quando il processo effettua una chiamata di sistema e quando è usata la proprietà *setuid* o *setgid*. Un processo ha due distinti stati di *running* – *user running* e *kernel running* ([Paragrafo 5.4.1](#)). Nello stato user-running, un processo ha accesso allo spazio di memoria e ad altre risorse allocate per esso, e ai file nel file system in base alla sua uid. Il processo effettua una transizione nello stato kernel-running tramite una chiamata di sistema. In questo stato, può accedere alle strutture dati del kernel ed, inoltre, ai contenuti dell'intera memoria. Esso ritorna allo stato user-running quando ritorna dalla

chiamata di sistema. Quindi, un cambio di dominio di protezione si verifica implicitamente quando un processo effettua una chiamata di sistema e quando ritorna da essa.

La chiamata *setuid* può essere effettuata in due modi. Un processo può effettuare una chiamata di sistema *setuid*  $<id>$  per cambiare la sua uid a  $<id>$ , e un'altra chiamata di sistema *setuid* con il suo id per ritornare al suo uid originario. In alternativa, l'uid può essere cambiato implicitamente quando un processo esegue una *exec* per eseguire un programma. Il secondo uso si realizza come segue: sia P un programma memorizzato in un file chiamato P. Se il proprietario di P richiede al kernel che P sia eseguito con la proprietà *setuid*, il kernel setta il bit *setuid* nell'inode del file P. Quando P è eseguito da un processo che ha il permesso di esecuzione, il kernel si accorge che il bit *setuid* del file P è impostato, e temporaneamente cambia la uid del processo che esegue P alla uid del proprietario di P. Questa azione effettivamente pone il processo in un dominio di protezione i cui privilegi di accesso sono identici a quelli del proprietario di P. Questa proprietà può essere usata per evitare la violazione della riservatezza discussa nell'Esempio 15.2 in questo modo: l'utente  $u_j$  imposta il bit *setuid* del programma *invest*. L'utente  $u_i$  attribuisce ad  $u_j$  un accesso in lettura al file *finance* prima di invocare *invest*. Ora, il programma *invest* è eseguito con la uid di  $u_j$ . Quindi, *invest* può accedere al file *finance* dell'utente  $u_i$ , ma non può accedere ad alcun altro file di proprietà di  $u_i$ . Analogamente, anche la proprietà *setgid* fornisce un metodo di cambiamento temporaneo del group id di un processo.

### 15.9.3 Linux

Linux autentica un utente al momento del login aggiungendo un valore "salt" alla sua password e cifrando il risultato tramite MD5. Opzionalmente, esso impiega un file di shadow password accessibile solo da root. In più, Linux fornisce *pluggable authentication module* (PAM), attraverso i quali un'applicazione può autenticare un utente in ogni istante tramite una libreria di moduli di autenticazione caricabile dinamicamente. Questa soluzione fornisce flessibilità perché lo schema di autenticazione usato in un'applicazione può essere cambiato senza dover ricompilare l'applicazione. Uno sviluppatore di applicazioni può usare i PAM per incrementare la sicurezza dell'applicazione in molti modi - impiegare uno schema di cifratura delle password di sua scelta, impostare un limite di risorse per gli utenti in modo tale che non possano utilizzare una quantità eccessiva di una risorsa per lanciare così un attacco di tipo denial-of-service, e consentire a specifici utenti di effettuare il login solo in determinati istanti da specifiche locazioni.

L'amministratore di sistema mantiene un file di configurazione PAM per ogni applicazione autorizzata a utilizzare PAM. Ogni file di configurazione PAM specifica come deve essere eseguita l'autenticazione e quali azioni si devono intraprendere, come il montaggio di home directory o il logging dell'evento di autenticazione, dopo l'autenticazione di un utente. Il file di configurazione stabilisce, inoltre, il meccanismo da impiegare quando un utente vuole cambiare la sua password. PAM permette che diversi moduli di autenticazione siano "stacked"; cioè questi moduli vengono invocati uno dopo l'altro. Un'applicazione può usare questa proprietà per autenticare un utente tramite vari mezzi come l'identificazione biometrica e quella basata su password, allo sopo di incrementare la sicurezza.

Linux fornisce la protezione di accesso ai file basata su user id e group id di un processo. Quando un server come NFS accede a un file per conto di un utente, la protezione del file dovrebbe essere eseguita usando la user id e il group id dell'utente, anziché quelli del server. Per facilitare ciò, Linux fornisce le chiamate di sistema *fsuid* e *fsgid* tramite cui un server può temporaneamente assumere l'identità del suo client.

Come descritto nel Paragrafo 4.8.2, il kernel di Linux supporta i moduli caricabili del kernel. Questa caratteristica è stata utilizzata per fornire controlli di accesso avanzati tramite moduli chiamati *Linux security module* (LSM). L'uso di questi moduli permette di supportare molti modelli di sicurezza differenti. Lo schema di base di LSM è semplice: il kernel invoca una funzione di validazione degli accessi prima di accedere a un oggetto. Un LSM fornisce questa funzione, che può consentire o negare l'accesso. Il Security Enhanced Linux (SELinux) della U.S. National Security Agency ha costruito meccanismi di controllo degli accessi aggiuntivi tramite LSM, che forniscono il controllo degli accessi obbligatorio.

Il kernel di Linux fornisce la patch exec-shield, che abilita la protezione contro i buffer overflow e le strutture dati per lanciare attacchi alla sicurezza.

## 15.9.4 Windows

Il modello per la sicurezza di Windows ha molti elementi dei sistemi di classe C2 e B2 secondo i criteri TCSEC (Paragrafo 14.5.2). Esso fornisce il controllo degli accessi discrezionale, la protezione del riuso degli oggetti, la verifica degli eventi relativi alla sicurezza, un security reference monitor (SRM) che rafforza il controllo degli accessi, e un trusted path per l'autenticazione che dovrebbe ostacolare gli attacchi di masquerading lanciati tramite un cavallo di Troia. Tra le caratteristiche più importanti, fornisce sicurezza per l'elaborazione client-server tramite i token di accesso, che sono analoghi alle capability (Paragrafo 15.7).

La sicurezza di Windows si basa sull'uso di *identificativi di sicurezza* (SID); un identificativo di sicurezza è assegnato a un utente, a un host o a un *dominio*, che è composto da diversi host. I campi importanti in un SID sono un identificativo a 48 bit dell'authority value, che identifica l'host o il dominio che ha emesso il SID, e uno a 32 bit subauthority o identificativo relativo (RID) che sono usati principalmente per generare un SID univoco per entità create dallo stesso host o dominio.

Ogni processo e thread ha un *token di accesso* che identifica il suo contesto di sicurezza. (Ricordiamo che usiamo il termine *processo* genericamente sia per un processo che per un thread.) Un token di accesso è generato quando un utente effettua il login, ed è associato al processo di inizializzazione creato per l'utente. Un processo può creare più token di accesso tramite la funzione *LogonUser*. Un token di accesso contiene un SID per l'account utente e un SID per il group account. Questi campi sono utilizzati dal security reference monitor per decidere se il processo che ha il token di accesso può eseguire certe operazioni su un oggetto. Un token di accesso contiene, inoltre, un array di privilegi che indica alcuni privilegi speciali tenuti dal processo, come il privilegio per creare backup di file, di impersonare un client, e di effettuare lo shutdown di un host. Può, inoltre, contenere pochi superprivilegi per caricare e scaricare driver, acquisire la proprietà di oggetti, e creare nuovi token di accesso.

Un oggetto, come un file, ha un *security descriptor*, che contiene l'id del proprietario dell'oggetto, una *discretionary access control list* (DACL) e una *system access control list* (SACL). La DACL è utilizzata per specificare quali utenti possono accedere all'oggetto e in che maniera, mentre la SACL è utilizzata per generare un log di controllo delle operazioni eseguite sull'oggetto. Sia DACL che SACL sono liste di *access control entry* (ACE); tuttavia, sono ACE con ruoli differenti in queste due liste. Un'ACE in una DACL o indica che all'utente specificato è consentito l'accesso all'oggetto, oppure indica che all'utente è vietato l'accesso all'oggetto. Questa soluzione permette la protezione a grana media e aiuta a rendere la DACL compatta; tuttavia, deve essere analizzata l'intera DACL per determinare se a uno specifico utente è consentito accedere all'oggetto nel modo specificato. Un oggetto che può contenere altri oggetti, come una directory, è detto oggetto *container*; chiameremo gli oggetti in esso contenuti i suoi "oggetti figli". Un'ACE nella DACL di un oggetto contenitore contiene dei flag per indicare come va applicata l'ACE agli oggetti figli, cioè nello stesso modo o in qualche altro modo. Un'importante opzione è che l'ACE può essere ereditata da un oggetto figlio che è esso stesso un oggetto contenitore, ma non può essere ulteriormente ereditata da oggetti che possono essere creati nell'oggetto figlio. Questa caratteristica aiuta a limitare la propagazione dei privilegi del controllo degli accessi. Un'ACE nella SACL indica che un'operazione sull'oggetto da parte di un utente o gruppo di utenti deve essere verificata. Viene inserita una entry nel log di audit quando viene eseguita una qualsiasi di queste operazioni.

La caratteristica *impersonation* nel modello di sicurezza di Windows fornisce sicurezza nell'elaborazione client-server. Quando un server esegue alcune operazioni su oggetti per conto di un client, queste operazioni saranno sottoposte ai privilegi di accesso del client, piuttosto che a quelli del server; altrimenti, il client potrebbe essere in grado di realizzare operazioni su questi oggetti che vanno oltre i suoi privilegi di accesso. Analogamente, il security audit log che è generato quando il server accede a un oggetto per conto di un client conterrebbe l'identità del client, piuttosto che quella del server. Questi requisiti sono soddisfatti entrambi, consentendo al server di assumere temporaneamente l'identità del client tramite l'impersonation.

L'impersonation è implementata come segue: quando un client invoca un server, esso indica il tipo di impersonation che vuole che il server compia (il server non può eseguire impersonation senza il consenso del client). Se l'impersonation è abilitata, un *impersonation token* è creato dal token del client e attribuito al server. Il server presenta l'impersonation token, piuttosto che il suo token di accesso, quando esegue le operazioni sugli oggetti. Effettivamente, l'access token e l'impersonation token agiscono come le

capability discusse nel Paragrafo 15.7. Inoltre, per assicurare la sicurezza, il server può creare un token ristretto da un token di impersonation. Un token di questo tipo conterrà un sottoinsieme dei privilegi contenuti nell'impersonation token; esso è come un sottoinsieme di capability discusso nel Paragrafo 15.7.

In Windows Vista sono state aggiunte alcune nuove caratteristiche di sicurezza per renderlo un SO più sicuro.

- *Risolvere il problema degli attacchi di buffer overflow nelle architetture Intel 80 × 86:* ricordiamo dal Paragrafo 15.2.2 che in questa architettura lo stack cresce verso gli indirizzi bassi di memoria. Vista pone i puntatori di ritorno e i parametri della chiamata di funzione più in alto nello stack dei dati locali per evitare la loro alterazione per overflow. I tentativi di esecuzione di codice, introdotto illecitamente come dati, sono ostacolati utilizzando la caratteristica di no-execute (NX) dei processori contrassegnando con flag le parti della memoria usate per tenere dati come zone di no-execute.
- *Individuare la corruzione dell'heap:* un intruso può lanciare un attacco di buffer overflow nell'heap. Per evitare ciò, i metadati come i puntatori nell'heap sono codificati eseguendo un OR esclusivo con un numero casuale. La corruzione dell'heap, attraverso overflow o altro, modificherebbe dei metadati, di conseguenza la sua decifratura fallirebbe. Quando si verifica questa situazione, il kernel termina il processo.
- *Evitare accessi al codice di sistema:* parti del codice di sistema vengono caricate in maniera casuale in una delle 256 possibili locazioni in memoria, per rendere difficile a un intruso l'accesso a esse. I puntatori di funzioni che esistono in memoria per tempi lunghi sono offuscati dall'esecuzione di un OR esclusivo con numeri casuali.
- *Evitare l'uso non corretto dei privilegi:* i servizi di sistema non vengono eseguiti nell'account di sistema, come accadeva nelle precedenti versioni di Windows. Vengono eseguiti in account con privilegi minori. Anche i processi avviati dagli amministratori di sistema vengono eseguiti in modalità con minori privilegi e il kernel chiede l'autenticazione da parte dell'amministratore stesso quando il suo processo sta per eseguire una funzione che richiede i privilegi di amministratore. Se l'autenticazione ha esito positivo, altre finestre sullo schermo vengono nascoste per evitare lo spoofing dell'interfaccia utente e del mouse.
- *Protezione degli accessi di rete:* a meno che un sistema non sia conforme alle norme imposte dall'amministratore, esso avrà l'accesso bloccato o un accesso limitato alla rete.

## Riepilogo

Un obiettivo fondamentale di un SO è quello di assicurare la non interferenza nelle elaborazioni e nell'uso delle risorse degli utenti. Tuttavia, gli utenti hanno la necessità di condividere alcune delle loro risorse, come programmi e dati memorizzati in file, con altri. Di conseguenza, un aspetto importante nella implementazione della non interferenza è sapere quali accessi a una risorsa sono legittimi e quali accessi costituiscono interferenza. I tentativi di interferenza possono essere sollevati al di fuori o all'interno del sistema. Le misure impiegate per contrastare tali minacce costituiscono, rispettivamente, la *sicurezza* e la *protezione*. L'autenticazione è la tecnica chiave della sicurezza; essa determina se una persona è un utente registrato di un sistema. L'autorizzazione è la tecnica chiave della protezione. Essa determina se a un utente è consentito accedere a una risorsa. In questo capitolo abbiamo studiato l'implementazione delle tecniche di autenticazione e di autorizzazione.

Una persona o un programma che effettua un tentativo di interferenza è detto *intruso*. Gli intrusi possono impiegare vari mezzi che sfruttano i punti deboli nella sicurezza di un sistema, o per impersonare un utente oppure per negare l'uso legittimo delle risorse da parte di utenti. Questi mezzi includono: *cavalli di Troia*, *virus*, *worm* o l'uso di *buffer overflow*. Le minacce imposte dagli intrusi sono contrastate ponendo attenzione al caricamento di programmi non noti in un sistema e rimuovendo i punti deboli nella sicurezza.

Il servizio di autenticazione di un SO mantiene in un database i nomi degli utenti registrati e le informazioni usate per identificarli. Esso usa la *cifratura* (*encryption*), cioè una trasformazione algoritmica dei dati, per evitare che gli intrusi accedano e facciano un uso scorretto del database di autenticazione. Il *cifrario a blocchi* (*block cipher*)

*cipher*) e il *cifrario a flusso (stream cipher)* sono tecniche di cifratura ampiamente usate; gli standard di cifratura *digital encryption standard* (DES) e *advanced encryption standard* (AES) sono stati ampiamente trattati.

Il servizio di autorizzazione di un SO ha una *struttura di protezione* che contiene due tipi di informazioni. Un *privilegio di accesso* rappresenta un diritto di un utente ad accedere a uno specifico file in un determinato modo. *L'informazione di protezione* di un file indica quali utenti possono accedere al file e in che maniera.

L'organizzazione della struttura di protezione controlla quanta discrezionalità un utente può esercitare specificando quali utenti possono accedere ai suoi file e in che maniera; questa è detta *granularità della protezione*. Le *liste di controllo degli accessi, le liste di capability e i domini protezione* sono strutture di protezione alternative.

A un sistema di elaborazione viene assegnata una classificazione di sicurezza basata sul grado di conformità ai requisiti di sicurezza e protezione. La sua capacità di fornire protezione a grana fine e supportare l'amministratore di sistema nell'implementazione delle politiche di sicurezza sono fattori determinanti della sua classificazione di sicurezza.

## Domande

- 15.1. Classificare ciascuna delle seguenti affermazioni come vera o falsa.
- Il meccanismo di autenticazione è usato per distinguere tra utenti e non utenti di un sistema.
  - Un authentication token contiene la lista dei privilegi di accesso posseduti da un utente.
  - Il database di autorizzazione è utilizzato dai meccanismi di sicurezza.
  - La cifratura dell'informazione assicura la sua integrità.
  - Il masquerading è un attacco alla sicurezza.
  - Un virus lancia un attacco alla sicurezza solo se esplicitamente scaricato da un utente.
  - La tecnica di buffer overflow può essere utilizzata per lanciare un attacco alla sicurezza.
  - Le sottostringhe identiche in un testo in chiaro, quando cifrate con un cifrario a flusso, portano sempre a sottostringhe identiche nel suo testo cifrato.
  - Per autenticare un utente all'istante di login, un SO decifra la forma cifrata delle password utente memorizzate nel database di autenticazione e confronta i risultati con la password presentata dall'utente.
  - Il password aging limita l'esposizione di una password agli attacchi da parte di un intruso.
  - Due capability di un oggetto possono attribuire identici privilegi di accesso ai loro holder.
  - La cifratura viene usata per proteggere le capability software.
- 15.2. Quale delle seguenti è una violazione della protezione?
- L'utente X che ha un privilegio in scrittura sul file alpha dell'utente Y scrive dati non validi in alpha.
  - Un non utente riesce a leggere i dati memorizzati in un file beta in un sistema di elaborazione.
  - L'utente X riesce a leggere il file alpha dell'utente Y anche se non ha i privilegi di lettura su esso.
  - Nessuna delle (a)-(c).
- 15.3. Accoppiare gli item in ciascuna colonna.
- |                                                |                               |
|------------------------------------------------|-------------------------------|
| i. Lista di controllo degli accessi stile-Unix | i. Protezione a grana fine    |
| ii. Matrice di controllo degli accessi (ACM)   | ii. Protezione a grana grossa |
| iii. Domini di Protezione                      | iii. Protezione a grana media |

## Problemi

- 15.1. Illustrare la procedura da seguire per effettuare i cambiamenti nei database di autenticazione e autorizzazione di [Figura 15.1](#).
- 15.2. Elenicare gli attacchi alla sicurezza che non possono essere evitati dalla cifratura.
- 15.3. Discutere se la cifratura può assicurare la segretezza, la riservatezza e l'integrità dei dati.
- 15.4. Formulare una regola di sicurezza che elimina i punti deboli nella sicurezza dell'Esempio 15.1.
- 15.5. Descrivere le condizioni sotto le quali un attacco al testo in chiaro scelto può essere lanciato contro le password.
- 15.6. Commentare l'impatto della granularità della protezione sulle dimensioni delle varie strutture di protezione. Suggerire metodi di riduzione della dimensione della lista di controllo degli accessi (access control list (ACL)) quando è implementata la protezione a grana media.
- 15.7. Un file è cifrato usando una funzione di cifratura  $E$  e una chiave  $k$ . Nessun altro controllo di protezione viene effettuato dal file system. Se un utente vuole condividere il file con un altro utente, egli rende disponibili  $E$  e  $k$  all'altro utente. Confrontare lo schema precedente per la protezione dei file con uno schema di protezione che usa un'access control list, sulla base di (a) facilità nell'attribuzione dei privilegi di accesso agli utenti o nella revoca, e (b) attribuzione di differenti tipi di privilegi di accesso allo stesso file.
- 15.8. Alcuni vecchi sistemi operativi usavano associare le password ai file e permettere a qualsiasi programma che presentava una password valida per il file di accedervi. Confrontare questo schema di protezione con uno schema di protezione basato su capability con lo stesso criterio del Problema 15.7.
- 15.9. Capability review è il processo con cui un SO trova tutti i processi che hanno una capability per uno specifico oggetto  $obj_i$ . Descrivere come può essere eseguita un'operazione di review in un sistema che usa capability hardware o software.
- 15.10. Un SO esegue la validazione delle capability software come segue: quando una nuova capability è creata, l'object manager memorizza una copia della capability per usi propri. Quando un processo vuole eseguire un'operazione su un oggetto, la capability presentata da esso è confrontata con le capability memorizzate. L'operazione è consentita solo se esiste un matching delle capability con l'object manager. È questo lo schema sicuro (foolproof)? Permette la revoca selettiva dei privilegi di accesso?
- 15.11. Un SO crea i server per offrire vari servizi agli utenti. Nel gestire una richiesta di servizio fatta da un utente, un server può dover accedere alle risorse su richiesta dell'utente. Tali accessi alle risorse devono essere subordinati ai privilegi di accesso dell'utente, piuttosto che a quelli del server.
  - a. A tale scopo è proposto il seguente schema: quando un utente invoca un servizio, invia il suo authentication token al server. Quando il server richiede l'accesso a una risorsa, esso presenta al servizio di autorizzazione l'authentication token dell'utente, piuttosto che il suo authentication token. In questo modo, il suo uso delle risorse sarà subordinato ai privilegi di accesso dell'utente. Come assicurare che un intruso non possa sfruttare questa soluzione per impersonare (masquerade) un utente? (Suggerimento: assicurare che l'authentication token dell'utente non possa essere alterato.)
  - b. Progettare uno schema usando le capability.
- 15.12. Spiegare come i buffer analoghi a quelli per la traduzione degli indirizzi usati nelle memorie virtuali (Paragrafo 12.2.2) o una cache memory possono essere usati nello schema di [Figura 15.12](#) per velocizzare gli accessi agli oggetti.
- 15.13. Diversi nodi di un sistema distribuito possono creare nuovi oggetti concorrentemente. Descrivere uno schema che possa assicurare l'univocità degli object id in un SO distribuito.
- 15.14. Studiare la letteratura Unix e descrivere le tecniche adottate in Unix per (a) trovare l'id dell'utente che possiede un file, e (b) decidere se un utente appartiene allo stesso user group dell'owner del file.

## | Note bibliografiche

Ludwig (1998) descrive differenti tipi di virus, mentre in Ludwig (2002) sono descritti i virus delle e-mail. Spafford (1989) tratta il worm Internet Morris che causò gravi problemi nel 1988, e Berghel (2001) descrive il worm Code Red del 2001.

Landwehr (1981) discute i modelli formali della sicurezza informatica. Voydock e Kent (1983) trattano le problematiche di sicurezza nei sistemi distribuiti e le tecniche pratiche usate per affrontarle.

Shannon (1949) è autore del classico lavoro sulla sicurezza informatica. Il lavoro riporta le proprietà di diffusione e confusione dei cifrari. Denning e Denning (1979) e Lempel (1979) espongono in modo corretto una panoramica relativa alla sicurezza dei dati e alla crittologia, rispettivamente. Schneier (1996) e Ferguson e Schneier (2003) sono testi sulla crittografia, mentre Pfleeger e Pfleeger (2003) hanno scritto un testo sulla sicurezza informatica. Stallings (2003) discute la crittografia e la sicurezza di rete.

Naor e Yung (1989) descrivono circa le funzioni hash unidirezionali (one-way). Rivest (1991) descrive la funzione del digest message MD4. L'obiettivo di MD4 è rendere impraticabile dal punto di vista computazionale la produzione di due messaggi con lo stesso message digest, oppure un messaggio con un dato digest. MD4 è estremamente veloce e resiste, con successo, agli attacchi di criptoanalisi. Rivest (1992) descrive MD5, che è conservativo e un po' più lento di MD4. Preneel (1998) descrive le primitive di crittografia per le informazioni di autenticazione.

La protezione degli accessi basata su matrici e i domini di protezione sono discussi da Lampson (1971) e Popek (1974). Organick (1972) discute gli anelli di protezione in MULTICS. La proprietà *setuid* di Unix è descritta nella maggior parte dei libri su Unix.

Dennis e Van Horn (1966) hanno effettuato uno studio, ampiamente consultato, sul concetto di capability. Levy (1984) descrive alcuni sistemi basati sulle capability. Mullender e Tanenbaum (1986) e Tanenbaum (2001) descrivono le capability software di Amoeba. Anderson et al. (1986) trattano le capability software con una condizione sul containment.

La Trusted Computer System Evaluation Criteria (TCSEC) del Dipartimento della Difesa americano offre una classificazione delle caratteristiche di sicurezza dei sistemi di elaborazione. Essa è descritta in DoD (1985).

Spafford et al. (2003) discutono la sicurezza in Solaris, Mac OS, Linux e nei sistemi operativi FreeBSD. Wright et al. (2002) trattano i moduli di sicurezza in Linux. Russinovich e Solomon (2005) discutono le caratteristiche legate alla sicurezza in Windows.

1. Anderson, M., R.D. Pose, and C.S. Wallace (1986): "A password-capability system," *The Computer Journal*, **29** (1), 1-8.
2. Berghel, H. (2001): "The Code Red worm," *Communications of the ACM*, **44** (12), 15-19.
3. Denning, D.E., and P.J. Denning (1979): "Data security," *Computing Surveys*, **11** (4).
4. Dennis, J.B., and E.C. Van Horn (1966): "Programming semantics for multiprogrammed computations," *Communications of the ACM*, **9** (3).
5. DoD (1985): *Trusted Computer System Evaluation Criteria*, U.S. Department of Defense.
6. Ferguson, N., and B. Schneier (2003): *Practical Cryptography*, John Wiley, New York.
7. Fluhrer, S., I. Mantin, and A. Shamir (2001): "Weaknesses in the key scheduling algorithm of RC4," *Proceedings of 8th Annual Workshop on Selected Areas in Cryptography*.
8. Lampson, B.W. (1971): "Protection," *Operating Systems Review*, 8 (1), 18-24.
9. Landwehr, C.E. (1981): "Formal models for computer security," *Computing Surveys*, **13** (3), 247-278.
10. Lempel, A. (1979): "Cryptology in transition," *Computing Surveys*, **11** (4), 286-303.
11. Levy, H.M. (1984): *Capability-Based Computer Systems*, Digital Press, Burlington, Mass.
12. Ludwig, M.A. (1998): *The Giant Black Book of Computer Viruses*, 2nd ed., American Eagle, Show Low, Ariz.
13. Ludwig, M.A. (2002): *The Little Black Book of Email Viruses*, American Eagle, Show Low, Ariz.
14. Menezes, A., P. van Oorschot, and S. Vanstone (1996): *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Fla.
15. Mullender, S.P., and A. Tanenbaum (1986): "The design of a capability-based

- distributed operating system," *Computer Journal*, **29** (4).
- 16. Nachenberg, C. (1997): "Computer virus-antivirus coevolution," *Communications of the ACM*, **40**, 46-51.
  - 17. Naor, M., and M. Yung (1989): "Universal one-way hash functions and their cryptographic applications," *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 33-43.
  - 18. Oppliger, R. (1997): "Internet security: firewalls and beyond," *Communications of the ACM*, **40** (5), 92-102.
  - 19. Organick, E.I. (1972): *The MULTICS System*, MIT Press, Cambridge, Mass.
  - 20. Pfleeger, C.P., and S. Pfleeger (2003): *Security in computing*, Prentice Hall, Englewood Cliffs, N.J.
  - 21. Popek, G.J. (1974): "Protection structures," *Computer*, 7 (6), 22-33.
  - 22. Preneel, B. (1998): *Cryptographic primitives for Information Authentication - State of the art in applied cryptography*, LNCS 1528, Springer Verlag, 1998.
  - 23. Rivest, R. (1991): "The MD4 message digest algorithm," *Proceedings of Advances in Cryptology - Crypto'90, Lecture Notes in Computer Science, volume 537*, Springer-Verlag, 303-311.
  - 24. Rivest, R. (1992): "The MD5 Message digest algorithm," Request for Comments, RFC 1321.
  - 25. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
  - 26. Schneier, B. (1996): *Applied cryptography*, 2nd ed., John Wiley, New York.
  - 27. Shannon, C.E. (1949): "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, October 1949.
  - 28. Spafford, E.H. (1989): "The Internet worm: crisis and aftermath," *Communications of the ACM*, **32** (6), 678-687.
  - 29. Spafford, G., S. Garfinkel, and A. Schwartz (2003): *Practical UNIX and Internet Security*, 3rd ed., O'Reilly, Sebastopol, Calif.
  - 30. Stallings, W. (2003): *Cryptography and Network Security: Principles and Practice*, 3rd ed., Prentice Hall, N.J.
  - 31. Stiegler, H.G. (1979): "A structure for access control lists," *Software - Practice and Experience*, **9** (10), 813-819.
  - 32. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
  - 33. Voydock, V.L., and S.T. Kent (1983): "Security mechanisms in high level network protocols," *Computing Surveys*, **15** (2), 135-171.
  - 34. Wofsey, M.M. (1983): *Advances in Computer Security Management*, John Wiley, New York.
  - 35. Wright, C., C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman (2002): "Linux Security modules: General security support for the Linux kernel," *Eleventh USENIX Security Symposium*.

---

## PARTE 5

# Sistemi operativi distribuiti

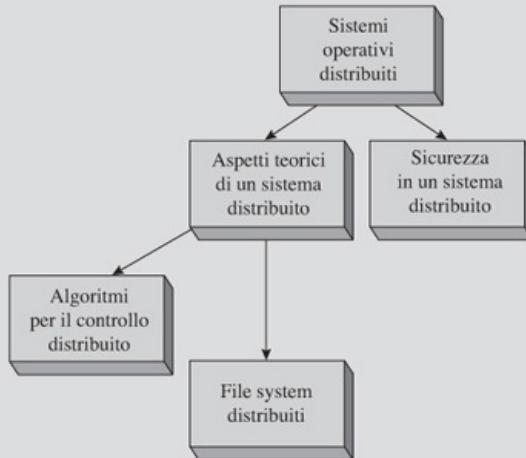
---

Un sistema distribuito è costituito da parecchi nodi, ciascuno dei quali è un sistema di calcolo dotato di clock e memoria, che possono comunicare tra loro attraverso la rete. I sistemi operativi distribuiti sfruttano le potenzialità offerte da un sistema distribuito in vari modi. In primo luogo, essi semplificano l'organizzazione di un'applicazione in termini di *calcolo distribuito*, ovvero permettono di suddividere il calcolo tra più processi in esecuzione su nodi diversi del sistema. Quindi, per servire tali processi in modo efficiente, i sistemi operativi distribuiti bilanciano il carico computazionale sui vari calcolatori migrando, se necessario, processi da un nodo all'altro. In tal modo, i processi di una stessa applicazione possono competere per l'uso di CPU in nodi diversi, garantendo da un lato un'esecuzione più veloce dell'applicazione, e dall'altro un miglioramento generale delle prestazioni del sistema. Infine, i sistemi operativi distribuiti possono sfruttare l'eventuale ridondanza di risorse e collegamenti disponibili sulla rete per fornire un'elevato grado di affidabilità.

Per garantire tali benefici, i sistemi operativi effettuano funzioni di controllo, quali quelli relativi ai processi e ai deadlock, sulla base dell'intero sistema distribuito. Tuttavia, nello svolgere tali funzioni, i sistemi operativi non possono utilizzare le nozioni di *tempo* e *stato* tradizionalmente considerate in ambito monoprocesso, proprio a causa della natura distribuita del sistema. Pertanto, le funzioni di controllo vengono a loro volta effettuate in un'ottica distribuita, ovvero vengono gestite da vari processi, strettamente cooperanti, eseguiti su calcolatori diversi.

La stessa cooperazione garantisce anche una maggior tolleranza ai guasti, in quanto, generalmente, un guasto in un sistema distribuito non blocca l'intero sistema, ma si riflette solo su alcune parti di esse. Pertanto, i sistemi operativi distribuiti utilizzano particolari tecniche di tolleranza ai guasti per minimizzare l'impatto degli stessi.

Tuttavia, la presenza di una comunicazione di rete può avere anche implicazioni negative sul sistema. La comunicazione di rete è lenta, e questo può inficiare negativamente le prestazioni del sistema, in particolare se i processi accedono a file remoti attraverso la rete. Per prevenire un eccessivo degrado delle prestazioni, i sistemi operativi distribuiti utilizzano, quindi, tecniche per ridurre il traffico di rete. Infine, visto che la rete rende il sistema particolarmente sensibile ad attacchi esterni, i sistemi operativi distribuiti impiegano tecniche speciali anche per fornire un'adeguato livello di sicurezza.



Ordine temporale in cui i capitoli di questa parte dovrebbero essere trattati nel corso.

## Capitolo 16 - Sistemi operativi distribuiti

Un sistema distribuito è costituito da componenti hardware, quali calcolatori e infrastrutture di rete, componenti software, quali *protocolli di rete* e *applicazioni distribuite*, e da un sistema operativo. Il capitolo descrive le caratteristiche fondamentali di tali componenti e il modo in cui queste influenzano la *velocità di calcolo*, l'*affidabilità* e le *prestazioni* offerte da un sistema distribuito.

## Capitolo 17 - Problematiche teoriche di un sistema distribuito

*Tempo* e *stato* sono due aspetti fondamentali per un sistema operativo convenzionale. Tuttavia, tali nozioni non possono essere usate allo stesso modo in un sistema distribuito. Quest'ultimo, infatti, è costituito da parecchi calcolatori, ognuno con il proprio clock e la propria memoria, che comunicano tramite scambio di messaggi e sono soggetti a ritardi di comunicazione non prevedibili. Il capitolo presenta definizioni alternative alle tradizionali nozioni di tempo e stato. Tali nozioni vengono usate nella progettazione di algoritmi di controllo e nelle tecniche di ripristino usate nei sistemi operativi distribuiti.

## Capitolo 18 - Algoritmi di controllo distribuiti

Per implementare le funzioni di controllo, i sistemi operativi distribuiti utilizzano algoritmi che coinvolgono vari nodi del sistema. Il capitolo definisce cosa si intende per correttezza di un algoritmo distribuito, e presenta alcuni algoritmi necessari per effettuare cinque operazioni di controllo fondamentali: *mutua esclusione*, *gestione dei deadlock*, *elezione del nodo coordinatore*, *schedulazione dei processi* e *dei processi*.

## Capitolo 19 - File system distribuiti

Un file system distribuito memorizza i file in più nodi del sistema, pertanto un file può trovarsi su nodi diversi. Le prestazioni e l'affidabilità dei file system distribuiti sono determinate dal modo in cui viene gestito l'accesso ai file. Il capitolo descrive diversi modi di gestire l'accesso nei vari nodi del sistema distribuito. Vengono, inoltre, presentati il *file caching* necessario per fornire un buon livello di prestazioni, e il concetto di *file server stateless*, utilizzato per garantire un elevato grado di affidabilità.

## Capitolo 20 - Sicurezza nei sistemi distribuiti

La presenza di una rete rende i sistemi distribuiti sensibili agli attacchi all'identità. Il capitolo descrive le tecniche di *autenticazione* e le strategie di *sicurezza* usate nei

sistemi operativi distribuiti per contrastare tali attacchi. Vengono, inoltre, presentate le tecniche per verificare l'autenticità dei dati.

---

# CAPITOLO 16

## Sistemi operativi distribuiti

---

### Obiettivi di apprendimento

- Definizione e caratteristiche di un sistema distribuito
- Definizione e caratteristiche di un sistema operativo distribuito
- Comunicazione inter-processo
- Paradigmi di calcolo distribuito
- Problematiche di rete
- Problematiche relative alla progettazione di un sistema operativo distribuito

Un sistema distribuito è costituito da un insieme di calcolatori, ciascuno fornito di memoria e clock propri, connessi tramite una rete e controllati da un unico sistema operativo distribuito. I benefici di un tale sistema sono molteplici: condivisione delle risorse collocate su nodi diversi, affidabilità, grazie alla presenza nel sistema di più CPU e di risorse ridondanti, e più elevata velocità di esecuzione, derivata dalla possibilità di distribuire i processi di una stessa applicazione su più calcolatori. Per ottenere tali benefici, un sistema distribuito si basa su quattro componenti fondamentali: la presenza di più *sistemi di calcolo*, la disponibilità di una *rete di interconnessione*, l'esistenza di *applicazioni distribuite* e l'uso di un *sistema operativo distribuito*.

Il ruolo di tali componenti può essere riassunto come segue. Ogni calcolatore rappresenta un *nodo* del sistema distribuito e le sue caratteristiche architetturali contribuiscono all'incremento delle prestazioni e al livello di affidabilità fornite dal sistema. Il sistema operativo integra le funzionalità dei nodi del sistema distribuito per fornire condivisione delle risorse, velocità di calcolo e affidabilità. Per implementare tali caratteristiche è necessario disporre di applicazioni distribuite, le cui operazioni possono essere eseguite su nodi diversi del sistema. Tali applicazioni utilizzano opportuni *protocolli di comunicazione interprocesso* che, sfruttando i tradizionali *protocolli di rete*, trasferiscono messaggi in modo affidabile tra i vari nodi del sistema.

In questo capitolo, verranno trattate le caratteristiche delle sopracitate componenti di un sistema distribuito in modo da creare una base di conoscenze necessarie all'approfondimento dei sistemi operativi distribuiti. Inoltre, verranno identificate una serie di problematiche tipiche dei sistemi operativi distribuiti derivate proprio dalla natura distribuita del loro ambiente di esecuzione. Tali problematiche saranno quindi approfondite nei capitoli successivi.

### 16.1 Caratteristiche dei sistemi distribuiti

Un sistema distribuito è composto da almeno due calcolatori, ognuno con memoria e clock propri, e da alcuni componenti hardware per la gestione della rete, e deve essere in grado di fornire parte delle funzioni di controllo di un sistema operativo (Definizione 3.8). I benefici di un sistema distribuito sono stati discussi precedentemente nel Paragrafo 3.8 e sono riassunti in [Tabella 16.1](#).

| Caratteristiche            | Descrizioni                                                                                                                                                                                                                                                                                    |
|----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Condivisione delle risorse | Un'applicazione può usare risorse collocate in calcolatori diversi.                                                                                                                                                                                                                            |
| Affidabilità               | Un sistema distribuito fornisce continuità del servizio, malgrado la presenza di guasti. Ciò è ottenuto grazie alla ridondanza di risorse, collegamenti di rete e servizi del sistema operativo.                                                                                               |
| Velocità di calcolo        | I processi di un'applicazione multiprocesso possono essere eseguiti in parallelo su calcolatori diversi, riducendo in tal modo la durata dell'esecuzione.                                                                                                                                      |
| Comunicazione              | Processi in esecuzione su calcolatori diversi possono comunicare in modo affidabile usando i servizi forniti dal sistema operativo.                                                                                                                                                            |
| Scalabilità                | I <i>sistemi aperti</i> permettono di aggiungere nuovi sottosistemi in un sistema distribuito senza dover rimpiazzare o aggiornare i sottosistemi esistenti. In tal modo, il costo per incrementare la capacità di un sistema distribuito è proporzionale alla capacità aggiuntiva desiderata. |

**Tabella 16.1** Benefici di un sistema distribuito.

L'uso dei sistemi distribuiti è cresciuto rapidamente negli anni 90 grazie al calo drastico dei costi delle componenti hardware e all'avvento dei cosiddetti *sistemi aperti* che, adottando interfacce non proprietarie definite in modo preciso per garantire la comunicazione sia tra i componenti interni del sistema che verso l'esterno, hanno facilitato la crescita incrementale dei sistemi di calcolo. Tali interfacce sono tipicamente sviluppate e approvate da organi di standardizzazione e pertanto vengono prontamente adottate dalle aziende produttrici di calcolatori. Il loro utilizzo permette di espandere un sistema di calcolo in modo incrementale aggiungendo nuovi componenti e sottosistemi. La LAN è un eccellente esempio di sistema aperto. I sistemi di calcolo, siano essi supercomputer o semplici PC, possono essere connessi a una stessa LAN visto che utilizzano un'interfaccia standard. Quando un sistema distribuito implementa la comunicazione per mezzo di una LAN, la sua capacità di calcolo può essere aumentata in modo incrementale connettendo nuovi nodi alla stessa LAN.

I benefici dei sistemi distribuiti elencati in [Tabella 16.1](#) sono realizzati usando i seguenti componenti hardware e software:

- *componenti hardware*: sistemi di calcolo e hardware di rete come cavi e router;
- *componenti software*: moduli del sistema operativo che gestiscono la creazione e la schedulazione di applicazioni distribuite e l'uso di risorse remote, linguaggi di programmazione che supportano la scrittura di applicazioni distribuite, e software di rete che garantisce una comunicazione affidabile.

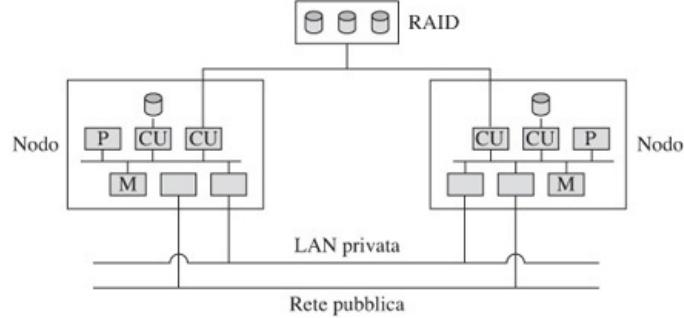
Per indicare un sistema di calcolo che fa parte di un sistema distribuito, si usano diversi termini. Nel seguito, verrà adottata la seguente convenzione: i termini *host* e *nodo* verranno usati per indicare un calcolatore, rispettivamente, in senso fisico e in senso logico, mentre *sito* verrà usato per indicare la locazione in cui risiede un *host* all'interno del sistema distribuito. Entità quali processi e risorse verranno invece chiamate *locali* o *remote*, rispettivamente, quando risiedono nello stesso sito o in siti differenti.

## 16.2 Nodi di un sistema distribuito

Un sistema distribuito può contenere diversi tipi di nodi. Un nodo *minicomputer* ha una singola CPU condivisa per servire applicazioni di utenti diversi. Un nodo *workstation* ha una singola CPU utilizzata per servire una o più applicazioni di un unico utente. Un nodo *multiprocessore*, chiamato *processor pool*, è composto invece da un numero di CPU che può eccedere il numero di utenti, le cui applicazioni vengono servite in parallelo.

Un *cluster* è costituito da un insieme di host che lavorano in modo strettamente integrato. Solitamente, un cluster è un nodo di un sistema di calcolo, e ogni suo host è un nodo all'interno del cluster. La [Figura 16.1](#) esemplifica lo schema di un cluster con due nodi. Ognuno di essi è un sistema di calcolo con memoria e dispositivi di I/O privati. I

nodi possono condividere anche un sistema di memorizzazione di massa di tipo RAID, che offre un tasso di trasferimento dati molto elevato e un altrettanto elevato livello di affidabilità (Paragrafo 14.3.5), o *storage area network*, ovvero una rete di dispositivi di massa ad alta velocità (Paragrafo 14.3.4). Ogni nodo è inoltre connesso a due reti: una *LAN privata* a cui accedono solo i nodi del cluster, e una *rete pubblica* attraverso la quale è possibile comunicare con altri nodi del sistema distribuito esterni al cluster.



**Figura 16.1** Architettura di un cluster.

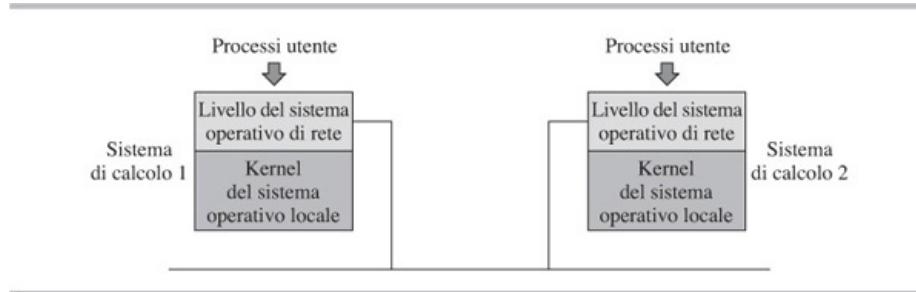
Tutte le operazioni di controllo dei nodi sono effettuate da un software apposito (*cluster software*). Tale software contribuisce a incrementare sia la velocità di calcolo, schedulando processi differenti di una stessa applicazione su nodi diversi del cluster, che l'affidabilità sfruttando la ridondanza di CPU e risorse all'interno del cluster. Il Paragrafo 16.3 descrive l'implementazione di tali caratteristiche nel cluster server di Windows e nel cluster di Sun.

### 16.3 Integrazione dell'attività dei nodi di un sistema distribuito

Per conseguire i benefici riassunti in [Tabella 16.1](#), ovvero condivisione delle risorse, affidabilità e incremento della velocità di calcolo, i processi di un'applicazione devono essere distribuiti in nodi diversi del sistema (1) ognqualvolta sia possibile, per incrementare la velocità di calcolo e sfruttare le risorse disponibili in modo efficiente, e (2) quando necessario, per garantire l'affidabilità del sistema. Ciò si ottiene integrando l'attività dei vari nodi del sistema tramite l'integrazione dei loro kernel. In questo paragrafo verranno pertanto illustrati differenti modi per integrare l'attività dei nodi. Nel Paragrafo 16.8, invece, verranno discusse le problematiche di progetto di un sistema operativo distribuito.

#### Sistemi operativi di rete

I *sistemi operativi di rete* sono la prima forma di sistema operativo per architetture distribuite. Il loro obiettivo consiste nel permettere la condivisione di risorse tra due o più sistemi di calcolo che eseguono ciascuno un proprio sistema operativo. Come illustrato in [Figura 16.2](#), un sistema operativo di rete è un livello software che si colloca tra il sistema operativo locale e i processi utente. Se un processo richiede l'accesso a una risorsa locale, il sistema operativo di rete passa la richiesta al kernel del sistema operativo locale. Al contrario, se la richiesta prevede l'accesso a una risorsa remota, il sistema operativo di rete intercetta la richiesta e la soddisfa contattando il livello software corrispondente del nodo remoto che detiene la risorsa. Molti sistemi operativi di rete sono stati sviluppati come estensioni di Unix. Per esempio, il *Newcastle connection*, noto anche come *Unix United*, è un sistema operativo di rete sviluppato all'Università di Newcastle. In Newcastle connection l'accesso a file remoti avviene per mezzo di chiamate di sistema identiche a quelle usate per l'accesso a file locali.



**Figura 16.2** Un sistema operativo di rete.

Un sistema operativo di rete è più semplice da implementare rispetto a un sistema distribuito completamente integrato. Tuttavia, in tale contesto, i sistemi operativi locali mantengono la loro identità, visibile agli utenti, e operano in modo indipendente senza integrare le loro funzionalità. In alcuni sistemi operativi di rete, gli utenti devono accedere esplicitamente a un sistema operativo remoto per usare le risorse condivise da quest'ultimo. In tal caso, l'utente deve pertanto essere a conoscenza di dove sono posizionate le varie risorse per poterle usare. Un sistema operativo di rete, inoltre, non può bilanciare od ottimizzare l'uso delle risorse. Pertanto, mentre le risorse di un nodo potrebbero essere pesantemente utilizzate, analoghe risorse in altri nodi potrebbero essere scarsamente utilizzate o addirittura non usate. Un sistema operativo di rete non può nemmeno fornire tolleranza ai guasti. Infatti, le applicazioni eseguite su un tale sistema indicano esplicitamente le risorse con il loro identificatore quando intendono usarle, e pertanto qualora queste ultime divenissero non disponibili, le applicazioni dovrebbero essere terminate.

### **La gestione dei cluster di Windows e di Sun**

Il software che gestisce un cluster non è un vero e proprio sistema operativo distribuito, ma presenta parecchie caratteristiche di quest'ultimo, quali per esempio la possibilità di implementare la ridondanza per mezzo delle risorse disponibili nel cluster, e di incrementare la velocità di calcolo sfruttando la presenza di più CPU.

Il *cluster server* di *Windows* fornisce supporto per la tolleranza ai guasti nei cluster con due o più nodi server. Le applicazioni devono usare delle apposite API (*application program interface*) per accedere ai servizi del cluster. La tolleranza ai guasti viene garantita attraverso l'uso di dischi RAID di livello 0, 1 o 5 (Paragrafo 14.3.5) che sono condivisi tra tutti i nodi server. Quando si presenta un guasto in un server o un server viene spento, il cluster server trasferisce le sue funzionalità verso un altro server senza causare l'interruzione del servizio.

Il cluster è gestito da *algoritmi di controllo distribuiti* (Capitolo 18). Tali algoritmi richiedono che tutti i nodi abbiano una vista consistente del cluster. A tale scopo, ogni nodo dispone di un *manager*, che contiene la lista dei nodi del cluster. Ogni manager periodicamente invia un messaggio, *heartbeats*, agli altri manager per rilevare la presenza di eventuali guasti. Se un manager identifica un guasto, invia un nuovo messaggio, contenente i dettagli del guasto, a tutti gli altri manager tramite la rete privata. In tal modo, ogni manager è in grado di mantenere aggiornata la lista dei nodi. L'aggiornamento di tale lista viene indicato con il termine *regroup event*.

Un cluster server può fornire risorse fisiche, risorse logiche o servizi. Le risorse sono implementate in *dynamic link library* (DLL) che esportano una corrispondente interfaccia DLL. Ogni risorsa appartiene a un *gruppo* all'interno di un determinato nodo del cluster. Tuttavia, nel caso si verifichi un guasto, un gruppo può essere spostato in un altro nodo. I gruppi sono attivati e disattivati dal gestore delle risorse (*resource manager*) del nodo a cui appartengono. Se una risorsa smette di funzionare, il gestore delle risorse informa il *failover manager* e trasferisce l'esecuzione del gruppo a cui la risorsa appartiene tramite una operazione di *handover*. Il gruppo viene quindi riavviato in un nodo differente. Quando si rileva un guasto in un nodo, tutti i gruppi appartenenti al nodo vengono trasferiti verso altri nodi per consentire l'uso delle risorse associate nonostante il guasto. Tale meccanismo viene semplificato dall'uso di un disco condiviso. Quando un nodo viene riavviato dopo un guasto, il failover manager decide quali gruppi possono ritornare in esecuzione su di esso tramite un'operazione di *fallback*. Tale operazione viene eseguita per salvaguardare un uso efficiente delle risorse. Le operazioni di handover e fallback possono essere effettuata anche manualmente.

Il traffico di rete in ingresso verso il cluster, viene distribuito tra i vari nodi tramite un'operazione di bilanciamento del carico. Al cluster viene assegnato un unico indirizzo IP, ma i messaggi in ingresso vengono inviati a tutti i nodi server del cluster. Quindi, in base al carico corrente, solo uno dei server accetta il messaggio e risponde. Quando un nodo smette di funzionare il suo carico viene distribuito tra gli altri nodi, e viceversa, quando un nuovo nodo viene aggiunto al cluster, la distribuzione del carico viene riconfigurata per trasferire parte del traffico in ingresso verso il nuovo nodo.

Il *cluster Sun* integra uno o più sistemi di calcolo operanti con il sistema operativo Solaris per fornire affidabilità e scalabilità dei servizi. L'affidabilità viene garantita tramite operazioni di *failover*, per mezzo delle quali i servizi in esecuzione su un nodo che smette di funzionare sono trasferiti in un altro nodo. La scalabilità viene invece garantita distribuendo il carico tra i vari server. Le tre componenti fondamentali di un cluster Sun sono: gestione globale dei processi, file system distribuito e rete. La gestione globale dei processi fornisce identificatori di processo globali su tutto il sistema. Ciò è utile per implementare la migrazione dei processi, con cui un processo viene trasferito da un nodo all'altro per bilanciare il carico computazionale tra i vari nodi, o per incrementare la velocità di calcolo. Il file system distribuito permette invece a un processo di continuare a usare gli stessi percorsi (*path name*) per accedere ai file anche dopo la migrazione.

### **Amoeba**

Amoeba è un sistema operativo distribuito sviluppato alla Vrije Universiteit, in Olanda, durante gli anni '80. Lo scopo principale di Amoeba consiste nel creare un ambiente di esecuzione distribuita trasparente all'utente che assomigli il più possibile a quanto fornito da un sistema operativo time-sharing tradizionale come, per esempio, Unix. In secondo luogo, Amoeba si prefigura come un ambiente adatto per la programmazione distribuita e parallela.

L'architettura di sistema di Amoeba è costituita da tre componenti principali: un terminale grafico, un processor pool, e servizi vari quali file server e print server. Il terminale grafico è composto da tastiera, mouse e monitor connessi a un calcolatore. Il processor pool presenta le caratteristiche descritte nel Paragrafo 16.2. Il microkernel di Amoeba viene eseguito sia su server, sia su processor pool, che su terminali, e svolge quattro funzioni.

1. Gestione di processi e thread.
2. Supporto per la gestione della memoria a basso livello.
3. Supporto per la comunicazione.
4. Gestione del sistema di I/O a basso livello.

Amoeba supporta l'esecuzione di thread a livello kernel e fornisce due protocolli di comunicazione: uno implementa il modello di comunicazione client-server tramite il meccanismo di *chiamata a procedura remota* (RPC), l'altro supporta la comunicazione di gruppo. Entrambi i protocolli si basano sul protocollo di rete ISO FLIP (*Fast Local Internet Protocol*) sottostante (Paragrafo 16.6.6) per la trasmissione dei messaggi.

In Amoeba, la maggior parte delle funzioni offerte dai kernel tradizionali come, per esempio, il boot del sistema, la creazione e la schedulazione dei processi, la gestione del file system, sono implementate da server esterni al microkernel. Tale approccio riduce la dimensione del microkernel e lo rende adatto a un'ampia gamma di sistemi di calcolo che si estende dai server ai pool processor. In Amoeba il concetto principale è costituito dagli *oggetti*. Questi vengono gestiti dai server e sono protetti con le tecniche descritte nel Paragrafo 15.7.

Quando un utente accede ad Amoeba, viene avviata una shell su uno degli host, mentre i comandi impartiti successivamente vengono eseguiti da processi creati in altri host del sistema. Le applicazioni sono distribuite tra i vari host e non esiste il concetto di macchina locale. Le risorse del sistema sono quindi altamente integrate tramite l'uso del modello di esecuzione basato sul concetto di pool processor. Quando un comando viene impartito, il sistema operativo alloca più di un pool processor per la sua esecuzione, e quando necessario, i vari pool processor sono condivisi tra più utenti.

## **16.4 Comunicazione affidabile tra processi**

In un sistema operativo convenzionale, la comunicazione avviene tra processi che

eseguono sullo stesso host e sono distinti da un identificatore unico assegnato dallo stesso kernel. Al contrario, in un sistema distribuito, la comunicazione può avvenire tra processi che non sono in esecuzione sullo stesso nodo. Pertanto, un sistema operativo distribuito deve poter assegnare nomi univoci a tutti processi in esecuzione nei vari nodi e deve essere in grado di implementare un meccanismo per identificare la loro locazione all'interno del sistema. Tali aspetti sono trattati nel Paragrafo 16.4.1.

Identificare la locazione di un processo è fondamentale per poter implementare un meccanismo di scambio di messaggi tramite la rete. Tuttavia, in alcuni casi, la trasmissione di un messaggio può fallire a causa di guasti sui collegamenti di rete o sui nodi intermedi del cammino che unisce processo mittente e destinatario. Pertanto, i processi devono implementare un opportuno meccanismo per garantire una comunicazione affidabile. Tale meccanismo si basa su un protocollo di comunicazione inter-processo (*protocollo IPC*) costituito da un insieme di regole e convenzioni per la gestione dei guasti che possono presentarsi durante la trasmissione del messaggio. I processi mittente e destinatario invocano le routine del protocollo quando eseguono chiamate alle funzioni *send* e *receive*. Tali routine effettuano tutte le operazioni necessarie per garantire affidabilità durante la trasmissione dei messaggi.

La **Tabella 16.2** riassume i tre meccanismi principali di un protocollo IPC, ovvero *acknowledgment*, *timeout* e *ritrasmissione*. Un acknowledgment informa il processo mittente dell'avvenuta ricezione del messaggio da parte del processo destinazione. Se invece, il mittente non riceve l'acknowledgment in un intervallo di tempo predefinito, scatta il timeout e il messaggio viene ritrasmesso.

| Meccanismo                     | Descrizione                                                                                                                                                                                                                          |
|--------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Acknowledgment                 | Alla ricezione di un messaggio, il protocollo di comunicazione prevede l'invio al mittente da parte del destinatario di un messaggio di ricevuta, chiamato <i>acknowledgment</i> .                                                   |
| Timeout                        | Il protocollo di comunicazione specifica un intervallo di tempo entro cui il mittente di un messaggio deve ricevere l'acknowledgment. Allo scadere del tempo, in assenza di acknowledgment scatta un evento di tipo <i>timeout</i> . |
| Ritrasmissione di un messaggio | All'occorrenza di un timeout, il protocollo di comunicazione prevede la ritrasmissione del messaggio da parte del mittente.                                                                                                          |

**Tabella 16.2** Meccanismi per l'affidabilità in un sistema distribuito.

Il protocollo di comunicazione è implementato come segue. Quando un processo invia un messaggio, la routine del protocollo invocata dal processo esegue una chiamata di sistema per richiedere la generazione di un'interruzione (*timeout interrupt*) alla scadenza di un intervallo di tempo predefinito. Successivamente, quando il messaggio viene ricevuto dal processo destinazione, quest'ultimo invoca una routine per inviare l'acknowledgment al mittente. Tuttavia, se nel sito del mittente viene generato un timeout interrupt prima della ricezione dell'acknowledgment, il messaggio viene ritrasmesso e il mittente reimposta la richiesta per la generazione di un nuovo timeout interrupt. Un meccanismo analogo viene implementato per garantire la trasmissione affidabile di un eventuale messaggio di risposta da parte del destinatario verso il mittente. I protocolli di comunicazione inter-processo sono trattati nei Paragrafi 16.4.2 e 16.4.3.

### 16.4.1 Naming dei processi

Tutte le entità di un sistema distribuito, siano esse processi o risorse, vengono identificate da un nome unico in base al seguente meccanismo. Ogni host viene etichettato con un nome unico (*host\_name*), numerico o simbolico, all'interno del sistema distribuito, e ogni processo o risorsa collocata in un host viene etichettata con un identificatore unico (*process\_id*) all'interno dell'host. Pertanto, la coppia (*<host\_name>*, *<process\_id>*) può essere usata per identificare un processo in modo univoco. In tal modo, un processo che intende inviare un messaggio a un altro processo può usare la coppia (*risorse\_umane*, *P<sub>j</sub>*) come nome del processo destinatario, dove *risorse\_umane* indica

per esempio il nome dell'host su cui gira il processo  $P_j$ .

Per localizzare facilmente un host nella rete Internet, quest'ultima è partizionata in un insieme di *domini* con nomi unici, ognuno dei quali può essere partizionato in domini più piccoli anch'essi con nomi unici, e così via. Il nome di un host è unico all'interno di un dominio, ma potrebbe non esserlo in Internet, pertanto un host viene identificato in modo univoco concatenando i nomi dei domini a cui appartiene, separati dal carattere punto, partendo dal dominio più piccolo fino ad arrivare a quello più grande. Per esempio, il nome `Everest.cse.iitb.ac.in` si riferisce al server `Everest`, collocato all'interno del dipartimento di Informatica e Ingegneria dell'Indian Institute of Technology di Bombay, che si trova nel dominio riservato alle università indiane.

Lo spazio dei nomi di dominio è organizzato in modo gerarchico; il livello superiore della gerarchia è occupato da un dominio radice privo di nome. Tale dominio contiene un piccolo numero di *domini di primo livello* che rappresentano organizzazioni di tipo specifico od organizzazioni nazionali. Nel nome `Everest.cse.iitb.ac.in`, "in" è il dominio di primo livello che rappresenta l'India e "ac" è il nome del dominio che contiene le università. Quindi, "ac.in" è un dominio di secondo livello usato per le università indiane.

Ogni nodo connesso a Internet è contraddistinto da un indirizzo unico noto come indirizzo IP. Il *domain name system* (DNS) è un servizio della rete Internet che restituisce gli indirizzi IP corrispondenti ai nomi associati agli host. Il DNS è costituito da un *name server* per ogni dominio contenente una directory che memorizza gli indirizzi IP di ogni nodo del dominio. Quando un processo in esecuzione in un host  $h_i$  desidera inviare un messaggio a un altro processo ( $<\text{host\_name}>$ ,  $<\text{process\_id}>$ ),  $h_i$  effettua un'operazione di *risoluzione del nome* per determinare l'indirizzo IP di  $<\text{host name}>$ . La risoluzione del nome da parte di  $h_i$ , detto processo risolutore, avviene come segue. Il risolutore conosce l'indirizzo del name server del dominio radice e gli invia il nome  $<\text{host\_name}>$ . Il name server radice risponde restituendo l'indirizzo IP del name server di primo livello associato a  $<\text{host\_name}>$ . Quindi, il risolutore invia la richiesta al name server di primo livello, che restituisce l'indirizzo IP del name server del dominio di secondo livello, e così via fino a quando un name server restituisce l'indirizzo IP dell'host richiesto.

La risoluzione del nome tramite name server può essere lenta, e pertanto ciascun risolutore può memorizzare localmente alcune informazioni acquisite dai name server. In tal modo la risoluzione di nomi risulta essere più veloce, esattamente come l'uso di una cache per le directory velocizza la ricerca l'accesso ai file contenuti in esse (Paragrafo 13.15). Un indirizzo IP può essere memorizzato in locale per un intervallo di tempo noto come *time to live*, che generalmente è fissato in un'ora. Il name server di un dominio è replicato per garantire affidabilità del servizio e prevenire la congestione.

### 16.4.2 Semantica dei protocolli di comunicazione

Per *semantica IPC* si intende l'insieme delle proprietà di un protocollo IPC. Essa dipende dal meccanismo di acknowledgment e ritrasmissione usato nel protocollo IPC. La [Tabella 16.3](#) riassume i tre tipi principali di semantica usata nei protocolli di comunicazione inter-processo.

| Semantica                      | Descrizione                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Semantica <i>at-most-once</i>  | Il processo destinatario riceve il messaggio al più una volta. Tale semantica si ha quando il ricevente di un messaggio non invia l'acknowledgment e il mittente non effettua la ritrasmissione dei messaggi.                                                                                                                                                   |
| Semantica <i>at-least-once</i> | Il processo destinatario riceve il messaggio almeno una volta. Tale semantica si ha quando il ricevente di un messaggio invia l'acknowledgment, e il mittente effettua la ritrasmissione del messaggio se non riceve l'acknowledgment prima del timeout.                                                                                                        |
| Semantica <i>exactly-once</i>  | Il processo destinatario riceve il messaggio esattamente una sola volta. Tale semantica si ha quando l'invio dell'acknowledgment e la ritrasmissione del messaggio viene effettuata seguendo la semantica <i>at-least-once</i> , ma il protocollo IPC riconosce messaggi duplicati e li scarta in modo da recapitare ogni messaggio esattamente una volta sola. |

**Tabella 16.3** Semantica IPC.

Si parla di semantica *at-most-once* quando non sono previsti né l'acknowledgment né la ritrasmissione. Tale semantica è utilizzata quando la perdita di un messaggio non rappresenta un serio rischio per la correttezza dell'applicazione, o quando l'applicazione stessa è in grado di risolvere il problema derivante dall'eventuale perdita di un messaggio. Per esempio, un'applicazione che riceve periodicamente informazioni da altri processi è in grado di rilevare l'eventuale perdita di un messaggio atteso e, in conseguenza di ciò, richiederne la ritrasmissione. Non usando acknowledgment e ritrasmissione esplicativi, la semantica *at-most-once* garantisce una comunicazione altamente efficiente.

Si parla di semantica *at-least-once*, invece, quando vengono usati sia il meccanismo di acknowledgment sia la ritrasmissione. In tal caso, il processo destinatario può ricevere lo stesso messaggio più di una volta, nel caso in cui l'acknowledgment vada perso o arrivi al processo mittente in ritardo a causa di problemi di congestione di rete. La semantica *at-least-once* può essere usata solo per operazioni invarianti che, pertanto, non vengono influenzate negativamente dalla ricezione di *messaggi duplicati*.

Infine, si parla di semantica *exactly-once* quando il protocollo di comunicazione usa acknowledgment e ritrasmissione, ma scarta automaticamente i messaggi duplicati. Tale semantica maschera i guasti transienti derivanti dalla perdita di messaggi inviati dal mittente o di acknowledgment spediti dal destinatario. Tuttavia, il costo di un protocollo che usa la semantica *exactly-once* è molto elevato a causa del tempo richiesto per la gestione di guasti e messaggi duplicati.

### 16.4.3 Protocolli IPC

Un protocollo IPC specifica le azioni che devono essere effettuate per garantire che un messaggio spedito da un processo mittente arrivi a destinazione, e che la successiva risposta da parte del processo destinatario arrivi al mittente. I paragrafi seguenti riportano una classificazione dei protocolli IPC e un paio di esempi.

#### Protocolli affidabili e non affidabili

Un *protocollo affidabile* garantisce che il messaggio e la sua risposta non vadano persi, tramite l'uso di una semantica di tipo *at-least-once* o *exactly-once*. Al contrario, un *protocollo non affidabile*, usando la semantica *at-most-once*, non garantisce che un messaggio e la relativa risposta giungano a destinazione. Conseguentemente, i protocolli affidabili hanno un costo superiore rispetto a quelli non affidabili dovuto alla gestione degli acknowledgment e dell'eventuale ritrasmissione dei messaggi.

#### Protocolli bloccanti e non bloccanti

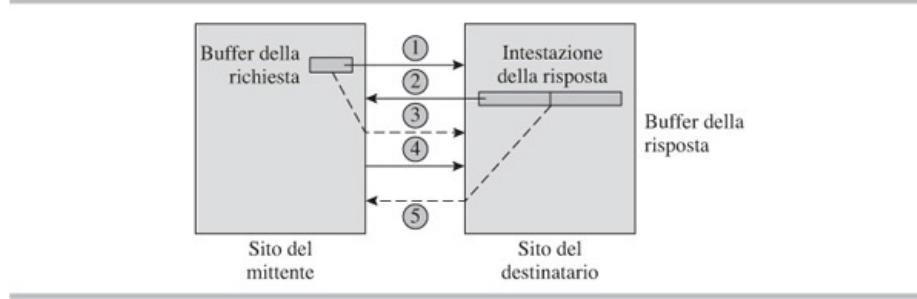
Come discusso nel [Capitolo 9](#), si è soliti bloccare un processo che esegue una chiamata di sistema di tipo *receive*, fino alla ricezione di un messaggio. Risulta meno evidente, invece, la necessità di bloccare un processo che esegue una chiamata di sistema di tipo

*send*. Tuttavia, bloccare il processo mittente può semplificare e ridurre il costo del protocollo di comunicazione adottato e migliorarne la relativa semantica. Per esempio, se un processo mittente rimane bloccato finché il messaggio inviato non viene ricevuto dal processo destinatario, non è necessario che il protocollo bufferizzi il messaggio. Inoltre, il blocco del mittente permette di implementare una semantica simile a quella adottata nelle tradizionali chiamate di funzione.

Un protocollo è *bloccante* se, dopo la spedizione di un messaggio, il processo mittente rimane bloccato fino a quando riceve una risposta; in caso contrario, il protocollo è di tipo *non bloccante*. Si assume che, nel caso di protocollo non bloccante, la notifica al mittente dell'avvenuta ricezione del messaggio da parte del destinatario avvenga tramite interrupt, per permettere al mittente di effettuare eventuali azioni conseguenti. I protocolli bloccanti e non bloccanti sono chiamati, rispettivamente, anche protocolli *sincroni* e *asincroni*.

### **Il protocollo di richiesta-risposta-acknowledgment**

Il protocollo di richiesta-risposta-acknowledgment (RRA) è un protocollo che permette lo scambio di messaggi di richiesta e risposta in modo affidabile. La ricezione del messaggio da parte del destinatario viene notificata al mittente tramite l'invio della relativa risposta, pertanto, non è necessario l'invio esplicito di un acknowledgment. Al contrario, il mittente invia un acknowledgment al destinatario per notificare la ricezione della risposta. L'Algoritmo 16.1 implementa una versione bloccante del protocollo RRA e la [Figura 16.3](#) schematizza le relative operazioni.



**Figura 16.3** Versione bloccante del protocollo RRA.

#### **Algoritmo 16.1 (Versione bloccante del protocollo RRA)**

1. *Quando un processo effettua una richiesta:* la richiesta viene copiata in un buffer locale e inviata tramite messaggio al processo destinatario. Quindi, viene effettuata una chiamata di sistema per richiedere l'invio di un interrupt alla scadenza di un timeout. Il processo mittente viene bloccato in attesa di ricevere la risposta dal processo destinatario.
2. *Quando il processo destinatario riceve un messaggio:* il processo analizza la richiesta contenuta nel messaggio e prepara la risposta corrispondente. La risposta viene copiata in un buffer locale e inviata al processo mittente. Quindi, viene effettuata una chiamata di sistema per richiedere l'invio di un interrupt alla scadenza di un timeout.
3. *Quando scade il timeout nel sito del processo mittente:* la copia della richiesta memorizzata nel buffer viene ritrasmessa.
4. *Quando il processo mittente riceve la risposta:* il processo mittente invia un acknowledgment al processo destinatario e svuota il buffer contenente la richiesta.
5. *Quando scade il timeout nel sito del processo destinatario:* la copia della risposta memorizzata nel buffer viene ritrasmessa.
6. *Quando il processo destinatario riceve l'acknowledgment:* il processo destinatario svuota il buffer contenente la risposta.

Poiché il processo mittente rimane bloccato finché non riceve la risposta, è sufficiente avere un unico buffer per memorizzare la richiesta nel sito del mittente, indipendentemente dal numero di messaggi inviati. Il processo destinatario, invece, non viene bloccato in attesa di ricevere l'acknowledgment, e quindi può gestire richieste da

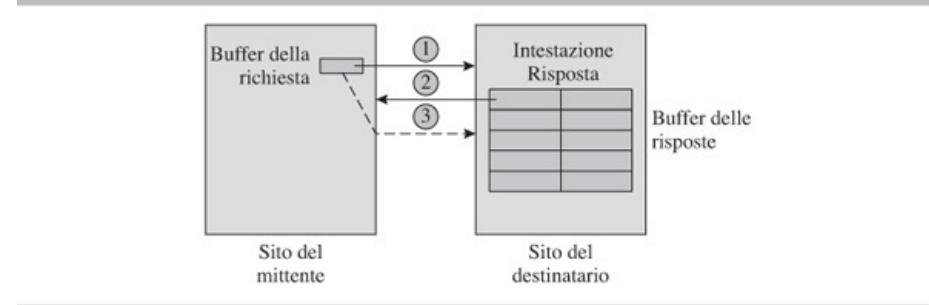
altri processi mentre attende l'arrivo dell'acknowledgment. Pertanto, il sito del destinatario necessita di un buffer per memorizzare i messaggi di risposta per ogni processo mittente.

Qualora la comunicazione tra mittente e destinatario preveda più di uno scambio di messaggi di tipo richiesta-risposta, il numero di messaggi di acknowledgment può essere ridotto usando il *piggybacking*. Tale tecnica si basa sul fatto che l'invio di una richiesta da parte del mittente è implicitamente una conferma della ricezione della risposta precedente, visto che il mittente rimane bloccato finché non riceve la risposta. Quindi, l'acknowledgment di una risposta viene implicitamente incluso dal mittente nel messaggio corrispondente alla richiesta successiva. In tal modo, il mittente deve inviare un apposito messaggio di acknowledgment solo per confermare la ricezione dell'ultima risposta.

Il protocollo RRA usa la semantica at-least-once in quanto richieste e corrispondenti risposte non possono andare perse; tuttavia, esse potrebbero essere ricevute più di una volta. Come riportato in [Tabella 16.3](#), per ottenere una semantica di tipo exactly-once il destinatario deve essere in grado di scartare le richieste duplicate. Ciò può essere ottenuto come segue. Il mittente etichetta ciascuna delle sue richieste con numeri sempre crescenti. Il numero assegnato a una richiesta viene copiato anche nella risposta e nell'acknowledgment corrispondenti, nonché nel campo intestazione (*header*) del buffer locale preservato dal sito del destinatario per le risposte pendenti. Anche il processo destinatario memorizza il numero dell'ultima richiesta ricevuta da parte del processo mittente. Alla ricezione di una richiesta, se il suo numero di sequenza è minore o uguale a quello memorizzato dal destinatario, significa che la richiesta è un duplicato di un messaggio già ricevuto. In tal caso, il processo destinatario controlla il buffer locale delle risposte pendenti. Se quest'ultimo contiene già una copia della risposta corrispondente alla richiesta duplicata, tale copia viene ritrasmessa al processo mittente. In caso contrario, o la copia della risposta è già stata rimossa dal buffer a causa della ricezione del corrispondente acknowledgment, e quindi si tratta di una richiesta vecchia che non va più tenuta in considerazione, oppure il processo destinatario sta ancora gestendo la richiesta, e quindi invierà la risposta successivamente. In entrambe le situazioni, la richiesta duplicata viene semplicemente scartata.

### **Il protocollo di richiesta-risposta**

Il protocollo di richiesta-risposta (RR) effettua la ritrasmissione di una richiesta quando scade un timeout. L'Algoritmo 16.2 mostra una versione non bloccante del protocollo RR che garantisce la semantica di tipo exactly-once, mentre la [Figura 16.4](#) schematizza le corrispondenti operazioni.



**Figura 16.4** Versione non bloccante del protocollo RR.

#### **Algoritmo 16.2 (Versione non bloccante del protocollo RR)**

1. *Quando un processo effettua una richiesta:* la richiesta viene copiata in un buffer locale e inviata in un messaggio al processo destinatario. Successivamente, viene effettuata una chiamata di sistema per richiedere l'invio di un interrupt alla scadenza di un timeout. Il mittente prosegue quindi la sua elaborazione.
2. *Quando il processo destinatario riceve un messaggio:* se il messaggio non è il duplicato di una richiesta già ricevuta, il processo destinatario analizza la richiesta contenuta nel messaggio, prepara la risposta, la copia in un buffer locale e la invia al processo mittente. Altrimenti, se si tratta di una richiesta duplicata, il

- destinatario recupera la risposta dal buffer locale e la inoltra nuovamente al mittente.
3. *Quando scade il timeout nel sito del processo mittente:* la copia della richiesta memorizzata nel buffer locale viene ritrasmessa.
  4. *Quando il sito del processo mittente riceve una risposta:* viene generato un interrupt per notificare al processo mittente l'arrivo della risposta. Il mittente, di conseguenza, svuota il buffer locale.

Il processo mittente non invia esplicitamente l'acknowledgment della risposta. A differenza del protocollo RRA, l'acknowledgment non è nemmeno implicitamente inserito in un'eventuale successiva richiesta del mittente, in quanto quest'ultimo potrebbe aver inviato la nuova richiesta prima di avere ricevuto la risposta della sua richiesta precedente. Di conseguenza, il processo destinatario deve preservare sempre una copia delle sue risposte, necessitando pertanto di una grossa quantità di spazio per la bufferizzazione.

Se le richieste del processo mittente sono ricevute dal processo destinatario nello stesso ordine di invio, è possibile usare una versione semplificata del meccanismo adottato dal protocollo RRA per riconoscere e gestire gli eventuali messaggi duplicati. In tal caso, infatti, è sufficiente che il processo destinatario preservi in opportuni buffer le risposte di tutte le richieste pendenti e i relativi numeri di sequenza. A fronte di una richiesta duplicata, il destinatario cerca la corrispondente risposta nell'insieme dei buffer locali usando il numero di sequenza. Se la risposta viene trovata, il destinatario la ritrasmette al mittente. In caso contrario, il destinatario ignora la richiesta, visto che la corrispondente risposta verrà inviata appena il processo avrà terminato di elaborarla. Il Problema 16.5 propone di raffinare questo approccio per adattarlo al caso in cui le richieste possano essere ricevute in un ordine diverso rispetto a quello di invio.

Il protocollo precedente può essere semplificato nel caso venga utilizzato in applicazioni che prevedono l'uso di *operazioni idempotenti*. Un'operazione idempotente è caratterizzata dal fatto che il suo risultato è indipendente dal numero di volte che viene eseguita. Per esempio, l'operazione  $i := 5$  è idempotente, mentre l'operazione  $i := i + 1$  non lo è. Quando un'applicazione è composta solo da operazioni idempotenti, la consistenza dei dati non è influenzata se una richiesta viene processata più di una volta. In tal caso, non è più necessario l'uso di un meccanismo per la gestione delle richieste duplicate basato su bufferizzazione delle risposte ed eliminazione dei messaggi già ricevuti. Le operazioni di lettura e scrittura su file sono idempotenti, quindi è possibile usare una versione semplificata dal protocollo RR per implementare un file server remoto. In tal modo, il file server non ha bisogno di memorizzare informazioni relative allo stato delle richieste già servite, per cui può fornire un servizio *stateless* e più affidabile (Paragrafo 20.4.3).

## 16.5 Paradigmi di calcolo distribuito

I dati usati in un'applicazione possono essere memorizzati su diverse macchine di un sistema distribuito, in base alle seguenti considerazioni:

- *replica dei dati:* diverse copie di un insieme di dati  $D$  possono essere memorizzate su siti diversi del sistema per garantire disponibilità e accessi efficienti;
- *distribuzione dei dati:* componenti diverse di un insieme di dati  $D$  possono essere memorizzate su varie macchine del sistema, perché le informazioni da memorizzare sono molto voluminose, oppure perché le varie componenti sono generate e/o frequentemente usate su siti diversi.

Quando un insieme di dati  $D$  non viene né replicato né distribuito, il sistema operativo deve identificare opportunamente il sito dove memorizzarlo, in modo che il traffico di rete totale generato dalle applicazioni che accedono a  $D$  sia il minimo possibile. La [Tabella 16.4](#) riassume tre modalità di accesso ai dati in un sistema distribuito. Nella modalità di *accesso remoto*, i dati vengono letti o scritti sul sito in cui risiedono. Questa modalità non interferisce con le decisioni relative alla locazione dei dati; tuttavia, l'accesso risulta lento a causa della latenza di rete. La *migrazione dei dati* consiste nello spostare i dati dalla locazione originale verso la macchina in cui è in esecuzione il processo che li vuole usare. Tale modalità risulta difficoltosa se gli stessi dati sono usati da molte applicazioni. Nel caso peggiore, la migrazione di un dato potrebbe portare al

caso in cui una sola applicazione alla volta è in grado di usarlo. *La migrazione del processo computazionale* prevede che l'elaborazione venga spostata sulla macchina dove si trovano i dati che devono essere usati. In tal caso, non si ha interferenza né con eventuali copie replicate dei dati, né con tecniche che prevedono la distribuzione di un insieme di dati in siti diversi.

| Modalità di accesso                    | Descrizione                                                                                                                                                                                                                                                                     |
|----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Accesso remoto ai dati                 | L'applicazione accede ai dati tramite la rete. Questa modalità di accesso non interferisce con l'organizzazione dei dati e non richiede modifiche dell'applicazione. Tuttavia, l'esecuzione dell'applicazione è più lenta a causa dei ritardi dovuti alla comunicazione remota. |
| Migrazione dei dati                    | I dati vengono trasferiti verso il sito in cui è in esecuzione il processo che li deve usare. La migrazione dei dati fornisce un accesso ai dati efficiente, ma può interferire con la replica e la distribuzione dei dati.                                                     |
| Migrazione del processo computazionale | Un processo di calcolo viene trasferito verso il sito dove risiedono i dati che deve utilizzare. La migrazione del processo computazionale fornisce un accesso ai dati efficiente senza interferire con la loro organizzazione.                                                 |

**Tabella 16.4** Modalità di accesso ai dati in un sistema distribuito.

I sistemi operativi permettono di accedere ai dati per mezzo delle modalità riassunte in [Tabella 16.4](#). Come descritto nel Paragrafo 16.3, un sistema operativo di rete supporta l'accesso remoto dei dati. Il *File Transfer Protocol* (FTP) invece è un protocollo che permette la migrazione dei dati; in particolare, FTP consente di trasferire file mentre non sono in uso. La *migrazione dei processi* viene utilizzata, infine, per trasferire un'elaborazione o parte di essa, tra i nodi del sistema durante la sua esecuzione. La migrazione dei processi verrà descritta nel Paragrafo 18.8.2.

Un'*elaborazione distribuita* è costituita da parti che possono essere eseguite su macchine diverse per svariate ragioni: maggior efficienza nell'accesso ai dati, incremento della velocità di calcolo, affidabilità. Un *paradigma per il calcolo distribuito* è invece un modello di pratiche utili per progettare applicazioni distribuite. Gli aspetti principali affrontati da un tale paradigma sono la manipolazione dei dati e l'attivazione di sottoparti dell'elaborazione nei vari siti di un sistema distribuito. La [Tabella 16.5](#) riassume le caratteristiche di tre paradigmi di calcolo distribuito.

| Paradigmi                         | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Elaborazione Client-server        | Un processo server fornisce un servizio specifico a un insieme di processi client. Ogni client invoca il servizio del server inviandogli un messaggio. Il server fornisce il risultato richiesto rispondendo con un ulteriore messaggio. Le applicazioni usano ampiamente il paradigma client-server per effettuare accessi o manipolazioni remote dei dati.                                                                                                         |
| Chiamata a procedura remota (RPC) | Una procedura remota assomiglia a una procedura convenzionale a eccezione del fatto che viene eseguita su un nodo diverso rispetto a quello in cui la relativa chiamata di procedura è stata effettuata. Una procedura remota viene installata in un nodo dall'amministratore di sistema e viene registrata su un name server. Le chiamate a procedura remota vengono usate spesso per implementare la migrazione dell'elaborazione.                                 |
| Remote evaluation                 | La remote invocation si basa su istruzioni del tipo <code>at &lt;node&gt; eval &lt;code_segment&gt;</code> . Essa prevede che il codice contenuto in <code>&lt;code_segment&gt;</code> venga trasferito ed eseguito nel nodo indicato da <code>&lt;node&gt;</code> . Dopodiché il risultato viene restituito al nodo di partenza. Non è necessario installare il segmento di codice nel nodo remoto. Il linguaggio Java permette di effettuare la remote evaluation. |

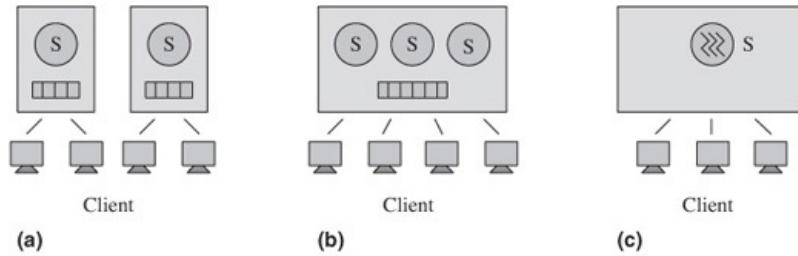
**Tabella 16.5** Paradigmi di calcolo distribuito.

Il paradigma di calcolo *client-server* si focalizza sulla manipolazione e sull'accesso remoto dei dati, mentre i paradigmi di *chiamata a procedura remota* e *remote evaluation* forniscono modi diversi per implementare la migrazione del calcolo.

### 16.5.1 Elaborazione client-server

Un *server* è un processo che fornisce un servizio specifico ai suoi processi client all'interno di un sistema distribuito. Qualunque processo può inviare un messaggio a un server e, pertanto, diventare un suo client. Il servizio offerto può avere una connotazione fisica, come per esempio accedere o stampare un file, oppure una connotazione computazionale, come per esempio calcolare alcune funzioni matematiche. Da ciò consegue che il ruolo di un server può variare all'interno di un ampio spettro di attività: dal semplice accesso a dati fino alla manipolazione dei dati. In quest'ultimo caso, il server può quindi giocare un ruolo fondamentale nelle attività di un'applicazione distribuita.

Un server può diventare un collo di bottiglia per il sistema nel caso in cui il tasso di richiesta dei client sia particolarmente elevato. La [Figura 16.5](#) rappresenta tre modi diversi per affrontare tale problema. Il primo meccanismo, mostrato in [Figura 16.5\(a\)](#), consiste nel suddividere i client in modo che ciascuno possa usare uno solo dei possibili server che forniscono il medesimo servizio. Così facendo, ogni server avrà una propria coda di richieste. Tale soluzione, tuttavia, può portare al caso in cui alcuni server siano particolarmente carichi, mentre altri rimangono inoperosi. Una soluzione alternativa, mostrata in [Figura 16.5\(b\)](#), prevede che molti server condividano in modo dinamico la stessa coda di richieste. In tal modo si ottiene un maggior livello di flessibilità rispetto a suddividere i client per usare sempre server diversi. Infine, la [Figura 16.5\(c\)](#) mostra un server multithread in cui ogni richiesta viene gestita da un thread separato appositamente creato. I vari thread competono tra loro per l'uso della CPU e delle altre risorse. Pertanto, se la funzionalità svolta dal server è di tipo I/O-bound, il sistema può sovrapporre l'esecuzione di più richieste contemporaneamente. Un altro modo per evitare che il server diventi un collo di bottiglia consiste chiaramente nel mantenere la maggior parte del carico computazionale nel processo client, per permettere al server di avere tempi di risposta più brevi.



**Figura 16.5** (a) Server con code indipendenti, (b) server con coda condivisa, (c) server basato su thread.

L'elaborazione client-server risulta un paradigma non particolarmente efficiente per il calcolo distribuito, in quanto non sono state sviluppate metodologie adeguate per strutturare applicazioni distribuite nella forma client-server. La mancanza di efficienza è dovuta principalmente al fatto che le entità coinvolte in un'applicazione distribuita sono generalmente in relazione simmetrica tra loro, e tale relazione è difficilmente modellabile usando il paradigma client-server. Infatti, il paradigma client-server è principalmente usato per implementare funzionalità su una rete LAN che non richiedono sforzo computazionale, quali, per esempio, l'accesso a file o semplici interrogazioni su basi di dati. In un paragrafo successivo si mostrerà come tali funzionalità possano essere implementate in modo efficiente utilizzando semplici protocolli come, per esempio il protocollo RR, piuttosto che il protocollo multilivello ISO.

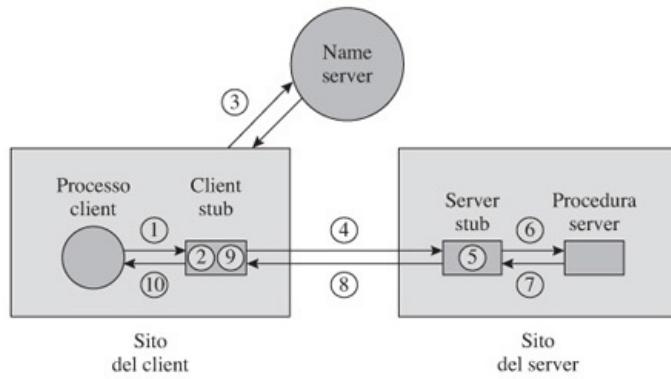
### 16.5.2 Chiamate a procedura remota

Una *chiamata a procedura remota* (RPC) è una caratteristica fornita dai linguaggi di programmazione per il calcolo distribuito. Come discusso precedentemente nel Paragrafo 8.4.2, sintassi e semantica di una RPC assomigliano a quelle di una tradizionale chiamata di procedura. Nella RPC:

`call <proc_id> (<message>)`

*<proc\_id>* è l'identificatore della procedura remota, e *<message>* è la lista dei parametri. La chiamata è implementata usando un protocollo bloccante. Il risultato della chiamata può essere restituito in uno dei parametri, o attraverso un valore di ritorno esplicito. Possiamo vedere la relazione tra il processo chiamante e la procedura chiamata come una relazione client-server, dove la procedura remota funge da server e il processo chiamante da client.

Lo schema di [Figura 16.6](#) mostra il meccanismo utilizzato per effettuare la risoluzione del nome, il passaggio dei parametri e il ritorno del risultato durante una chiamata a procedura remota. L'indirizzo IP del processo chiamato viene recuperato tramite il *domain name system* (DNS) descritto nel Paragrafo 16.4.1. Le funzioni del client stub e del server stub sono state descritte precedentemente nel Paragrafo 9.4.2 – il *client stub* converte i parametri passati alla procedura remota in un formato indipendente dalla macchina, mentre il *server stub* li riporta in una rappresentazione specifica per la macchina in cui la procedura viene eseguita. Al contrario, client stub e server stub giocano il ruolo opposto per la gestione del risultato della procedura. I numeri cerchietti in [Figura 16.6](#) identificano i passi effettuati durante l'esecuzione di una procedura remota.



**Figura 16.6** Implementazione di una chiamata a procedura remota (RPC).

1. Il processo client chiama il client stub con gli opportuni parametri usando il classico meccanismo di chiamata di procedura. Quindi, l'esecuzione del processo client viene sospesa in attesa che la chiamata venga completata.
2. Il client stub impacchetta i parametri tramite un'operazione nota come *marshaling*, li converte in un formato indipendente dalla macchina e li inserisce in un messaggio.
3. Il client stub interagisce con il name server per individuare l'identità della macchina dove risiede la procedura remota.
4. Il client stub invia il messaggio preparato al passo 2 alla macchina in cui risiede la procedura remota, usando un protocollo bloccante. Pertanto, il client stub, dopo l'invio, rimane bloccato in attesa di ricevere una risposta al suo messaggio.
5. Il server stub riceve il messaggio inviato dal client stub, lo spacchetta tramite un'operazione di *unmarshaling*, e converte i parametri nel formato specifico della macchina del server.
6. Il server stub effettua una chiamata convenzionale della procedura server, che implementa la funzionalità richiesta dal processo client, inviando i parametri estratti al passo precedente. Quindi, il server stub viene sospeso in attesa che la procedura termini la sua esecuzione.
7. La procedura server restituisce il risultato dell'elaborazione al server stub, il quale lo converte in un formato indipendente dalla macchina, esegue il marshaling e prepara il messaggio da inviare al client stub.
8. Il server stub invia il messaggio verso la macchina del client.
9. Il client stub riceve il messaggio, esegue l'unmarshaling e converte il risultato in un formato adatto alla macchina locale.
10. Il client stub inoltra il risultato al processo client che aveva inizialmente eseguito la chiamata a procedura remota.

Il passo 10 completa l'esecuzione della chiamata della procedura remota, e il processo client può continuare la sua esecuzione.

Nel passo 3, il client stub non ha bisogno di effettuare la risoluzione del nome a ogni esecuzione della stessa RPC. Al contrario, dopo la prima chiamata, il client può memorizzare nella cache le informazioni relative alla macchina su cui risiede la procedura remota per usi futuri. La risoluzione del nome può essere fatta anche in modo statico, ovvero prima dell'inizio dell'attività del processo client.

Durante una chiamata della procedura remota possono verificarsi dei guasti - nella connessione di rete, nel server o nello stesso client. Se la macchina client cessa di funzionare, la chiamata diventa un *orphan* e il suo risultato non è più utile a nessuno. Guasti nella connessione e nel server possono essere, invece, gestiti usando meccanismi basati su acknowledgment e ritrasmissione (Paragrafo 16.4). Idealmente, le chiamate di procedure remote dovrebbero adottare una semantica di tipo exactly-once; tuttavia, l'implementazione di una tale semantica è molto costosa. Al contrario, la semantica at-least-once è meno costosa, ma richiede che l'azione svolta dalla procedura remota sia idempotente o che questa sia in grado di scartare richieste duplicate.

La chiamata della procedura remota può essere usata come componente principale per il calcolo distribuito. Rispetto al paradigma client-server, la RPC presenta due vantaggi principali. In primo luogo, è possibile rendere operativa una procedura remota

semplicemente dichiarando il suo nome e la sua locazione al name server, attività molto più semplici rispetto all'installazione di un server. Inoltre, solo i processi che conoscono l'esistenza della procedura remota possono invocarla. Pertanto, l'uso di RPC garantisce una maggior sicurezza rispetto all'uso del paradigma client-server. Lo svantaggio principale di una RPC consiste, invece, nella mancanza di flessibilità – una procedura remota deve essere registrata sul name server, e quindi la sua locazione non può essere facilmente cambiata.

### 16.5.3 Remote evaluation

Il paradigma noto come remote evaluation è stato proposto da Stamos e Gifford (1990) ed è implementato attraverso l'istruzione:

```
at <node> eval <code_segment>
```

dove *<node>* è un'espressione che valuta l'identità di un nodo nel sistema distribuito, e *<code\_segment>* è un segmento di codice, generalmente costituito da una sequenza di istruzioni. Quando in un processo si esegue l'istruzione *at*, l'espressione *<node>* viene valutata per ottenere l'identità di un nodo, *<code\_segment>* viene eseguito in quel nodo, e il suo risultato, se ve ne è uno, viene restituito al processo.

La remote evaluation ha parecchi vantaggi rispetto ai paradigmi client-server e RPC. Richiede minimo supporto da parte del sistema operativo. La maggior parte del lavoro viene effettuata dal compilatore del linguaggio in cui il programma è stato scritto. Con l'aiuto del sistema operativo, il compilatore fa in modo che il codice *<code\_segment>* venga trasferito ed eseguito nel nodo destinatario. Il sistema operativo di tale nodo crea un processo ad hoc per eseguire il codice ricevuto e restituire il corrispondente risultato. Non è necessario né installare il segmento di codice *<code\_segment>* nel nodo destinazione, né utilizzare stub come nel caso delle chiamate di procedure remote.

La gestione della risoluzione dei nomi e del binding sono molto più semplici quando si usa la remote evaluation rispetto a un ambiente basato su RPC. La decisione su quale nodo debba essere usato per eseguire il segmento di codice viene presa in modo dinamico in base al carico computazionale dei vari nodi. *<code\_segment>* può essere costituito da una sezione di codice qualsiasi che possa essere eseguita in remoto; non ha bisogno di essere scritta con la sintassi tipica delle procedure. La remote evaluation può essere usata assieme ai paradigmi client-server ed RPC, ovvero il segmento di codice può invocare procedure durante la sua esecuzione o può esso stesso essere una procedura.

La remote evaluation può essere usata per incrementare la velocità di calcolo o per migliorare la sua efficienza. Per esempio, se parte di un'elaborazione richiede la manipolazione di una notevole quantità di dati che risiedono su un nodo remoto  $S_i$ , è possibile trasferire la sua esecuzione su  $S_i$ . Ciò permette di ridurre il traffico di rete necessario per accedere ai dati da manipolare. Similmente, se un utente desidera inviare un'email a un certo numero di persone che stanno usando  $S_i$ , il comando per spedire l'email può essere eseguito su  $S_i$ .

### 16.5.4 Casi di studio

#### RPC SUN

Le RPC Sun sono state progettate per la comunicazione client-server su NFS (Network File System), il file system di rete di Sun. NFS gestisce i file per mezzo di azioni idempotenti, pertanto le RPC di Sun adottano la semantica at-least-once. Ciò, da un lato, rende le RPC efficienti, ma dall'altro, richiede che le applicazioni che usano RPC gestiscano da sole eventuali richieste duplicate qualora si voglia adottare una semantica di tipo exactly-once.

Le RPC Sun si basano su un linguaggio, XDR, e un compilatore, *rpcgen*, specifici. Per usare una procedura remota, infatti, l'utente deve definirne un'opportuna interfaccia tramite XDR. Tale interfaccia contiene la specifica della procedura remota e dei suoi parametri. Quindi, l'interfaccia viene compilata tramite *rpcgen* che produce: un client stub, la procedura server, il server stub, un header file da includere nei programmi client e server, e due procedure per la gestione dei parametri che vengono invocate dal client stub e dal server stub. Sia il programma client che il programma server vengono compilati assieme ai relativi header file e stub. La procedura per la gestione dei parametri invocata dal client stub esegue il marshaling dei parametri e li converte in un

formato indipendente dalla macchina chiamato external data representation (XDR). La procedura che gestisce i parametri invocata dal server stub, invece, converte i parametri dal formato XDR al formato richiesto dalla macchina da cui è stata effettuata la chiamata a procedura.

Le RPC Sun hanno però alcune limitazioni. Innanzitutto, la procedura remota può usare un solo parametro. Tale limitazione può comunque essere facilmente aggirata passando come parametro una struttura contenente nei suoi campi tutti i dati desiderati. Una seconda limitazione consiste nel fatto che l'implementazione delle RPC non fa uso di un name server. Al contrario, ogni macchina contiene un *port mapper* che agisce come name server locale per memorizzare i nomi delle procedure e gli identificatori delle porte a cui le procedure sono associate. Il client, pertanto, prima effettua una richiesta al port mapper del sito remoto per trovare la porta usata dalla procedura remota di interesse, quindi, chiama la procedura tramite la sua porta. Lo svantaggio di tale meccanismo è dovuto al fatto che il chiamante deve conoscere il sito dove risiede la procedura remota.

### **Java Remote Method Invocation (RMI)**

L'invocazione di metodi remoti di Java, nota come Java RMI (*Java Remote Method Invocation*), si basa sul principio che un'applicazione server in esecuzione su una macchina possa creare un tipo di oggetto speciale chiamato *oggetto remoto* i cui metodi possano essere invocati da client in esecuzione su altre macchine. Il server sceglie un nome per il servizio offerto da ogni metodo dell'oggetto remoto e lo registra tramite un name server, chiamato `rmiregistry`, in esecuzione sulla stessa macchina del server. Il server `rmiregistry` tipicamente è in ascolto su una porta standard, per gestire eventuali richieste di registrazione e di invocazione dei metodi. I client conoscono l'indirizzo IP della macchina server e pertanto possono consultare il relativo `rmiregistry` per localizzare i servizi desiderati. Conseguentemente a una richiesta da parte di un client, `rmiregistry` restituisce un riferimento all'oggetto remoto che fornisce il servizio richiesto. Tramite tale riferimento, il client può quindi invocare il metodo desiderato. La sintassi per invocare il metodo remoto è simile a quella usata per accedere a un oggetto locale e rispecchia quella descritta nel Paragrafo 16.5.2. Per compilare il codice sorgente contenente i programmi client e server viene usato il compilatore `javac`, mentre per generare gli stub del client e del server viene usato il compilatore `rmic`.

Il client può passare come parametri di un metodo remoto anche oggetti di tipo speciale, detti *serializable*. Il codice di tali oggetti viene caricato nella macchina server durante l'operazione di unmarshaling dei parametri, e può essere invocato dall'oggetto che offre il servizio remoto. In tal modo è possibile ottenere un effetto analogo alla remote evaluation descritta nel Paragrafo 16.5.3 come segue: un server registra un servizio remoto `r_eval` che necessita di un oggetto serializable `alpha` come parametro e chiama il metodo `alpha.gamma`. Quando un client crea un oggetto serializable e lo passa come parametro a `r_eval`, quest'ultimo caricherà il codice dell'oggetto e invocherà il suo metodo `gamma`. L'effetto ottenuto corrisponde a eseguire parte del codice del client sulla macchina server. Client diversi possono usare lo stesso servizio `r_eval` per eseguire codice diverso sulla macchina server.

## **16.6 Networking**

Il termine *networking* include sia il concetto di hardware di rete che quello di software di rete. Pertanto, per networking si intendono le tecnologie di rete e la progettazione di reti di calcolatori, ma anche aspetti software necessari per implementare la comunicazione tra processi. Le problematiche principali del networking sono riassunte in [Tabella 16.6](#). Tipo, topologia e tecnologia sono aspetti relativi alla progettazione di una rete. Tutte le altre problematiche riguardano lo scambio di messaggi tra processi - trovare l'indirizzo IP di un nodo in cui è collocato il processo destinazione, decidere la strada che deve seguire un messaggio per raggiungere un nodo e garantire che il messaggio sia spedito in modo efficiente e sicuro. Per determinare l'indirizzo IP di un host abbiamo già descritto il DNS nel Paragrafo 16.4.1. Tutti gli altri aspetti del networking saranno invece presentati in questo paragrafo.

| Problematiche                           | Descrizione                                                                                                                                                                                                                                                                               |
|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Tipo di rete                            | Il <i>tipo di rete</i> è determinato dalla distribuzione geografica degli utenti e delle risorse del sistema. I due tipi di rete principali sono <i>wide area network</i> (WAN) e <i>local area network</i> (LAN).                                                                        |
| Topologia di rete                       | La <i>topologia di rete</i> rappresenta l'organizzazione secondo cui i nodi sono fisicamente collegati tra loro tramite cavi di rete (detti <i>link</i> ). Essa influenza la velocità e l'affidabilità della comunicazione e il costo dell'hardware di rete.                              |
| Tecnologia di rete                      | La <i>tecnologia di rete</i> riguarda le strategie per la trasmissione dei dati nella rete. Essa influenza la larghezza di banda e la latenza della rete.                                                                                                                                 |
| Naming dei processi                     | Tramite il DNS, la coppia ( <i>&lt;host_name&gt;</i> , <i>&lt;process_id&gt;</i> ) che identifica il processo destinazione viene tradotta nella coppia (IP address, <i>&lt;process_id&gt;</i> ).                                                                                          |
| Strategia di connessione                | La <i>strategia di connessione</i> decide come instaurare cammini per lo scambio di dati tra processi comunicanti. Essa influenza il throughput e l'efficienza della comunicazione tra i processi.                                                                                        |
| Strategia di instradamento              | La <i>strategia di instradamento (routing)</i> decide la strada che deve seguire un pacchetto per raggiungere il nodo destinatario lungo il sistema. Essa influenza il ritardo con cui il messaggio viene trasmesso.                                                                      |
| Protocollo di rete                      | Il <i>protocollo di rete</i> è costituito da un insieme di regole e convenzioni che assicurano l'effettiva comunicazione sulla rete. Solitamente viene usata una gerarchia di protocolli per separare la gestione di vari aspetti relativi alla trasmissione e all'affidabilità dei dati. |
| Larghezza di banda e latenza della rete | La <i>larghezza di banda</i> di una rete è il tasso di trasferimento dei dati nella rete. La <i>latenza</i> è il tempo speso prima che i dati vengano spediti al sito destinatario.                                                                                                       |

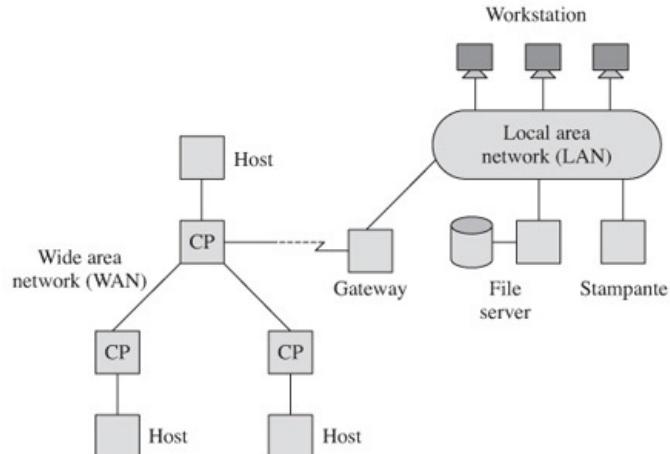
**Tabella 16.6** Problematiche del networking.

### 16.6.1 Tipi di rete

Una *wide area network* (WAN) collega risorse e utenti che sono geograficamente distanti. In passato le WAN sono state usate per permettere l'uso di costosi mainframe a un ampio numero di utenti di diverse organizzazioni residenti in posti lontani, oppure per permettere la comunicazione e lo scambio di dati tra vari utenti.

Successivamente, con l'avvento dei più economici personal computer, un numero sempre più ampio di organizzazioni iniziò a installare molti PC all'interno dei propri uffici. I dati usati dagli utenti di tali PC e le periferiche, quali per esempio stampanti laser di ottima qualità, divennero risorse critiche. Pertanto, furono installate *local area network* (LAN) per collegare le risorse collocate all'interno dello stesso ufficio o dello stesso edificio. Visto che tutte le risorse e gli utenti di una LAN appartenevano alla stessa organizzazione, non c'era motivo per condividere dati e risorse verso l'esterno. Quindi, poche LAN furono connesse alle WAN, sebbene la tecnologia per rendere questo possibile esistesse già. Lo scenario cambiò nuovamente con l'avvento di Internet, tant'è che al giorno d'oggi la maggior parte delle LAN e delle WAN sono connesse a Internet.

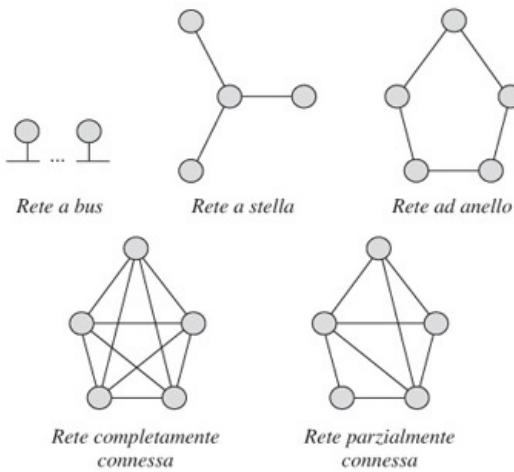
La [Figura 16.7](#) illustra i tipi di rete LAN e WAN. Nell'esempio, la LAN è costituita da personal computer, stampanti e da un file server. È connessa a una WAN tramite un *gateway*, ovvero un calcolatore che permette di collegare due o più reti tra di loro. Nella WAN vengono usati processori speciali chiamati *communication processor* (CP) per facilitare la comunicazione dei messaggi tra host distanti. Le LAN usano costosi cavi ad alta velocità quali i cavi di Categoria 5 o le fibre ottiche che garantiscono un elevato tasso di trasferimento dei dati. Per motivi di costo, le WAN usano, invece, linee pubbliche per trasferire dati e, pertanto, di solito non sono in grado di supportare tassi di trasferimento particolarmente elevati.



**Figura 16.7** Tipi di rete.

### 16.6.2 Topologia di rete

La [Figura 16.8](#) mostra cinque diverse topologie di rete. Esse si differenziano per costo dell'hardware di rete, velocità di comunicazione e affidabilità. La topologia basata su *bus* prevede che gli host siano organizzati in modo simile a quanto avviene tra bus e componenti interne di un PC. Tutti gli host sono connessi direttamente al bus, e pertanto il costo dell'hardware di rete è basso. Tuttavia, in ogni istante di tempo, solo una coppia di host può comunicare lungo il bus. Il bus garantisce generalmente tassi di trasferimento elevati, eccezion fatta in caso di contesa. La topologia a bus è usata nelle LAN basate su Ethernet.



**Figura 16.8** Topologie di rete.

Nella topologia a *stella*, ogni host è connesso solamente all'host centrale. Tale topologia è utile quando il sistema distribuito contiene un server, e i nodi eseguono processi che lo usano. L'affidabilità di una rete a stella dipende dall'affidabilità dell'host centrale. I ritardi di comunicazione tra un host e il server centrale, o tra due host, dipendono dalla congestione dell'host centrale. Le reti Fast Ethernet usano la topologia a stella. In una rete *ad anello*, ogni host ha due vicini. Quando un host desidera comunicare con un altro, il relativo messaggio viene propagato lungo l'anello finché non raggiunge l'host destinatario. Di conseguenza, il carico di comunicazione di un host è elevato anche quando nessuno dei suoi processi sta comunicando. In un anello unidirezionale, il link trasporta messaggi solo in una direzione, mentre in un anello bidirezionale, il link può portare messaggi in entrambe le direzioni. Naturalmente, gli

anelli unidirezionali e bidirezionali hanno caratteristiche di affidabilità diverse – una rete ad anello bidirezionale è immune da guasti che colpiscono un singolo host o un singolo link, mentre una rete ad anello unidirezionale non può esserlo.

In una rete *completamente connessa* esiste un link tra ogni coppia di host. Quindi, la comunicazione tra una coppia di host è immune da guasti che colpiscono gli altri host, e in una rete con  $n$  host, l'immunità si estende anche a guasti che colpiscono fino a  $(n - 2)$  link. Al contrario, uno o più host possono rimanere isolati solo se il numero di guasti supera  $(n - 2)$ . In tal caso, si parla di *partizionamento della rete*. Una rete *parzialmente connessa* contiene meno link di una rete completamente connessa. Pertanto, il costo è inferiore, ma la possibilità di avere un partizionamento della rete aumenta e può avvenire anche a fronte di guasti su un basso numero di host o link.

### 16.6.3 Tecnologie di rete

In questo paragrafo considereremo tre tecnologie di rete. Le tecnologie *Ethernet* e *token ring* sono usate in reti locali, mentre la tecnologia *Asynchronous Transfer Mode* (ATM) è usata nelle reti ISDN.

#### **Ethernet**

La rete Ethernet è una rete a bus semplice o multiplo (*branching bus*) basata su un circuito costituito da cavi collegati da ripetitori. Parecchie entità, chiamate *stazioni*, sono connesse allo stesso cavo. I dati sono trasmessi in unità chiamate *frame*. Ogni frame contiene gli indirizzi sorgente e destinazione, e un campo dati. Ogni stazione è costantemente in ascolto sul bus. Quando sul bus transita un frame indirizzato alla stazione, il frame viene copiato in un buffer, mentre i frame non indirizzati alla stazione vengono ignorati. La rete Ethernet originale opera a un tasso di trasmissione pari a 10 Mbit per secondo, ma esistono varianti che garantiscono tassi più elevati, come la Fast Ethernet che opera a 100 Mbit, la Gigabit Ethernet e la 10 Gigabit Ethernet.

Un *bridge* è usato per connettere tra loro LAN di tipo Ethernet. Si tratta di un calcolatore che riceve frame da una Ethernet e in base all'indirizzo destinazione, li trasferisce verso un'altra Ethernet a cui è connesso.

Dal momento che la topologia base di una Ethernet è a bus, in ogni istante di tempo può essere attiva solo una comunicazione. Per gestire tale comunicazione si usa una tecnologia nota come “carrier sense multiple access with collision detection” (CSMA/CD). In base a essa, una stazione, che desidera inviare un messaggio, si mette in ascolto del traffico esistente sulla rete per verificare se è in corso la trasmissione di un segnale. Tale verifica si chiama *carrier sensing*. Se non viene rilevato alcun segnale, la stazione inizia a trasmettere il suo frame. Tuttavia, altre stazioni potrebbero aver rilevato l'assenza di segnale sul cavo di rete e quindi potrebbero iniziare a loro volta una trasmissione. In questo caso, i loro frame interferirebbero l'un l'altro causando un voltaggio anormale sul cavo. Una simile situazione è nota come *collisione*. Una stazione che rileva una collisione emette un segnale speciale a 32 bit detto *jam*. Di conseguenza, tutte le stazioni che ricevono il segnale *jam* diventano consapevoli che è in atto una collisione. In seguito, le stazioni che stavano trasmettendo interrompono la comunicazione e si mettono in attesa per un intervallo di tempo di durata casuale prima di ricominciare la trasmissione dei loro frame. Ovviamente, non vi è alcuna garanzia che alla ripresa della trasmissione la collisione non si ripresenti; tuttavia, il meccanismo descritto aiuta a far in modo che prima o poi tutti i frame vengano trasmessi e ricevuti senza collisioni. La dimensione dei frame deve essere sufficientemente grande per facilitare l'identificazione di eventuali collisioni. Per una rete Ethernet a 10 Mbps o 100 Mbps (Mbps è una abbreviazione per  $2^{20}$  bit per secondo) è sufficiente una dimensione pari a 512 bit, mentre servono 4096 bit per una Ethernet di tipo Gigabit.

#### **Token ring**

Una rete token ring si basa su una topologia ad anello e usa la nozione di *token* per decidere, in ogni istante, quale stazione possa trasmettere. Il token è un messaggio speciale che circola sulla rete caratterizzato da un bit di stato, che può essere *libero* o *occupato*. Se il bit di stato è *occupato*, significa che è in corso la trasmissione di un messaggio nella rete, mentre se è *libero* significa che la rete è inutilizzata. Una stazione che desidera trasmettere un messaggio rimane in attesa fino a quando riceve un token con il bit di stato *libero*. Dopodiché cambia lo stato del bit a *occupato* e inizia a trasmettere il messaggio. Pertanto, visto che ciascun messaggio segue sempre un token

con il bit di stato impostato a *occupato*, è possibile trasmettere solo un messaggio alla volta. La dimensione del messaggio non è predefinita e non è necessario spezzarlo in frame di dimensione standard.

Quando una stazione vede transitare un messaggio nella rete, verifica se è destinato ad essa; solo la stazione destinataria esegue una copia del messaggio. Quando la stazione che ha originariamente spedito il messaggio vede transitare nella rete il token con bit di stato *occupato*, riporta il bit di stato a *libero* in modo da rendere la rete disponibile per la trasmissione di altri messaggi. Nelle reti che supportano il rilascio anticipato del token, il bit di stato di quest'ultimo viene impostato a *libero* dalla stazione destinataria del messaggio. Chiaramente, l'attività su una rete token ring si blocca se il token viene perso a causa di errori nella comunicazione. Per ripristinare il normale funzionamento, è necessario che una delle stazioni sia costantemente in ascolto del traffico di rete per verificare la presenza del token, e creare uno nuovo nel caso in cui vada perso.

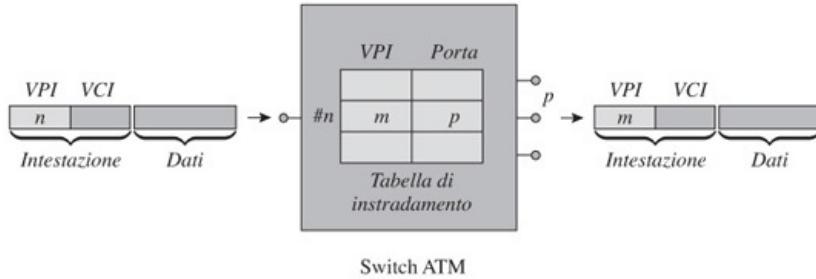
### **Tecnologia Asynchronous Transfer Mode (ATM)**

L'ATM è una tecnologia di rete a circuito virtuale (*virtual-circuit*) basata su commutazione di pacchetto (*packet-switching*) (Paragrafi 16.6.4 e 16.6.5). Nella terminologia ATM, il circuito virtuale e il pacchetto sono chiamati, rispettivamente, *virtual path* (cammino virtuale) e cella. L'ATM implementa un cammino virtuale tra due siti riservando una parte specifica della banda dei link fisici situati nel percorso di rete che collega i siti stessi. Nel caso in cui uno stesso link sia comune a più cammini virtuali, esso distribuisce il traffico dei vari cammini su base statistica in modo che ognuno riceva una ben precisa porzione della banda del link. In tal modo, le celle che vengono trasmesse in un cammino virtuale non subiscono ritardi dovuti al traffico relativo ad altri cammini virtuali.

Il principio di riservare parte della banda a una specifica comunicazione è ulteriormente esteso dagli stessi host di una rete ATM. Immaginiamo per esempio di instaurare un cammino virtuale tra due host X e Y. Quando un processo  $P_i$  in X desidera comunicare con un processo  $P_j$  in Y, i due host possono instaurare un *canale virtuale* tra  $P_i$  e  $P_j$  riservando parte della banda assegnata a un cammino virtuale esistente tra X e Y. Questo doppio meccanismo garantisce che il traffico dei messaggi in transito tra una coppia di processi non subisca ritardi dovuti al traffico generato da altre coppie di processi.

La tecnologia ATM mira a fornire un ambiente real-time adatto ad applicazioni multimediali per garantire un efficiente trasferimento di diverse tipologie di traffico come voce, video e dati ad alta velocità. La dimensione di una cella ATM è pari a 53 byte ed è il risultato di un compromesso tra celle piccole, adatte alla comunicazione di traffico voce che mal tollera ritardi troppo lunghi, e celle grandi, adatte invece al trasferimento di dati per i quali è conveniente ridurre al minimo l'aggravio dovuto alla suddivisione di un messaggio in pacchetti e alla sua successiva ricomposizione. Ogni cella contiene un'intestazione di 5 byte e un campo dati di 48 byte. L'intestazione contiene due informazioni: *l'identificatore del cammino virtuale* (VPI) e *l'identificatore del canale virtuale* (VCI).

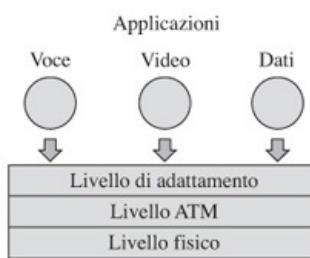
La [Figura 16.9](#) riporta in modo schematico il funzionamento di un commutatore (*switch*) ATM. Esso contiene una tabella di instradamento costituita da un record di dati per ogni cammino virtuale definito nello switch. Il record contiene due campi: il campo VPI e il campo *porta*. In [Figura 16.9](#), l'identificatore del cammino virtuale della cella entrante è  $n$ , e il record  $n$ -esimo della tabella di instradamento contiene i valori  $m$  e  $p$ . Lo switch copia  $m$  nel campo VPI della cella e invia la cella modificata sulla porta  $p$ . Questo semplice meccanismo garantisce che gli identificatori assegnati ai cammini virtuali non debbano essere unici nel sistema; è sufficiente che lo siano all'interno dello switch. Le operazioni compiute dallo switch sono effettuate da componenti hardware per garantire una commutazione estremamente veloce, nell'ordine di pochi microsecondi. Ciò permette di avere velocità di trasmissione su reti WAN simili a quelle di una LAN.



**Figura 16.9** Uno switch ATM.

Le applicazioni specificano la banda desiderata durante la creazione del corrispondente cammino virtuale. Quindi, il sistema operativo instaura un cammino virtuale riservando la banda richiesta su ogni singolo link, sceglie un identificatore di cammino virtuale unico in ogni switch, e aggiorna la tabella di instradamento. Durante la gestione del traffico nei vari canali virtuali di uno stesso cammino, gli host usano informazioni statistiche per distribuire la banda a ogni applicazione in modo appropriato. In tal modo, applicazioni diverse possono trasmettere messaggi simultaneamente a diverse velocità sullo stesso cammino virtuale.

Una rete ATM è basata su un'architettura a maglia costituita da tante stelle. Gli switch ATM sono connessi tra loro in modo da costruire una maglia (*mesh*), mentre gli host sono connessi agli switch in modo da formare una rete a stella. In tal modo è possibile avere un cammino tra ogni coppia di nodi. La [Figura 16.10](#) mostra i livelli del protocollo ATM. Il livello fisico effettua il trasferimento di celle nella rete. Il livello ATM si occupa della trasmissione di messaggi tra nodi ATM eseguendo operazioni di multiplexing e demultiplexing dei canali virtuali all'interno dei cammini virtuali, schedulazione e instradamento delle celle. Il livello di adattamento fornisce differenti tipi di servizi per diverse tipologie di traffico quali comunicazioni voce, video e dati. Per ogni tipologia di traffico, il livello di adattamento fornisce specifici protocolli.



**Figura 16.10** Protocollo ATM.

#### 16.6.4 Strategie di connessione

Una *connessione* è un cammino tra due processi comunicanti. Una strategia di connessione, chiamata anche *tecnica di commutazione*, determina *quando* dovrebbe essere instaurata una connessione tra una coppia di processi, e *per quanto tempo* dovrebbe essere mantenuta. La scelta della tecnica di commutazione influenza l'efficienza della comunicazione tra una coppia di processi e il throughput dei link usati nella comunicazione. La [Figura 16.11](#) mostra tre strategie di connessione. Il simbolo  $m_i$  denota un messaggio, mentre  $pc_j(m_i)$  identifica il *pacchetto*  $j$ -esimo del messaggio  $m_i$ . Il concetto di pacchetto ha lo stesso significato definito successivamente in questo paragrafo.

##### Commutazione di circuito

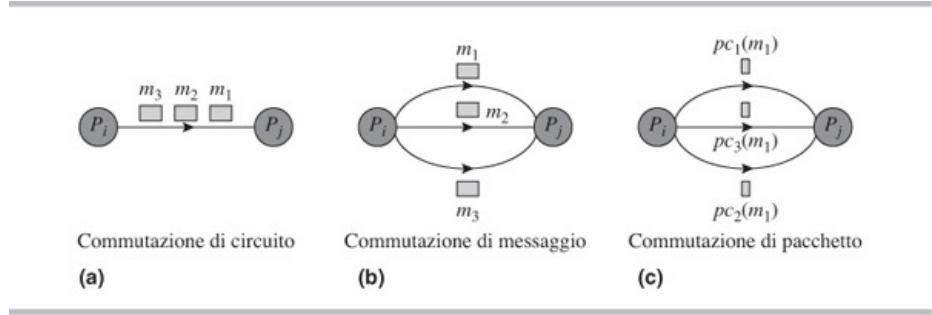
Un *circuito* è una connessione usata in modo esclusivo da una coppia di processi per scambiare messaggi [[Figura 16.11\(a\)](#)]. Il circuito viene instaurato quando i processi decidono di comunicare, ovvero, prima che il primo messaggio venga spedito, e viene

distrutto dopo che l'ultimo messaggio è stato ricevuto. L'installazione di un circuito coinvolge varie azioni quali, decidere il cammino che dovrà essere percorso dai pacchetti attraverso la rete, e riservare le risorse necessarie per la comunicazione. Ogni circuito è identificato da un codice unico, che deve essere esplicitamente usato dai processi per spedire e ricevere messaggi.

Il vantaggio della commutazione di circuito consiste nel fatto che dopo l'installazione del circuito i messaggi non subiscono ulteriori ritardi. Tuttavia, la preparazione del circuito e la necessità di riservare opportune risorse di comunicazione induce un certo ritardo iniziale nella comunicazione. Pertanto, l'uso della commutazione di circuito è giustificato solamente se la densità dei messaggi trasferiti nel sistema è generalmente bassa, mentre il traffico tra la coppia di processi è medio-alto.

### Commutazione di messaggio

Con la commutazione di messaggio viene stabilita una connessione per ogni messaggio scambiato tra due processi. In tal modo messaggi diversi scambiati dalla stessa coppia di processi possono seguire cammini diversi nel sistema [Figura 16.11(b)]. La commutazione di messaggio richiede un continuo aggravio per la gestione dei messaggi e può causare ritardi dovuti al tempo richiesto per instaurare la connessione, pertanto, il suo uso è giustificato solo quando il traffico tra due processi è leggero. Tuttavia, questo tipo di commutazione non riserva risorse di comunicazione per una coppia di processi durante tutta la durata della comunicazione, quindi, altri processi possono usare contemporaneamente la stessa connessione, o alcuni dei suoi link. Chiaramente, il traffico di rete deve essere sufficientemente alto per trarre beneficio da una tale condivisione delle risorse.



**Figura 16.11** Strategie di connessione: commutazione di circuito, di messaggio e di pacchetto.

### Commutazione di pacchetto

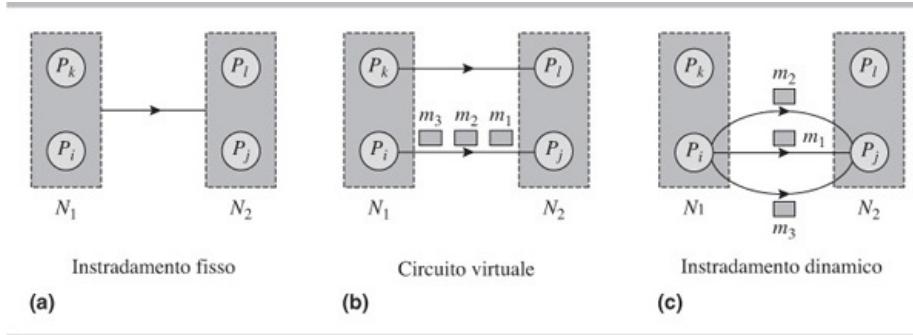
Nella commutazione di pacchetto, un messaggio viene diviso in parti di dimensione standard, chiamate *pacchetti* per ognuno dei quali viene instaurata una connessione individuale. Pertanto, i pacchetti di uno stesso messaggio possono essere trasmessi lungo cammini diversi [Figura 16.11(c)] e possono essere ricevuti dal sito destinazione in ordine diverso da quello di spedizione. L'utilizzo della commutazione di pacchetto induce due tipi di costi aggiuntivi: il pacchetto deve includere nell'intestazione informazioni per la sua identificazione - l'identificatore del messaggio a cui appartiene, il numero di sequenza interno al messaggio, l'identificatore dei processi mittente e destinatario - e i pacchetti devono essere riassemblati per ricostruire il messaggio originale una volta giunti a destinazione. D'altra parte, l'uso di pacchetti di dimensione fissa riduce i costi di ritrasmissione in caso di errore. Inoltre, i link non sono monopolizzati da coppie specifiche di processi, e quindi tutte le coppie di processi comunicanti ricevono un servizio equo. Le precedenti caratteristiche rendono la commutazione di pacchetto particolarmente conveniente nel caso di processi interattivi.

A causa dei costi necessari per instaurare una connessione, è comune l'uso di protocolli privi di connessione (*connectionless*) per inviare messaggi e pacchetti. In tali protocolli, il nodo che dà origine alla comunicazione sceglie semplicemente uno dei suoi vicini e gli invia il messaggio. Se il vicino in questione non è il nodo destinatario, esso salva il messaggio nella sua memoria e sceglie uno dei suoi vicini per inoltrare il messaggio a sua volta. Tale processo si ripete finché il messaggio non giunge a destinazione. Questo modo di trasmettere i messaggi è chiamato *store-and-forward*, ovvero memorizza e inoltra. Allo stesso modo possono essere trasferiti anche i pacchetti

al posto dei messaggi. Una trasmissione priva di connessione si adatta meglio alla densità di traffico dei vari link rispetto alla commutazione di messaggio o di pacchetto, poiché ogni nodo può scegliere il link su cui inviare il messaggio o il pacchetto. La sua implementazione viene realizzata tipicamente scambiando informazioni sul traffico di rete tra i vari nodi e mantenendo una tabella in ogni nodo per indicare a quale vicino inviare un messaggio o un pacchetto per raggiungere un determinato nodo destinatario. Tuttavia, ogni nodo deve avere una memoria sufficientemente grande per bufferizzare messaggi e pacchetti pendenti quando i suoi link in uscita sono congestionati.

### 16.6.5 Instradamento

La funzione di instradamento viene invocata ogni volta che viene instaurata una connessione. Essa decide quale cammino nella rete dovrebbe essere usato dalla connessione. La scelta fatta dalla strategia di instradamento influenza l'abilità di poter adattare la comunicazione ai vari cambiamenti relativi alla tipologia di traffico esistente nel sistema. La [Figura 16.12](#) mostra tre strategie di instradamento.



**Figura 16.12** Strategie di instradamento: instradamento fisso, circuito virtuale e instradamento dinamico.

#### Instradamento fisso

Con l'instradamento fisso, la comunicazione tra due nodi viene effettuata sempre tramite un cammino predefinito [[Figura 16.12\(a\)](#)]. Quando i processi residenti in tali nodi desiderano comunicare, viene stabilita una connessione su tale cammino. L'instradamento fisso è semplice ed efficiente da implementare – ogni nodo contiene solamente una tabella per memorizzare i cammini verso gli altri nodi del sistema; tuttavia, è poco flessibile e mal si adatta a eventuali guasti di nodi e link e ai cambiamenti che possono verificarsi in merito alla densità del traffico. L'uso dell'instradamento fisso può quindi comportare ritardi e basso throughput.

#### Circuito virtuale

Con il circuito virtuale, il cammino da usare per trasferire messaggi o pacchetti tra due nodi viene definito all'inizio di ogni sessione di comunicazione. Dopodiché, lo stesso cammino viene usato per tutti i messaggi inviati durante la stessa sessione [[Figura 16.12\(b\)](#)]. Per decidere il migliore cammino da usare durante una nuova sessione vengono usate informazioni relative alla densità di traffico e ai ritardi di comunicazione lungo i vari link del sistema. Questa strategia può pertanto adattarsi ai cambiamenti del traffico o a eventuali guasti su nodi o link del sistema, e garantisce throughput e tempi di risposta buoni.

#### Instradamento dinamico

Con l'instradamento dinamico, viene potenzialmente selezionato un cammino diverso per ogni messaggio o pacchetto da spedire. Quindi, messaggi differenti trasferiti tra una stessa coppia di processi e pacchetti differenti dello stesso messaggio possono usare cammini diversi [[Figura 16.12\(c\)](#)]. Questa caratteristica permette alla strategia di instradamento di rispondere in modo più efficace ai cambiamenti della tipologia di traffico e a eventuali guasti su nodi o link del sistema, e permette di ottenere throughput e tempi di risposta migliori rispetto a quanto garantito da un circuito virtuale. Nei nodi della rete Arpanet, progenitrice di Internet, venivano continuamente scambiate informazioni sulla densità del traffico di rete e sui ritardi di comunicazione lungo i vari

link. Tali informazioni erano usate per determinare in ogni istante il miglior cammino verso la destinazione desiderata.

### 16.6.6 Protocolli di rete

Un *protocollo di rete* è un insieme di regole e convenzioni usate per implementare meccanismi di comunicazione attraverso la rete. Gli aspetti da tenere in considerazione sono molteplici, quali la necessità di assicurare un adeguato livello di riservatezza dei dati trasmessi, garantire una comunicazione efficiente, gestire errori di trasmissione, ecc. In pratica, per gestire in modo modulare i vari aspetti coinvolti nella comunicazione di rete si usa una *gerarchia di protocolli*. All'interno di essa, ogni protocollo gestisce un numero limitato di funzionalità e fornisce un'interfaccia verso i protocolli di livello superiore e inferiore. I livelli della gerarchia agiscono come i livelli di astrazione di un sistema operativo (Paragrafo 1.1) e, pertanto, forniscono gli stessi benefici – una qualunque entità che usi le funzionalità fornite da un protocollo al livello *i*-esimo della gerarchia non ha bisogno di conoscere i dettagli con cui vengono implementati i protocolli di livello inferiore. Di conseguenza, i protocolli di livello più basso gestiscono gli aspetti relativi alla trasmissione dei dati come, per esempio, la rilevazione di errori nella trasmissione; i protocolli di livello intermedio si occupano della gestione dei pacchetti e del relativo instradamento; infine, i protocolli di livello più alto lavorano sugli aspetti semantici relativi alle applicazioni come, per esempio, l'atomicità delle azioni o la riservatezza dei dati.

#### **Il protocollo ISO**

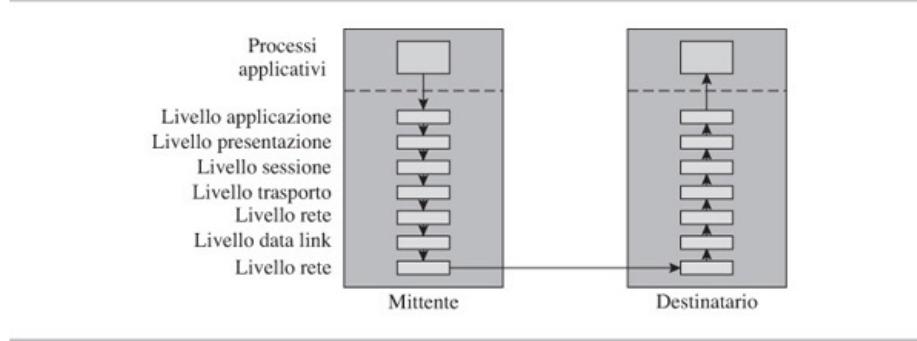
L'organizzazione internazionale per la standardizzazione (ISO) ha sviluppato un modello di riferimento, noto come *modello OSI (Open Systems Interconnection reference model*, o *protocollo ISO* o *stack di protocolli ISO*), per implementare la comunicazione all'interno di un sistema aperto. Il modello è costituito dai sette livelli di protocollo riassunti in [Tabella 16.7](#).

| <b>Livello</b>                  | <b>Funzionalità</b>                                                                                                                                             |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>1. Livello fisico</b>        | Fornisce meccanismi elettrici per la trasmissione dei bit su un link fisico.                                                                                    |
| <b>2. Livello data link</b>     | Organizza i bit ricevuti in frame. Gestisce la rilevazione di errori nei frame. Gestisce il flusso di controllo.                                                |
| <b>3. Livello rete</b>          | Effettua la commutazione e l'instradamento.                                                                                                                     |
| <b>4. Livello trasporto</b>     | Crea i pacchetti da spedire. Assembla i pacchetti in arrivo. Implementa la gestione degli errori e gestisce la ritrasmissione. Gestisce il flusso di controllo. |
| <b>5. Livello sessione</b>      | Instaura e termina le sessioni. Gestisce le operazioni di ripristino delle applicazioni in seguito a errori.                                                    |
| <b>6. Livello presentazione</b> | Gestisce la semantica dei dati tramite il cambio di rappresentazione, la compressione e la cifratura/decifratura dei dati quando necessario.                    |
| <b>7. Livello applicazione</b>  | Fornisce l'interfaccia di rete per le applicazioni.                                                                                                             |

**Tabella 16.7** Livelli dello stack dei protocolli ISO.

La [Figura 16.13](#) mostra lo schema operativo del modello OSI durante lo scambio di un messaggio tra due processi. L'applicazione mittente genera il messaggio e lo invia al livello di applicazione. Quest'ultimo aggiunge al messaggio un'intestazione contenente alcune informazioni di controllo. Quindi, il messaggio passa attraverso i livelli di presentazione e di sessione che aggiungono ulteriori intestazioni. Il livello di presentazione cambia la rappresentazione dei dati ed effettua la cifratura/decifratura del messaggio, mentre il livello di sessione stabilisce una connessione tra i processi mittente e ricevente. Successivamente, il livello di trasporto spezza il messaggio in pacchetti e li inoltra al livello di rete, il quale determina il link su cui dovrà essere spedito ogni pacchetto e inoltra l'identificatore del link e il relativo pacchetto al livello di data link. Quest'ultimo vede i pacchetti come una semplice stringa di bit a cui aggiunge informazioni per la rilevazione e la correzione di errori di trasmissione, quindi inoltra i

pacchetti al livello fisico affinché venga effettuata la trasmissione attraverso la rete. Alla ricezione del messaggio, invece, il livello di data link esegue la fase di rilevazione e correzione degli errori e ricostruisce i frame a partire dai bit ricevuti dal livello fisico, il livello di trasporto ricostruisce i messaggi, e il livello di presentazione riorganizza i dati contenuti nei messaggi nella rappresentazione richiesta dall'applicazione ricevente. Di seguito verranno discussi i dettagli relativi ai vari livelli del protocollo.



**Figura 16.13** Schema operativo dello stack dei protocolli ISO.

Il *livello fisico* è responsabile degli aspetti meccanici, elettrici, funzionali e procedurali relativi alla trasmissione fisica di sequenze di bit attraverso la rete. Esso è implementato nell'hardware del dispositivo di rete. Gli standard più comuni per il livello fisico sono RS-232C ed EIA-232D. Il *livello data link* fornisce meccanismi per la identificazione e la correzione degli errori di trasmissione e per la gestione del flusso di controllo. Esso spezza la sequenza di bit da inviare in blocchi di dimensione fissa detti *frame*, e aggiunge un CRC a ogni frame (Paragrafo 14.3). Inoltre, il livello data link gestisce il flusso di controllo spedendo i frame a un tasso compatibile con la capacità di ricezione del destinatario. Un protocollo comunemente usato a livello di data link è l'*HDLC (high-level data link control)*. *Bridge* e *switch* operano a questo livello.

Il *livello rete* è responsabile dell'instradamento dei dati tra due siti e della gestione della connessione, che tuttavia può essere opzionale. I protocolli più comunemente usati a questo livello sono X.25, un protocollo orientato alla connessione che usa circuiti virtuali, e l'*Internet Protocol (IP)* che, al contrario, non è orientato alla connessione. I router operano a questo livello. Nelle LAN e nei sistemi con connessione punto a punto, il livello di rete è perlopiù ridondante.

Il *livello trasporto* garantisce che la trasmissione dei messaggi avvenga senza errori. Esso spezza i messaggi in pacchetti e inoltra questi ultimi al livello di rete sottostante. Quindi, gestisce errori di comunicazione come, per esempio, la mancata ricezione di un pacchetto dovuto a guasti sui nodi o sui link del sistema. Tale caratteristica è fornita in modo simile a quanto avviene nei meccanismi adottati per garantire l'affidabilità nei protocolli IPC. Pertanto, il livello di trasporto usa strategie basate su timeout e ritrasmissione dei pacchetti (Paragrafo 16.4). Inoltre, il livello di trasporto esegue un controllo del flusso in modo che i dati siano trasferiti a un tasso gestibile dal ricevente. Il tasso di trasmissione effettivo dipende dalla dimensione del buffer di ricezione del nodo destinatario e dal tasso al quale quest'ultimo è in grado di estrarre i dati dal buffer. Il modello ISO prevede cinque classi di protocolli a livello di trasporto, chiamati TP0, TP1, ..., TP4. Altri protocolli di uso comune sono il *Transport Control Protocol (TCP)*, un protocollo orientato alla connessione che implementa meccanismi di affidabilità per la trasmissione dei messaggi, e lo *User Datagram Protocol (UDP)*, che invece è un protocollo non orientato alla connessione che non fornisce alcuna garanzia sull'effettivo esito della trasmissione di un messaggio.

Il *livello sessione* fornisce meccanismi per controllare il dialogo tra due entità che usano un protocollo orientato alla connessione. In particolare, esso fornisce: meccanismi per gestire l'autenticazione, modi diversi per instaurare un dialogo tra due entità (unidirezionale, bidirezionale alternato, bidirezionale simultaneo) e funzionalità per la gestione di checkpoint e strategie di ripristino in seguito a errori. Inoltre, il livello di sessione si fa carico di garantire che i messaggi scambiati tramite primitive *send* non bloccanti arrivino nell'ordine corretto (Paragrafo 16.4). Infine, fornisce un servizio di quarantena per mezzo del quale i messaggi possono essere bufferizzati nel sito del ricevente fino a quando non vengono esplicitamente rilasciati dal mittente. Tale servizio

è utile per eseguire azioni atomiche su file (Paragrafo 13.11.2) o per implementare paradigmi di esecuzione basati su transazioni atomiche (Paragrafo 19.4).

Il *livello presentazione* fornisce principalmente servizi utili a cambiare il modo in cui vengono rappresentati i dati nel sito del mittente per renderli compatibili con lo stile di rappresentazione adottato nel sito del ricevente. Inoltre, questo livello si occupa di garantire la riservatezza dei dati tramite meccanismi di cifratura, e di ridurre il volume dei dati tramite strategie di compressione.

Il *livello applicazione* supporta servizi specifici quali trasferimento di file, e-mail, accessi a macchine remote. Tra i vari protocolli a livello di applicazione vanno ricordati il *File Transfer Protocol* (FTP) per il trasferimento di file, X.400 per la gestione delle e-mail, ed *rlogin* per l'accesso remoto.

### TCP/IP

Il *Transmission Control Protocol/Internet Protocol* (TCP/IP) è un protocollo molto diffuso che viene usato per la comunicazione sulla rete Internet. Il modello TPC/IP ha meno livelli del modello OSI, pertanto, risulta essere più efficiente ma anche più complesso da implementare. La [Figura 16.14](#) mostra i vari livelli TPC/IP. Il livello più basso è costituito da un protocollo data link. Sopra questo si trova l'*Internet Protocol* (IP) assimilabile al protocollo di rete del modello OSI. Il protocollo IP può agire sopra qualunque tipo di protocollo data link ed effettua la trasmissione dei dati usando indirizzi IP a 32 o 128 bit, rispettivamente, nella versione IPv4 e nella più recente versione IPv6, per identificare i nodi destinazione. IP è un protocollo non orientato alla connessione, senza meccanismi atti a garantire l'affidabilità della trasmissione. Esso, infatti, non garantisce né che i pacchetti di un messaggio arrivino a destinazione privi di errori, né che ogni pacchetto arrivi in un'unica copia, né che i pacchetti vengano recapitati nello stesso ordine di invio. Tali caratteristiche di affidabilità sono invece fornite da protocolli presenti nei livelli superiori della gerarchia TCP/IP.

I protocolli dei livelli superiori forniscono meccanismi per la comunicazione tra processi - ogni host assegna un numero di porta a 16 bit unico a ogni suo processo, e un processo mittente usa un indirizzo per il processo destinatario composto dalla coppia (indirizzo IP, numero di porta). L'utilizzo dei numeri di porta permette ai vari processi in esecuzione su un host di inviare e ricevere messaggi contemporaneamente. Alcuni servizi molto noti, come FTP, telnet, SMTP e HTTP, sono identificati da numeri di porta standard assegnati direttamente da un'apposita organizzazione: *Internet Assigned Numbers Authority* (IANA). I numeri di porta per altre tipologie di servizi sono invece assegnati dal sistema operativo in ogni singolo host.

Come mostrato in [Figura 16.14](#), nel livello superiore al protocollo IP, corrispondente al livello di trasporto nel modello OSI, possono essere usati due diversi protocolli: il TPC o l'UDP. Il *Transmission Control Protocol* (TCP) è un protocollo affidabile orientato alla connessione. Esso instaura un circuito virtuale tra due processi e garantisce una comunicazione affidabile tramite la ritrasmissione dei messaggi che non vengono ricevuti entro un intervallo di tempo predefinito (Paragrafo 16.4 in merito ai meccanismi di acknowledgment e timeout usati per garantire la spedizione affidabile dei messaggi). Il costo aggiuntivo necessario per garantire l'affidabilità risulta essere alto se le velocità di trasmissione e ricezione di mittente e ricevente non corrispondono, oppure quando la rete è particolarmente congestionata. Pertanto, il TPC implementa meccanismi per il *controllo del flusso* per assicurare che il mittente non invii pacchetti più velocemente di quanto il ricevente sia in grado di accettare, e per il *controllo della congestione* per assicurare che il traffico sia sufficientemente regolato in modo da non sovraccaricare la rete.

|                 |                                                                                                 |                              |
|-----------------|-------------------------------------------------------------------------------------------------|------------------------------|
| Livelli ISO 5-7 | File transfer protocol (FTP), e-mail, accesso remoto, o protocollo specifico per l'applicazione |                              |
| Livello ISO 4   | Transmission Control Protocol (TCP)                                                             | User Datagram Protocol (UDP) |
| Livello ISO 3   | Internet Protocol (IP)                                                                          |                              |
| Livello ISO 2   | Protocollo di data link                                                                         |                              |

**Figura 16.14** La gerarchia di protocolli TCP/IP.

L'*User Datagram Protocol* (UDP) è invece un protocollo non affidabile, non orientato alla connessione, che non garantisce né che un pacchetto spedito sia effettivamente ricevuto, né che i vari pacchetti di un messaggio arrivino nell'ordine corretto.

Il costo richiesto da UDP è sicuramente inferiore rispetto a quello necessario per TPC, in quanto UDP non deve né instaurare né gestire un circuito virtuale, e non deve neppure implementare alcun meccanismo di affidabilità. Il protocollo UDP è solitamente impiegato in applicazioni multimediali, come le videoconferenze, dove la perdita occasionale di qualche pacchetto non comporta particolari problemi - al limite causa una scarsa qualità dell'immagine. Le applicazioni multimediali usano meccanismi interni per il controllo del flusso e della congestione come, per esempio, algoritmi per la riduzione della risoluzione delle immagini - che, di conseguenza, abbassano la loro qualità - quando il mittente, il ricevente o la rete risultano essere troppo carichi.

Il livello più alto della pila TCP/IP, corrispondente ai livelli 5-7 del modello OSI, è occupato da protocolli di livello di applicazione come, per esempio, FTP per il trasferimento di file, SMTP per la gestione delle e-mail, o i protocolli per gestire l'accesso remoto. Qualora al livello immediatamente sottostante venga usato UDP, il livello più alto della pila TPC/IP può anche essere occupato da un protocollo specifico implementato direttamente nell'applicazione.

### 16.6.7 Larghezza e latenza della banda di rete

Quando due nodi desiderano scambiare dati tra loro, l'hardware e i protocolli di rete partecipano al trasferimento dei dati sul link fisico, mentre le macchine predisposte a fornire servizi di comunicazione sulla rete, ovvero i *communication processor*, memorizzano e inoltrano i dati finché questi non raggiungono il nodo destinatario. In tale ambito, due aspetti fondamentali per misurare le prestazioni della rete sono il tasso a cui i dati possono essere spediti e la velocità con cui i dati raggiungono la loro destinazione.

Con il termine *network bandwidth* (letteralmente, larghezza della banda di rete) si indica il tasso a cui i dati vengono trasferiti nella rete. La banda è influenzata da vari fattori, come la capacità dei link o il tasso di errore e i ritardi imposti da router, bridge e gateway. Il termine *picco di banda* si usa per indicare il tasso di trasferimento massimo teorico tra due nodi. Tuttavia, la banda effettiva può essere più bassa del picco a causa di errori nella trasmissione, che inducono all'uso di timeout e ritrasmissione. Con il termine *latenza* si indica, invece, il tempo trascorso tra l'invio di un byte di dati dal nodo sorgente e la sua ricezione nel nodo destinatario. La latenza è tipicamente calcolata per il primo byte di dati trasmesso. Essa dipende dal tempo richiesto per processare i dati nei vari livelli della pila dei protocolli di rete e dalla congestione della rete stessa.

## 16.7 Modello di un sistema distribuito

Per valutare le proprietà di un sistema distribuito, quali per esempio l'impatto di un guasto nella sua funzionalità, o la latenza e il costo richiesto per la trasmissione di un messaggio, si usa solitamente un modello del sistema. Un sistema distribuito è tipicamente modellato come un grafo:

$$S = (N, E)$$

dove  $N$  ed  $E$  indicano, rispettivamente, l'insieme dei nodi e l'insieme degli archi. Ogni nodo può rappresentare un host, ovvero un sistema di calcolo, e ogni arco può rappresentare un link che connette due nodi; tuttavia, come discusso successivamente, i nodi e gli archi possono anche avere altre connotazioni. Il *grado* di un nodo è pari al numero di archi a esso connessi. Ogni nodo ha una *lista di importazione* che descrive risorse e servizi non locali che il nodo può usare, e una *lista di esportazione* che, viceversa, descrive risorse locali che sono accessibili ad altri nodi. Per semplicità, non includeremo il *name server* (Paragrafo 16.4.1) nel modello del sistema.

In pratica, si usano due grafi diversi per modellare un sistema distribuito. Il *modello fisico* viene usato per rappresentare l'organizzazione delle entità fisiche del sistema. In tale modello, i nodi e gli archi presentano le implicazioni descritte precedentemente, ovvero, un nodo è un sistema di calcolo mentre un arco è un link di comunicazione. Nel *modello logico*, invece, i nodi rappresentano entità logiche come, per esempio, i processi, mentre gli archi rappresentano le relazioni tra le entità. Il modello logico può usare archi indiretti o diretti. Un arco indiretto rappresenta una relazione simmetrica, come una

comunicazione bidirezionale tra due processi. Un arco diretto, invece, rappresenta una relazione asimmetrica, come la relazione padrefiglio tra due processi o una comunicazione unidirezionale. Va evidenziato che nodi e archi di un modello logico non hanno una corrispondenza uno a uno con le entità fisiche presenti in un sistema distribuito.

La **Tabella 16.8** mostra le proprietà che possono essere determinate analizzando il modello del sistema. Una proprietà molto importante è la *resiliency* del sistema, che consiste nella sua capacità di tollerare guasti senza subire partizionamenti. Un sistema  $k$ -*resilient* può tollerare una qualunque combinazione di guasti fino a un massimo di  $k$  guasti contemporanei. Se  $n'$  è il più piccolo grado tra i nodi del sistema, sono necessari almeno  $n'$  guasti affinché un nodo risulti isolato. Tuttavia, per partizionare il sistema è sufficiente un numero minore di guasti (Problema 16.7). L'Esempio 16.1 mostra come l'analisi del modello di un sistema possa essere usata anche per progettare la rete.

| Proprietà                                                    | Descrizione                                                                                                                                                                                                                                                                                        |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Impatto di un guasto                                         | I guasti possono isolare un nodo dal resto del sistema o addirittura <i>partizionare</i> il sistema, cioè spezzarlo in due o più parti tali che due nodi appartenenti a partizioni diverse non possano comunicare tra loro.                                                                        |
| Resilienza                                                   | Un sistema è considerato $k$ - <i>resilient</i> , dove $k$ è una costante, se $k$ è il numero massimo di guasti che il sistema può tollerare senza subire partizionamenti.                                                                                                                         |
| Latenza tra due nodi                                         | La latenza minima di un cammino di comunicazione dipende dalla latenza minima di ogni link incluso nel cammino. La latenza minima tra due nodi è la più piccola tra le latenze minime di tutti i cammini che collegano i due nodi.                                                                 |
| Costo per inviare un'informazione a tutti i nodi del sistema | Il costo di tale operazione dipende dalla topologia del sistema e dall'algoritmo usato per inviare l'informazione. In un sistema completamente connesso composto da $n$ nodi, il costo può essere misurato in $n - 1$ messaggi. Il costo può aumentare se il sistema non è completamente connesso. |

**Tabella 16.8** Proprietà determinabili analizzando il modello del sistema.

### Esempio 16.1 - Resilienza di un sistema

Supponendo che solo uno o due siti in un sistema possano guastarsi contemporaneamente, e che non vi siano mai guasti sui link di comunicazione, la disponibilità di una risorsa è garantita su tutto il sistema se esistono tre istanze della risorsa stessa in tre siti differenti. Supponendo, invece, che anche i link di comunicazione possano subire guasti, ma non più di due contemporaneamente, la disponibilità di una risorsa è garantita se esistono tre istanze della risorsa e ogni sito è connesso ad almeno tre link di comunicazione. In un sistema con tali caratteristiche, la risorsa in questione diventa indisponibile solo se avvengono contemporaneamente tre o più guasti.

Quando un nodo desidera inviare informazioni a tutti gli altri nodi di un sistema, esso potrebbe spedire messaggi contenenti le informazioni a ognuno dei suoi vicini, in modo che questi ultimi, alla prima ricezione di ogni messaggio lo inoltrino a loro volta ai propri vicini, e così via. Adottando tale meccanismo, un nodo riceverebbe la stessa informazione tante volte quanti sono gli archi a cui è connesso, e quindi sarebbero necessari  $m$  messaggi, con  $m$  pari al numero degli archi del sistema. Tuttavia, poiché ogni nodo necessita di ricevere ogni messaggio una sola volta, è possibile usare informazioni sulla topologia del sistema per ridurre il numero di messaggi da inviare. Per esempio, se il sistema è completamente connesso, basta usare un semplice protocollo secondo cui solo il nodo iniziale invia l'informazione a tutti i suoi vicini. Tale operazione richiederebbe chiaramente solo  $n - 1$  messaggi.

Sia i modelli fisici che quelli logici sono utili per determinare proprietà importanti. La latenza tra due nodi è determinata analizzando il modello fisico. Al contrario, l'analisi del modello logico è tipicamente utilizzata per determinare la complessità degli algoritmi di controllo usati in un sistema operativo distribuito. Approfondiremo tali aspetti nel [Capitolo 18](#).

## 16.8 Problematiche progettuali di un sistema operativo distribuito

L'utente di un sistema distribuito si aspetta che il suo sistema operativo abbia l'aspetto e le caratteristiche di un sistema operativo convenzionale, ma che garantisca, contemporaneamente, i benefici tipici di un sistema distribuito riassunti in [Tabella 16.1](#). Per soddisfare tali aspettative, il sistema operativo deve sfruttare appieno le capacità del sistema distribuendo i dati, le risorse, gli utenti e le relative applicazioni tra tutti i nodi in modo efficace. Da tale necessità emergono le seguenti problematiche progettuali.

### **Trasparenza delle risorse e dei servizi**

La *trasparenza* implica che i nomi delle risorse e dei servizi non devono dipendere dalla loro locazione nel sistema. Innanzitutto, ciò permette alle applicazioni di accedere in modo identico sia alle risorse locali che a quelle remote. In secondo luogo, permette al sistema operativo di cambiare liberamente la locazione di una risorsa, in quanto tale cambiamento non ha ripercussioni sul nome della risorsa e quindi non influenza le applicazioni che la stanno usando. Il sistema operativo può sfruttare la trasparenza per vari scopi, come, per esempio, migrazione dei dati per velocizzare le applicazioni, riduzione del traffico di rete, ottimizzazione dell'uso dei dischi. La trasparenza facilita, inoltre, la migrazione dei processi, in quanto l'elaborazione può continuare ad accedere alle stesse risorse esattamente come avveniva prima della migrazione. La trasparenza verrà trattata in dettaglio nel [Capitolo 19](#).

### **Distribuzione delle funzioni di controllo**

Una *funzione di controllo* è una funzione eseguita dal kernel per controllare le risorse e i processi del sistema. Esempi di funzioni di controllo sono: allocazione delle risorse, gestione dei deadlock, schedulazione dei processi. L'utilizzo di funzioni di controllo centralizzate all'interno di un sistema distribuito si scontra con due problemi principali. In primo luogo, a causa della latenza della rete, non è possibile ottenere informazioni consistenti relative allo stato corrente dei processi e delle risorse di tutti i nodi del sistema, pertanto una funzione centralizzata potrebbe non essere in grado di prendere le giuste decisioni. In secondo luogo, una funzione centralizzata rischia di diventare un potenziale collo di bottiglia e un punto critico (rispetto all'occorrenza di guasti) del sistema. Per gestire tali problemi, i sistemi operativi distribuiti implementano le funzioni di controllo tramite *algoritmi di controllo distribuiti*, le cui azioni sono effettuate nei vari nodi del sistema in modo coordinato. Nel [Capitolo 18](#) tratteremo alcuni di tali algoritmi, tra cui quelli per la gestione di deadlock, schedulazione dei processi e mutua esclusione.

### **Prestazioni del sistema**

Oltre alle tecniche adottate nei sistemi operativi convenzionali, i sistemi operativi distribuiti usano due strategie aggiuntive per garantire buone prestazioni complessive - migrazione dei dati e migrazione dell'elaborazione. La migrazione dei dati viene utilizzata per ridurre la latenza della rete e migliorare il tempo di risposta dei processi. La migrazione dell'elaborazione permette, invece, che il carico di lavoro venga distribuito equamente tra tutte le CPU del sistema in modo da ottenere ciò che in gergo informatico è noto come *bilanciamento del carico*.

La dimensione di un sistema distribuito tende a crescere nel tempo con l'aggiunta di nuovi nodi e utenti. Al crescere della dimensione del sistema, i tempi di risposta dei processi potrebbero però degradare, sia a causa dell'aumento del carico di lavoro assegnato alle risorse e ai servizi del sistema operativo, sia per il costo aggiuntivo richiesto per gestire le funzioni di controllo. A sua volta, il degrado delle prestazioni limita la crescita del sistema. È pertanto necessario che le prestazioni di un sistema distribuito siano *scalabili*. Ciò significa che al crescere della dimensione del sistema i ritardi e i tempi di risposta non dovrebbero aumentare, ma, al contrario, dovrebbe essere il throughput ad aumentare. Una tecnica per garantire la scalabilità consiste nell'usare cluster di host autosufficienti (Paragrafo 16.3) in modo che il traffico di rete non cresca via via che vengono aggiunti nuovi cluster al sistema. Nel [Capitolo 19](#), verrà spiegato come la tecnica del *file caching*, usata nei file system distribuiti, aiuti a raggiungere tale obiettivo.

### **Affidabilità**

Le tecniche per la tolleranza ai guasti mirano a garantire la continuità operativa del

sistema distribuito e la disponibilità delle sue risorse anche in caso di guasti. Eventuali guasti su link o nodi del sistema vengono generalmente tollerati aggiungendo un numero ridondante di risorse. In tal modo, se si presenta un guasto in una istanza di una determinata risorsa o in un percorso di rete che permette di raggiungere quest'ultima, le applicazioni potranno usare una diversa istanza della risorsa o un cammino differente per raggiungere la stessa istanza. Così facendo, la risorsa in questione diventa indisponibile solo nel caso di guasti inattesi.

Un altro aspetto rilevante per garantire l'affidabilità in un sistema distribuito è caratterizzato dalla necessità di gestire la consistenza dei dati. Se i dati di un sistema sono memorizzati in modo distribuito o sono replicati, è possibile che l'occorrenza di un guasto durante la loro modifica porti il sistema in uno stato di inconsistenza, dove alcune parti dei dati sono state aggiornate mentre altre no. Per evitare che ciò avvenga, i sistemi operativi distribuiti utilizzano una tecnica chiamata *commit a due fasi*.

Abbiamo già osservato come un'elaborazione possa essere suddivisa in varie parti, ciascuna eseguita su un diverso nodo di un sistema distribuito. Se si manifesta un guasto in un nodo o in un collegamento durante l'esecuzione di tale elaborazione, il sistema deve essere in grado, in primo luogo, di valutare il danno arrecato dal guasto e, successivamente, di *ripristinare* l'elaborazione riportandola a uno stato consistente precedentemente salvato tramite operazioni di backup. Inoltre, il sistema deve essere in grado di gestire le diverse situazioni che potrebbero essere state causa del guasto.

### Esempio 16.2 - Situazioni ambigue dovute a guasto

Si consideri un'applicazione distribuita costituita da due parti, rappresentate dai processi  $P_i$  e  $P_j$  in esecuzione, rispettivamente, sui nodi  $N_1$  e  $N_2$  (Figura 16.15). Il processo  $P_i$  invia una richiesta a  $P_j$  e si mette in attesa della relativa risposta. Tuttavia, il timeout impostato da  $P_i$  scatta prima dell'arrivo della risposta. Il timeout potrebbe essere stato causato da una qualunque delle seguenti situazioni.



**Figura 16.15** Problemi di ripristino in occasione di una richiesta remota.

1. Il processo  $P_j$  non ha mai ricevuto la richiesta da parte di  $P_i$ .
2. La gestione della richiesta da parte di  $P_j$  ha richiesto più tempo del previsto, e quindi allo scadere del timeout,  $P_j$  sta ancora processando la richiesta.
3. Il processo  $P_j$  ha iniziato a processare la richiesta, ma ha subito un guasto prima di poter completare il lavoro.
4. Il processo  $P_j$  ha completato la gestione della richiesta, ma la sua risposta è andata persa.

Nell'Esempio 16.2, il sistema operativo deve poter capire e, di conseguenza, gestire l'esatta causa che ha portato alla scadenza del timeout. Nel caso in cui il timeout sia dovuto al fatto che il nodo  $N_2$  abbia smesso di funzionare, il processo  $P_j$  dovrebbe essere rieseguito, eventualmente in un diverso nodo del sistema. Negli altri casi, invece, il processo  $P_j$  potrebbe aver già completato il suo lavoro, quindi la sua riesecuzione in un nodo diverso della rete potrebbe inficiare la consistenza dei dati (per esempio, potrebbe succedere che un dato venga aggiornato due volte) o semplicemente sprecare tempo di CPU.

### Sicurezza

L'aspetto relativo alla sicurezza acquista una nuova dimensione nel contesto di un sistema distribuito, poiché i messaggi scambiati nella rete passano attraverso alcuni calcolatori, i così detti *communication processor*, che possono operare in modo del tutto indipendente dal resto del sistema. Qualora un intruso prenda il controllo di un *communication processor*, egli può corrompere o modificare i dati contenuti nei messaggi che lo attraversano o usarli nel tentativo di mascherare la propria identità. Nel [Capitolo 20](#) verranno discusse alcune tecniche per gestire la sicurezza dei messaggi e i

meccanismi di autenticazione.

## Riepilogo

I benefici principali di un sistema distribuito sono la condivisione delle risorse, l'affidabilità, e la velocità di calcolo. Un sistema operativo distribuito implementa tali benefici integrando operazioni tipicamente eseguite su sistemi di calcolo isolati, garantendo una comunicazione affidabile attraverso la rete, e supportando in modo efficiente l'elaborazione distribuita. In questo capitolo, sono state presentate le tecniche principali implementate in un sistema operativo distribuito.

Un sistema distribuito è costituito da vari nodi connessi a una rete, dove ogni *nodo* può essere rappresentato da un singolo calcolatore oppure da un *cluster*, cioè un gruppo di calcolatori che condividono risorse e operano in modo strettamente integrato. Un cluster può garantire un sensibile aumento della velocità di calcolo e un elevato livello di affidabilità.

I processi che sono parte di una stessa *elaborazione distribuita* possono essere eseguiti su nodi diversi in modo da velocizzare l'elaborazione e trarre beneficio dalla condivisione delle risorse. Una tale elaborazione può usare dati posizionati in nodi remoti in tre modi: il meccanismo basato sull'accesso *remoto* permette di usare i dati attraverso la rete ovunque questi si trovino; la *migrazione dei dati* sposta i dati verso il nodo su cui è in esecuzione l'elaborazione che li usa; mentre la *migrazione dei processi* sposta parte del calcolo verso il nodo in cui sono posizionati i dati di interesse. Il *paradigma di calcolo distribuito* è un modello di calcolo che garantisce funzionalità per accesso remoto, migrazione dei dati e migrazione dei processi. Il modello *client-server* permette solo l'accesso remoto dei dati, mentre i modelli basati su *chiamata a procedura remota* (RPC) e *remote evaluation* permettono la migrazione del processo computazionale.

Processi in esecuzione su nodi diversi di un sistema distribuito comunicano tra loro usando un protocollo particolare detto *interprocess communication protocol* o *IPC protocol*. Il protocollo IPC è costituito da un insieme di regole che garantiscono una comunicazione efficace e utilizza il *domain name system* (DNS) per identificare la posizione dei processi destinatari. La *semantica IPC* descrive le proprietà del protocollo. Un protocollo è *affidabile* se è in grado di garantire che un messaggio raggiunga il processo destinatario indipendentemente dal verificarsi di guasti su nodi o link di comunicazione del sistema. L'affidabilità è ottenuta usando meccanismi di ritrasmissione. Tali meccanismi si basano sul fatto che alla ricezione di un messaggio, il processo ricevente deve inviare al mittente un messaggio speciale, detto *acknowledgment*. Qualora il mittente non riceva l'*acknowledgment* entro un determinato intervallo di tempo, egli invia nuovamente il messaggio. In base a questo protocollo, un messaggio potrebbe essere ricevuto dal destinatario più di una volta, pertanto in questo caso si parla di protocollo con semantica *at-least-once*. Al contrario, un protocollo che riconosce messaggi duplicati e li scarta automaticamente implementa una semantica di tipo *exactly-once*.

Le tecniche che implementano la comunicazione attraverso la rete devono affrontare problematiche derivanti da guasti a nodi e link del sistema, e devono saper gestire la densità di traffico nelle varie parti della rete. Quindi, a prescindere dalla semantica IPC, il software di rete deve assicurare un elevato livello di affidabilità e tolleranza ai guasti, deve essere in grado di instradare appropriatamente i messaggi sulla rete, e deve trasmettere i dati a un tasso sostenibile. La comunicazione attraverso la rete è implementata efficacemente tramite una gerarchia di protocolli chiamata *pila dei protocolli*, in cui ogni protocollo gestisce aspetti diversi della comunicazione. La pila di protocolli ISO è costituita da sette protocolli, mentre quella TCP/IP ha un numero di protocolli inferiore. Le prestazioni della rete possono essere misurate o tramite l'effettiva *larghezza di banda*, rappresentata dal tasso con cui i dati possono essere trasferiti nella rete, o tramite la *latenza*, ovvero il ritardo richiesto per il trasferimento dei dati.

Un sistema distribuito può essere modellato tramite grafi. Nel *modello fisico*, i nodi e gli archi del grafo rappresentano, rispettivamente, i nodi e i link del sistema distribuito. Nel *modello logico*, invece, i nodi e gli archi rappresentano, rispettivamente, processi e relazioni tra questi. Tali modelli sono usati sia per determinare le caratteristiche di

affidabilità di un sistema che per la progettazione degli algoritmi usati nei sistemi operativi distribuiti.

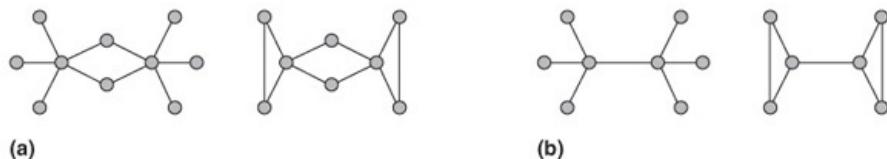
Le problematiche affrontate dai progettisti di un sistema operativo distribuito, quali la condivisione delle risorse, l'affidabilità e le prestazioni in ambiente distribuito, saranno discusse nei prossimi capitoli.

## Domande di riepilogo

- 16.1. Classificare ciascuna delle affermazioni seguenti come vera o falsa.
- Un guasto su un nodo partiziona una rete ad anello.
  - Quando si usa la commutazione di messaggio, tutti i messaggi scambiati tra una coppia di processi percorrono lo stesso cammino sulla rete.
  - L'instradamento dinamico si adatta ai guasti che possono occorrere sui nodi o sui link della rete.
  - Un messaggio inviato usando un cammino virtuale su una rete ATM può subire un ritardo attraversando un link a causa di una elevata densità di traffico.
  - La semantica at-least-once è implementata tramite un protocollo che riconosce e scarta i messaggi duplicati.
  - Il numero di sequenza di un messaggio esercita un ruolo attivo nell'implementazione della semantica di un protocollo IPC.
  - In un protocollo IPC affidabile e non bloccante, un processo ricevente può mantenere solo un buffer di risposta per ogni processo mittente.
  - Una chiamata a procedura remota è utile per implementare la migrazione dei dati.
  - Il trasferimento di  $n$  byte di dati tra due nodi richiede solo il 50% del tempo necessario per trasferire  $2 \times n$  byte.

## Problemi

- 16.1. Spiegare quali tra i meccanismi per la sincronizzazione dei processi usati in un sistema multiprocessore simmetrico possono essere adottati in un cluster ([Capitolo 10](#)).
- 16.2. Analizzare la possibilità di implementare protocolli bloccanti e non bloccanti attraverso i monitor. Quali sono le difficoltà implementative?
- 16.3. Descrivere brevemente i fattori che influenzano la durata dell'intervallo di timeout nel protocollo RRA illustrato nel Paragrafo 16.4.3.
- 16.4. Sviluppare schemi per scartare risposte duplicate ricevute nel sito del mittente, quando si usano la versione bloccante e non bloccante del protocollo RRA.
- 16.5. Quando si usa l'instradamento dinamico, le richieste effettuate da chiamate non bloccanti alla funzione *send* possono arrivare a destinazione in un ordine diverso da quello di invio. In tale contesto, spiegare il modo con cui un protocollo RR non bloccante dovrebbe scartare le richieste duplicate (Paragrafo 16.4.3).
- 16.6. Analizzare le proprietà del protocollo RRA presentato nel Paragrafo 16.4.3 dopo aver applicato la seguente modifica: il processo destinatario si blocca finché non riceve l'acknowledgment della sua risposta.
- 16.7. a. Determinare i guasti sui siti e sui link che possono essere tollerati dai sistemi mostrati in [Figura 16.16\(a\)](#).



**Figura 16.16** Esercitazione sulla resiliency di un sistema distribuito.

- b. Determinare dove dovrebbero essere posizionate le copie di un dato  $D$  nei sistemi di [Figura 16.16\(b\)](#) se  $D$  deve essere disponibile anche in presenza di due guasti su siti o link.
- 16.8. Sia  $C$  l'insieme dei cammini minimi tra tutte le coppie di nodi di un sistema distribuito. Il *diametro* del sistema,  $d$ , è dato dal numero di link presenti nel cammino  $c \in C$  formato dal maggior numero di link. Se il massimo ritardo di comunicazione lungo ogni link del sistema è  $\delta$ , indicare qual è il massimo ritardo di comunicazione del sistema e spiegare le condizioni che lo determinano.
- 16.9. Comparare i paradigmi RPC e remote evaluation sulla base delle seguenti caratteristiche:
- flessibilità;
  - efficienza;
  - sicurezza.

## Note bibliografiche

L'articolo proposto da Tanenbaum e van Renesse (1985) presenta una panoramica sui sistemi operativi distribuiti con particolare riferimento alle differenze tra protocolli di comunicazione bloccanti e non bloccanti. Tutti gli argomenti trattati in questo capitolo sono presentati anche nei libri di Sinha (1997), Tanenbaum e van Steen (2002), e Coulouris et al. (2005).

Il lavoro di Comer e Stevens (2000) descrive il modello di computazione client-server. L'articolo di Birrell e Nelson (1984) è focalizzato sull'implementazione delle chiamate a procedura remota, mentre Tay e Ananda (1990) riassumono le caratteristiche peculiari di tale paradigma di calcolo. L'articolo di Lin e Gannon (1985) descrive uno schema di RPC che implementa un semantica di tipo exactly-once. La remote evaluation è trattata da Stamos e Gifford (1990). Nel testo del Tanenbaum (2001) sono presentati il protocollo ISO, il modello client-server e il paradigma RPC, mentre Birman (2005) si focalizza solo sul modello clientserver e sulle RPC.

Infine, le tematiche relative alle reti di calcolatori sono abbondantemente trattate nei testi di Comer (2004), con particolare riguardo al protocollo TCP/IP, e Tanenbaum (2003), che invece è più focalizzato sul protocollo ISO. Una panoramica dei vari protocolli di rete è presentata anche da Stallings (2004), mentre Stevens e Rago (2005) descrivono le problematiche relative alla programmazione di rete in ambiente Unix.

1. Birman, K. (2005): *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Springer, Berlin.
2. Birrell, A.D., and B.J. Nelson (1984): "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, **2**, 39-59.
3. Comer, D. (2004): *Computer Networks and Internets*, 4th ed., Prentice Hall, Englewood Cliffs, N.J.
4. Comer, D., and D. Stevens (2000): *Internetworking with TCP/IP*, Vol. III: *Client-Server Programming and Applications, Linux/POSIX Socket Version*, Prentice Hall, Englewood Cliffs, N.J.
5. Coulouris, G., J. Dollimore, and T. Kindberg (2005): *Distributed Systems-Concepts and Design*, 4th ed., Addison-Wesley, New York.
6. Lin, K.J., and J.D. Gannon (1985): "Atomic remote procedure call," *IEEE Transactions on Software Engineering*, **11** (10), 1126-1135.
7. Sinha, P.K. (1997): *Distributed Operating Systems*, IEEE Press, New York.
8. Stallings, W. (2004): *Computer Networking with Internet Protocols*, Prentice Hall, Englewood Cliffs, N.J.
9. Stamos, J.W., and D.K. Gifford (1990): "Remote evaluation," *ACM Transactions on Programming Languages and Systems*, **12** (4), 537-565.
10. Stevens, W.R., and S.A. Rago (2005): *Advanced Programming in the Unix Environment*, 2nd ed., Addison-Wesley Professional.
11. Tanenbaum, A.S. (2001): *Modern Operating Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, N.J.
12. Tanenbaum, A.S. (2003): *Computer Networks*, 4th ed., Prentice Hall, Englewood Cliffs, N.J.
13. Tanenbaum, A.S., and M. van Steen (2002): *Distributed Systems: Principles and Paradigms*, Prentice Hall, Englewood Cliffs, N.J.

- 14. Tanenbaum, A.S., and R. Van Renesse (1985): "Distributed Operating Systems," *Computing Surveys*, **17** (1), 419-470.
- 15. Tay, B.H., and A.L. Ananda (1990): "A survey of remote procedure calls," *Operating Systems Review*, **24** (3), 68-79.

---

# CAPITOLO 17

## Problematiche teoriche di un sistema distribuito

---

### Obiettivi di apprendimento

- Nozioni di tempo e stato in un sistema distribuito
- Relazione di precedenza tra gli eventi di un sistema distribuito
- Clock logici, vettori di clock e loro sincronizzazione
- Registrazione dello stato globale di un sistema distribuito

I concetti di *tempo* e *stato* sono due aspetti fondamentali per un sistema operativo – esso, infatti, per effettuare l'allocazione delle risorse e la schedulazione dei processi, ha bisogno di conoscere sia l'ordine cronologico degli eventi che avvengono nel sistema, come per esempio l'ordine in cui vengono richieste le risorse, sia lo stato corrente delle risorse e dei processi. In un sistema di calcolo convenzionale, la gestione dei concetti di tempo e stato è semplificata dal fatto che esiste un'unica memoria e un'unica CPU. In un tale ambiente, in ogni istante può avvenire un unico evento, quindi la conoscenza dell'ordine degli eventi è implicita. Inoltre, il sistema operativo di un calcolatore convenzionale conosce lo stato di tutti i processi e di tutte le risorse locali.

Un sistema distribuito è costituito da molti sistemi di calcolo, ognuno con clock e memoria propri, dotati di una o più CPU. Tali sistemi comunicano tra loro tramite scambio di messaggi che possono subire ritardi non prevedibili. Di conseguenza, un sistema operativo distribuito non può conoscere né l'ordine cronologico degli eventi, né lo stato delle risorse e dei processi di tutti i nodi del sistema nello stesso istante di tempo. Perciò le problematiche teoriche più rilevanti relative alla progettazione di sistemi distribuiti sono: la definizione di alternative pratiche alle tradizionali nozioni di tempo e spazio, lo sviluppo di algoritmi per implementare tali alternative, e la dimostrazione della correttezza di questi algoritmi.

In questo capitolo verrà innanzitutto presentata la nozione di *precedenza degli eventi* che può essere usata per mantenere l'ordine cronologico in cui avvengono *alcuni* eventi in un sistema distribuito. Quindi, si descriveranno i concetti di *clock logico* e *vettori di clock*, che permettono di definire due alternative alla tradizionale nozione di tempo. Infine, verrà presentata la nozione di *stato consistente* che può essere usata in molte applicazioni come una valida alternativa al concetto di *stato globale* di un sistema distribuito. I concetti di tempo e stato definiti in questo capitolo sono utilizzati nella progettazione di molti algoritmi di controllo e algoritmi di ripristino implementati in un sistema operativo distribuito.

### 17.1 Le nozioni di tempo e stato

Il concetto di *tempo*, considerato la quarta dimensione; è necessario per indicare il momento in cui avviene un evento. Il concetto di *stato*, invece, dipende dalle caratteristiche dell'entità a cui è riferito; per esempio, lo stato di una cella di memoria è rappresentato dal valore contenuto in essa. Se una entità è costituita da più sottoentità, il suo stato corrisponde allo stato delle sue componenti. Lo *stato globale* di un sistema comprende gli stati di tutte le entità presenti nel sistema in un preciso istante di tempo. Un sistema operativo usa le nozioni di tempo e stato per effettuare la schedulazione delle risorse e della CPU. In particolare, il sistema operativo utilizza il tempo per conoscere l'istante in cui viene effettuata una determinata richiesta, o per definire l'*ordine cronologico* di tutte le richieste. Conoscere lo stato di una risorsa, invece, permette al sistema operativo di decidere se essa possa essere allocata. Un sistema operativo distribuito usa queste nozioni anche per effettuare operazioni di ripristino. In tal caso infatti, è fondamentale che il sistema operativo sia in grado di garantire che i processi

appartenenti a una applicazione distribuita vengano riportati in stati mutualmente consistenti a fronte del riavvio di un nodo su cui alcuni di essi erano in esecuzione al momento del guasto.

Nei sistemi monoprocesso, i concetti di tempo e stato sono semplici da gestire. In essi, infatti, esiste un unico clock e una sola CPU. Pertanto, il sistema operativo conosce il tempo in cui i processi hanno effettuato le richieste e di conseguenza è in grado di determinare il corrispondente ordine cronologico. In pratica, i sistemi operativi convenzionali usano il concetto di tempo in modo implicito. Quando si verifica un evento, infatti, esso viene inserito in una coda che ne determina implicitamente l'ordine cronologico. Il sistema operativo di un'architettura monoprocesso conosce anche lo stato di tutti i processi in esecuzione e di tutte le risorse logiche e fisiche del sistema.

In un sistema distribuito, ogni nodo rappresenta un sistema di calcolo con clock e memoria propri, ed è connesso agli altri nodi tramite collegamenti caratterizzati da ritardi di comunicazione non prevedibili. Di conseguenza, un nodo non può determinare precisamente il tempo in cui si verifica un evento in un altro nodo; inoltre, la sua percezione dello stato dei processi e delle risorse remote può essere obsoleta. Un sistema operativo distribuito non può quindi usare le stesse nozioni di tempo e stato adottate da un sistema operativo monoprocesso. In questo capitolo, vedremo come sia possibile sviluppare alternative, da adottare in ambiente distribuito, alle nozioni di tempo e stato usate nei sistemi monoprocesso. Tali nozioni saranno usate anche nel [Capitolo 18](#) per la progettazione di algoritmi di controllo distribuiti.

## 17.2 Stati ed eventi in un sistema distribuito

### 17.2.1 Stato locale e stato globale

Ogni entità di un sistema possiede un proprio stato. Lo stato di una cella di memoria è rappresentato dal valore contenuto in essa. Lo stato di una CPU è costituito dal contenuto del suo PSW e dei suoi registri. Lo stato di un processo è composto da vari aspetti: il suo stato di esecuzione (pronto, in attesa, in esecuzione, ...), lo stato del suo spazio di indirizzamento in memoria, lo stato della CPU o il contenuto dei campi del suo PCB, rispettivamente quando il processo è o non è nella coda di schedulazione della CPU, lo stato delle sue comunicazioni con altri processi (ovvero lo stato delle informazioni relative ai messaggi ricevuti e inviati). Lo stato di una entità è detto *stato locale*. Lo *stato globale* di un sistema all'istante di tempo  $t$ , invece, è dato dall'insieme degli stati locali di tutte le sue entità al tempo  $t$ .

Nel seguito, indicheremo lo stato locale di un processo  $P_k$  al tempo  $t$  con la notazione  $s_k^t$ , dove il pedice sarà omesso se l'identità del processo è implicitamente derivabile dal contesto. Lo stato globale di un sistema al tempo  $t$  sarà invece indicato con la notazione  $S_t$ . Se un sistema contiene  $n$  processi,  $P_1, \dots, P_n$ ,  $S_t \equiv \{s_1^t, s_2^t, \dots, s_n^t\}$ .

### 17.2.2 Gli eventi

Un *evento* può essere rappresentato dall'invio o dalla ricezione di un messaggio attraverso un *canale*, ovvero un cammino di comunicazione, ma anche da altri avvenimenti che non hanno nulla a che fare con i messaggi. In ogni caso, all'occorrenza di un evento su un processo, lo stato di quest'ultimo viene modificato. Un evento verrà rappresentato come segue:

*(identificatore del processo, vecchio stato, nuovo stato,  
descrizione dell'evento, canale, messaggio)*

dove *canale* e *messaggio* saranno indicati con il simbolo “-” qualora l'evento non sia relativo né all'invio né alla ricezione di un messaggio. Per esempio, l'evento  $e_i \equiv (P_k, s, s', send, c, m)$  corrisponde all'invio del messaggio  $m$  sul canale  $c$  da parte del processo  $P_k$ , il quale trovandosi nello stato iniziale  $s$ , entrerà nello stato finale  $s'$  ([Figura 17.1](#)).



**Figura 17.1** Cambiamento di stato di un processo  $P_k$  all'occorrenza di un evento ( $P_k, s, s', send, c, m$ ).

Alcuni eventi possono verificarsi solo in determinate condizioni; per esempio, l'evento corrispondente alla ricezione del messaggio  $m$  sul canale  $c$  può avvenire solo se  $c$  contiene  $m$ . Pertanto, dato lo stato corrente di un processo, solo alcuni eventi sono possibili.

### 17.3 Tempo, clock e precedenze tra eventi

Il *clock globale* di un sistema distribuito può essere costituito da un clock astratto a cui accedono tutti i siti del sistema ottenendo le stesse risposte. In tal modo, se due processi su nodi distinti effettuano una chiamata a sistema per conoscere il tempo corrente nello stesso istante, essi otterranno lo stesso valore. Al contrario, se la stessa chiamata viene eseguita a una distanza di  $\delta$  unità di tempo, i due processi otterranno valori che differiscono esattamente della quantità  $\delta$ . Sfortunatamente, un clock globale di tal genere è impossibile da implementare in pratica, a causa dei ritardi di comunicazione dovuti alla rete. Infatti, le richieste effettuate da due nodi differenti, nello stesso istante, per ottenere il valore del tempo corrente, possono subire l'effetto di ritardi di comunicazione di natura diversa prima di raggiungere il sito che gestisce il clock globale. Di conseguenza, le corrispondenti risposte fornirebbero valori diversi. Analogamente, due richieste effettuate a distanza di  $\delta$  unità di tempo l'una dall'altra non riceverebbero risposte con valori differenti esattamente per una quantità  $\delta$ .

Un'alternativa al clock globale precedentemente descritto, può essere ottenuta usando un *clock locale* per ogni processo a cui sia possibile accedere, tramite un'opportuna chiamata a sistema, ognqualvolta sia necessario conoscere il tempo corrente. Per implementare un servizio efficace di rilevamento dei tempi, i clock locali devono essere sincronizzati in modo opportuno. Il Paragrafo 17.3.2 descrive come sia possibile ottenere tale sincronizzazione usando la nozione di precedenza tra eventi.

#### 17.3.1 Precedenza tra eventi

Per indicare che un evento  $e_1$  avviene prima dell'evento  $e_2$ , ovvero che  $e_1$  precede  $e_2$ , si usa la notazione  $e_1 \rightarrow e_2$ . Così facendo è possibile organizzare un insieme di eventi in una sequenza ordinata, in cui ogni evento è il predecessore del successivo. Tale sequenza può essere usata in pratica per determinare l'ordine in cui gli eventi si verificano nel sistema, ovvero per definire un *ordinamento degli eventi*. Diremo che esiste un *ordine totale* rispetto alla relazione di precedenza " $\rightarrow$ ", se è possibile ordinare tutti gli eventi che avvengono in un sistema. Se invece solo alcuni eventi possono essere ordinati, ovvero, per essere più precisi, se esistono almeno due eventi che non possono essere ordinati, diremo che esiste un *ordine parziale*.

La **Tabella 17.1** riassume le regole fondamentali necessarie per ordinare gli eventi in un sistema distribuito. Il sistema operativo può determinare facilmente la precedenza tra eventi che occorrono all'interno dello stesso processo. Per definire un ordine tra eventi che occorrono in nodi diversi, è necessario invece osservare eventuali relazioni di causalità. Per esempio, eventi quali l'esecuzione di " $send P_3, <message m_i>$ " da parte del processo  $P_2$  e la ricezione del messaggio  $m_i$  da parte di  $P_3$  sono in relazione causale tra loro. Di conseguenza, l'evento corrispondente all'invio del messaggio in  $P_2$ , ovvero la causa, precede l'evento corrispondente alla ricezione dello stesso messaggio in  $P_3$ , ovvero l'effetto. Va osservato inoltre che la relazione di precedenza è transitiva, quindi  $e_1 \rightarrow e_3$  se  $e_1 \rightarrow e_2$  e  $e_2 \rightarrow e_3$ . Questa proprietà può essere usata per determinare la precedenza tra alcuni eventi che non sono in relazione causa-effetto tra loro, e non avvengono nello stesso processo. Per esempio, un evento  $e_i$ , precedente all'invio del messaggio  $m_i$  in  $P_2$ , precede un evento  $e_j$  successivo alla ricezione del messaggio  $m_i$  in

$P_3$ , poiché  $e_i$  precede l'invio del messaggio, che a sua volta precede la ricezione del messaggio, che a sua volta precede  $e_j$ .

| Categoria                    | Descrizione della regola                                                                                                                                                                                                       |
|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Eventi interni a un processo | La gestione degli eventi viene effettuata dal sistema operativo, esso pertanto conosce l'ordine in cui si verificano gli eventi di uno stesso processo.                                                                        |
| Eventi in processi diversi   | In una relazione <i>causal</i> , ovvero in una relazione di causa-effetto, l'evento corrispondente alla causa, che occorre in un processo, precede l'evento corrispondente all'effetto, che si manifesta su un altro processo. |
| Precedenza transitiva        | La relazione di precedenza è transitiva, ovvero $e_1 \rightarrow e_2$ e $e_2 \rightarrow e_3$ implica $e_1 \rightarrow e_3$ .                                                                                                  |

**Tabella 17.1** Regole per ordinare gli eventi in un sistema distribuito.

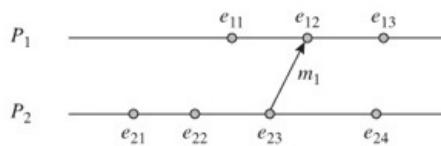
Usando le regole riportate in [Tabella 17.1](#), la precedenza tra ogni coppia di eventi  $e_i, e_j$  può essere classificata come segue:

- $e_i$  precede  $e_j$ : se esistono due eventi  $e_k$  ed  $e_l$  tali che  $e_k \rightarrow e_l$ ,  $e_i \rightarrow e_k$  oppure  $e_i \equiv e_k$ , ed  $e_l \rightarrow e_j$  oppure  $e_l \equiv e_j$ ;
- $e_i$  segue  $e_j$ : se esistono due eventi  $e_g$  ed  $e_h$  tali che  $e_g \rightarrow e_h$ ,  $e_j \rightarrow e_g$  oppure  $e_j \equiv e_g$ , ed  $e_h \rightarrow e_i$  oppure  $e_h \equiv e_i$ ;
- $e_i$  ed  $e_j$  sono concorrenti: se  $e_i$ , né precede né segue  $e_j$ .

Per schematizzare le attività di vari processi nel tempo, viene solitamente usato un *diagramma temporale* in cui i processi sono indicati sull'asse verticale, mentre il tempo è indicato sull'asse orizzontale. In tale diagramma, useremo la notazione  $e_{kn}$  per denotare l'evento  $e_n$  che si verifica nel processo  $P_k$ . L'Esempio 17.1 mostra come il diagramma temporale possa essere usato per determinare le precedenze tra eventi sfruttando la proprietà transitiva della relazione di precedenza. Inoltre, lo stesso esempio dimostra che in un sistema distribuito non è possibile ottenere un ordine totale degli eventi.

### Esempio 17.1 - Precedenza tra eventi

La [Figura 17.2](#) mostra gli eventi che occorrono in due processi,  $P_1$  e  $P_2$ . L'evento  $e_{23}$  corrisponde all'invio del messaggio  $m_1$ , mentre l'evento  $e_{12}$  corrisponde alla ricezione del messaggio  $m_1$ . Quindi  $e_{23} \rightarrow e_{12}$  e, per la proprietà transitiva della relazione “ $\rightarrow$ ”,  $e_{22} \rightarrow e_{12}$  e  $e_{21} \rightarrow e_{12}$ . Per transitività è vero anche che  $e_{22} \rightarrow e_{13}$  ed  $e_{21} \rightarrow e_{13}$ . L'evento  $e_{11}$ , invece, è concorrente rispetto agli eventi  $e_{21}$ ,  $e_{22}$ ,  $e_{23}$  ed  $e_{24}$ .



**Figura 17.2** Diagramma temporale per l'analisi della precedenza tra eventi.

### 17.3.2 Clock logici

Un sistema operativo necessita di un metodo pratico ed efficiente per ordinare gli eventi, e quindi non può basarsi sull'analisi delle relazioni di causalità. Inoltre, lo stesso metodo dovrebbe essere in grado di fornire un ordine totale degli eventi, così che il sistema operativo possa servire le richieste in modo FCFS. Tale ordine può essere ottenuto:

- incorporando precedenze tra eventi nell'ordine degli eventi;
- ordinando in modo arbitrario eventi concorrenti, per esempio, gli eventi  $e_{11}$  ed  $e_{21}$  in [Figura 17.2](#).

È possibile definire un modo semplice per ordinare gli eventi basandosi sul concetto di *timestamp* e sul fatto che ogni processo è associato a un *clock* locale a cui solo esso può accedere. Il timestamp di un evento è pertanto il valore assunto dal *clock* locale del processo corrispondente all'evento nel momento in cui quest'ultimo si verifica.

Supponiamo, per esempio, che  $ts(e_i)$  rappresenti il timestamp dell'evento  $e_i$ .

L'ordinamento degli eventi viene effettuato in accordo con i timestamp degli eventi, ovvero,  $\forall e_i, e_j: e_i \rightarrow e_j$  se  $ts(e_i) < ts(e_j)$  ed  $e_j \rightarrow e_i$  se  $ts(e_i) > ts(e_j)$ . Tuttavia, i *clock* locali di processi in esecuzione su nodi diversi possono non essere sincronizzati, a causa della possibile mancanza di sincronismo tra i *clock* di sistema dei rispettivi nodi, riducendo di conseguenza il livello di affidabilità di un meccanismo di ordinamento basato su timestamp. Per esempio, se l'evento  $e_i$  si verifica prima dell'evento  $e_j$ ,  $ts(e_i)$  dovrebbe essere minore di  $ts(e_j)$ ; ma, se il *clock* del processo in cui si verifica  $e_i$  è più veloce del *clock* del processo in cui si verifica  $e_j$ ,  $ts(e_i)$  potrebbe essere maggiore di  $ts(e_j)$ . Per evitare simili situazioni, è necessario sincronizzare i *clock* di tutti i processi.

La sincronizzazione dei *clock* locali è ottenuta usando la relazione di causalità tipica dello scambio di messaggi tra due processi. Si consideri un messaggio  $m$  inviato dal processo  $P_k$  al processo  $P_l$  e si supponga che i timestamp degli eventi corrispondenti all'invio e alla ricezione del messaggio siano, rispettivamente,  $t_k$  e  $t_l$ . La relazione di causa-effetto tra l'invio e la ricezione del messaggio implica che  $t_k$  deve essere minore di  $t_l$ . In caso contrario, l'anomalia può essere corretta incrementando il tempo del *clock* logico del processo ricevente in modo che risulti essere più grande di  $t_k$  prima di assegnare il timestamp all'evento corrispondente alla ricezione del messaggio.

L'Algoritmo 17.1 mostra formalmente come è possibile realizzare un tale meccanismo di sincronizzazione.

L'Algoritmo 17.1 implementa un meccanismo di sincronizzazione lasco poiché i *clock* dei due processi coinvolti sono mutuamente consistenti in corrispondenza dello scambio di un messaggio, ma possono essere invece discordanti in tutti gli altri istanti (Problema 17.1). La qualità della sincronizzazione dei *clock* dipende dalla frequenza con cui i processi si scambiano messaggi – maggiore è la frequenza di comunicazione, più accurata sarà la sincronizzazione. Un modo per migliorare la sincronizzazione consiste nell'usare messaggi speciali che vengono scambiati a intervalli brevi con l'unico scopo di preservare la sincronizzazione dei *clock*.

#### Algoritmo 17.1 (Sincronizzazione dei *clock*)

1. Quando un processo  $P_k$  desidera inviare un messaggio  $m$  al processo  $P_l$ :  $P_k$  esegue il comando “*send*  $P_l$ ,  $(ts(send(m)), m)$ ,” dove  $ts(send(m))$  è il timestamp ottenuto esattamente prima di inviare il messaggio  $m$ .
2. Quando il processo  $P_l$  riceve un messaggio: il processo  $P_l$  effettua le seguenti azioni
  - se  $clock$  locale ( $P_l$ )  $< ts(send(m))$  allora
  - $clock$  locale ( $P_l$ ):  $= ts(send(m)) + \delta$
  - assegna il timestamp all'evento corrispondente alla ricezione di  $m$

dove  $clock$  locale ( $P_l$ ) è il valore del *clock* locale del processo  $P_l$  e  $\delta$  è il ritardo di comunicazione medio della rete.

Va osservato che un tale meccanismo di sincronizzazione non è necessariamente in relazione con il tempo “reale”. Per esempio, se il *clock* del processo mittente è veloce, il *clock* del processo ricevente viene incrementato di conseguenza. Dal momento che i *clock* locali non mantengono il tempo “reale”, non è pertanto nemmeno necessario far avanzare il *clock* a ogni istante di tempo. È sufficiente che un processo incrementi il suo *clock* locale di una unità in corrispondenza di un evento, e che il *clock* locale venga sincronizzato, se necessario, in corrispondenza della ricezione di un messaggio. Un *clock* che si comporta in tal modo viene detto *clock logico*. Nel seguito, denoteremo il *clock* logico di un processo  $P_k$  con  $LC_k$ . I *clock* logici sono implementati in base alle seguenti regole:

- R1 un processo  $P_k$  incrementa  $LC_k$  di 1 ogni volta si verifica un evento su di esso;
- R2 quando il processo  $P_k$  riceve un messaggio  $m$  contenente  $ts(send(m))$ ,  $P_k$  setta il suo

clock in base a  $LC_k = \max (LC_k, ts(send(m)) + 1)$ .

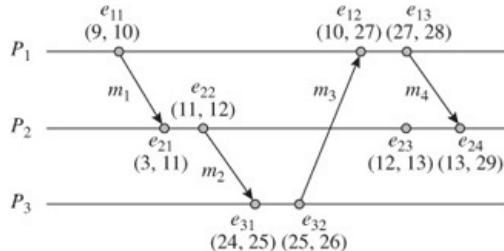
Il prossimo esempio mostra come sia possibile usare le regole precedenti per la sincronizzazione dei clock.

### Esempio 17.2 - Sincronizzazione dei clock logici

La [Figura 17.3](#) mostra i diagrammi temporali di un sistema costituito da tre processi. La coppia di numeri che appare in parentesi sotto ogni evento indica i valori del clock logico del processo prima e dopo l'evento. Il clock logico in  $P_1$  ha il valore 9 quando  $P_1$  decide di inviare il messaggio  $m_1$  a  $P_2$ . Al momento dell'invio, il clock viene incrementato in base alla regola R1 e il suo nuovo valore viene usato come timestamp dell'evento  $e_{11}$  corrispondente all'invio del messaggio  $m_1$ . Pertanto,  $m_1$  contiene il timestamp 10. Quando  $P_2$  riceve  $m_1$ , il valore del suo clock è pari a 3. Quindi,  $P_2$  prima porta il clock al valore 4 usando la regola R1 e, successivamente, lo sincronizza in base alla regola R2 portandolo al valore 11.

In seguito, quando  $P_2$  decide di inviare il messaggio  $m_2$ , il suo clock logico viene portato al valore 12 e, di conseguenza,  $m_2$  contiene il timestamp 12. Quando  $m_2$  arriva a destinazione,  $P_3$  applica le regole R1 e R2. Tuttavia, la regola R2 non ha alcun effetto dal momento che il valore del clock logico di  $P_3$  è maggiore del timestamp contenuto nel messaggio. Infine, quando  $P_3$  invia  $m_3$  a  $P_1$ , il clock di quest'ultimo viene sincronizzato con quello di  $P_3$  e, similmente, il clock di  $P_2$  si sincronizza con il clock di  $P_1$  quando  $P_1$  invia  $m_4$  a  $P_2$ .

I timestamp ottenuti usando i clock logici hanno la seguente proprietà:  $ts(e_i) < ts(e_j)$  se  $e_i \rightarrow e_j$ . Tuttavia, l'implicazione opposta non è garantita per eventi che si verificano in processi diversi; ovvero,  $e_i$  potrebbe non essere un predecessore di  $e_j$  anche se  $ts(e_i) < ts(e_j)$ . Tale situazione può presentarsi quando  $e_i$  ed  $e_j$  avvengono, rispettivamente, in due processi  $X, Y$ , senza che vi sia una sincronizzazione diretta dei clock di  $X$  e  $Y$ . La mancanza di sincronizzazione può essere dovuta sia all'assenza di scambio di messaggi tra i due processi, sia al fatto che il clock del processo  $Y$  è più veloce di quello del processo  $X$  (a causa dell'occorrenza di un maggior numero di eventi su  $Y$  rispetto a quanto non avvenga su  $X$ ). Una situazione di tal genere è rappresentata in [Figura 17.3](#), dove  $e_{32}$  si verifica "prima di"  $e_{23}$  ma è associato a un timestamp maggiore di quello di  $e_{23}$ .



**Figura 17.3** Sincronizzazione di clock logici.

### Timestamp univoci

In alcune situazioni, gli eventi che si verificano in processi diversi possono ottenere timestamp identici. Ciò accade se gli eventi avvengono quando i clock logici dei corrispondenti processi hanno lo stesso valore. Di conseguenza, tali timestamp non possono essere usati per ottenere un ordinamento totale degli eventi. Questo problema può essere risolto usando come timestamp di un evento  $e_i$  la seguente coppia di valori:

$$pts(e_i) \equiv (tempo\ locale, id\ del\ processo)$$

In tal modo, gli eventi non possono mai avere timestamp identici e il loro ordinamento può essere effettuato definendo una relazione di precedenza tra eventi come segue:

$$\begin{aligned}
e_i \text{ precede } e_j \text{ se } & \quad (i) \ pts(e_i).\text{tempo locale} < pts(e_j).\text{tempo locale}, \text{ o} \\
& \quad (ii) \ pts(e_i).\text{tempo locale} = pts(e_j).\text{tempo locale} \text{ e} \\
& \quad pts(e_i).\text{id del processo} < pts(e_j).\text{id del processo}
\end{aligned} \tag{17.1}$$

dove  $pts(e_i).\text{tempo locale}$  e  $pts(e_i).\text{id del processo}$  rappresentano, rispettivamente, il tempo locale e l'identificatore del processo in cui avviene  $pts(e_i)$ . Va osservato che questa nozione di precedenza tra eventi fornisce lo stesso ordinamento per  $e_i$  ed  $e_j$  che si otterrebbe usando i clock logici, qualora i processi avessero tempi locali diversi nel momento in cui si verificano  $e_i$  ed  $e_j$ .

### 17.3.3 Vettori di clock

Un vettore di clock è un vettore contenente  $n$  elementi, uno per ogni processo in esecuzione su un sistema distribuito. Denoteremo il vettore di clock di un processo  $P_k$  con  $VC_k$ , e il suo  $l$ -esimo elemento con  $VC_k[l]$ . Gli elementi di un vettore di clock  $VC_k$  hanno il seguente significato:

$$\begin{aligned}
VC_k[k] & \quad \text{il clock logico del processo } P_k; \\
VC_k[l], l \neq k & \quad \text{il più alto tra i valori dei clock logici del processo } P_l \text{ noti al} \\
& \quad \text{processo } P_k - \text{cioè il valore più alto di } VC_l[l] \text{ noto a } P_k.
\end{aligned}$$

Il timestamp di un evento  $e_i$  che si verifica in un processo  $P_k$  è il valore assunto da  $VC_k$  quando avviene  $e_i$ . Pertanto, il timestamp è rappresentato da un vettore di elementi, detto *vector timestamp*. Denoteremo il vector timestamp di un evento  $e_i$  con  $vts(e_i)$ , e l'elemento  $l$ -esimo di  $vts(e_i)$  con  $vts(e_i)[l]$ . Quando il processo  $P_k$  invia un messaggio  $m$  al processo  $P_l$ , esso include nel messaggio il valore  $vts(send(m))$ . I vettori di clock sono implementati in base alle seguenti regole:

R3 un processo  $P_k$  incrementa  $VC_k[k]$  di 1 ogni volta si verifica un evento su di esso;

R4 quando il processo  $P_k$  riceve un messaggio  $m$  contenente  $vts(send(m))$ ,  $P_k$  setta il suo clock come segue:

$$\text{per ogni } l: VC_k[l] = \max(VC_k[l], vts(send(m))[l])$$

In accordo con le regole precedenti,  $VC_k[k] \geq VC_l[k]$  per ogni  $l$ . Se  $e_i$  ed  $e_j$  sono due eventi consecutivi nel processo  $P_k$ , per la regola R3 vale che  $vts(e_j)[k] = vts(e_i)[k] + 1$ . Se, invece,  $e_i$  ed  $e_j$  sono eventi corrispondenti all'invio e alla ricezione di un messaggio, rispettivamente, nei processi  $P_g$  e  $P_k$ ,  $P_k$  incrementa  $VC_k[k]$  in base alla regola R3 quando si verifica  $e_j$ , e quindi aggiorna  $VC_k$  in base alla regola R4 prima di assegnare il timestamp a  $e_j$ . Di conseguenza,  $vts(e_i)[l] \leq vts(e_j)[l]$ , per ogni  $l$  e  $vts(e_i)[k] < vts(e_j)[k]$ . Rappresenteremo questa condizione come  $vts(e_i) < vts(e_j)$ . La precedenza tra due eventi  $e_i$  ed  $e_j$  si ottiene come segue:

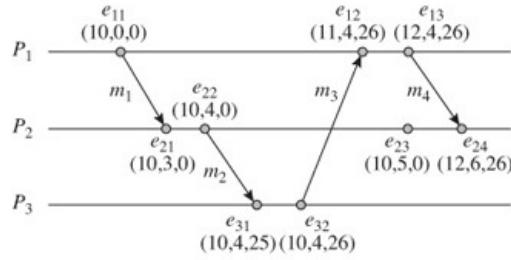
- $e_i$  precede  $e_j$ : per ogni  $l$ :  $vts(e_i)[l] \leq vts(e_j)[l]$ , ma esiste  $k$  tale che  $vts(e_i)[k] \neq vts(e_j)[k]$ ;
- $e_i$  segue  $e_j$ : per ogni  $l$ :  $vts(e_i)[l] \geq vts(e_j)[l]$ , ma esiste  $k$  tale che  $vts(e_i)[k] \neq vts(e_j)[k]$ ;
- $e_i, e_j$  sono concorrenti: esistono  $k, l$  tali che  $vts(e_i)[k] < vts(e_j)[k]$  e  $vts(e_i)[l] > vts(e_j)[l]$ .

L'assegnazione di timestamp tramite vettori di clock è contraddistinta da due importanti proprietà: ogni evento ha un unico timestamp derivante dalle regole R3 ed R4, e  $vts(e_i) < vts(e_j)$  se e solo se  $e_i \rightarrow e_j$ . Tali proprietà sono illustrate nel prossimo esempio.

#### Esempio 17.3 - Sincronizzazione dei vettori di clock

La Figura 17.4 mostra la sincronizzazione dei vettori di clock per il sistema di Figura 17.3. Alla ricezione del messaggio  $m_1$ ,  $VC_2[2]$  viene incrementato di una unità e  $VC_2[1]$  viene aggiornato al valore 10. Analogamente, quando il processo  $P_3$  riceve il messaggio  $m_2$ ,  $VC_3[3]$  viene incrementato di una unità mentre  $VC_3[1]$  e  $VC_3[2]$  sono aggiornati di

conseguenza. Gli eventi  $e_{32}$  ed  $e_{23}$  sono concorrenti, poiché  $vts(e_{32})[2] < vts(e_{23})[2]$  e  $vts(e_{23})[3] > vts(e_{32})[3]$ .



**Figura 17.4** Sincronizzazione dei vettori di clock.

Il fatto che  $vts(e_i) < vts(e_j)$  se e solo se  $e_i \rightarrow e_j$  implica che i vettori di clock, da soli, non forniscono un ordinamento totale degli eventi. Tuttavia, è possibile ottenere un ordine totale usando come timestamp, per ogni evento  $e_i$ , una coppia di valori  $pvts(e_i) \equiv (\text{tempo locale, id del processo})$  e la seguente relazione di ordinamento:

$$e_i \text{ precede } e_j \text{ se } \begin{cases} (i) \text{ } pvts(e_i).\text{tempo locale} < pvts(e_j).\text{tempo locale}, \text{ o} \\ (ii) \text{ } e_i \text{ ed } e_j \text{ sono eventi concorrenti e} \\ \text{ } pvts(e_i).\text{id del processo} < pvts(e_j).\text{id del processo} \end{cases} \quad (17.2)$$

dove  $pvts(e_i).\text{tempo locale}$  e  $pvts(e_i).\text{id del processo}$  sono, rispettivamente, il tempo locale e l'id del processo in  $pvts(e_i)$ .

## 17.4 Registrazione dello stato di un sistema distribuito

Come anticipato nel Paragrafo 17.2.1, lo stato globale di un sistema distribuito all'istante di tempo  $t$  è costituito dall'insieme degli stati locali di tutte le entità del sistema nello stesso istante di tempo  $t$ . Tuttavia, non è possibile fare in modo che tutti i nodi registrino i loro stati nello stesso istante poiché i clock locali non sono perfettamente sincronizzati. Si consideri il sistema distribuito mostrato in [Figura 17.5](#). Un'applicazione bancaria è costituita dal processo  $P_1$  nel nodo  $N_1$  e dal processo  $P_2$  nel nodo  $N_2$ , i quali eseguono le seguenti operazioni:



**Figura 17.5** Un sistema per il trasferimento di fondi.

1.  $P_1$  addebita \$100 sul conto A;
2.  $P_1$  invia un messaggio a  $P_2$  per accreditare \$100 sul conto B;
3.  $P_2$  accredita \$100 sul conto B.

Gli stati registrati dai nodi  $N_1$  e  $N_2$  sarebbero incosistenti se il saldo del conto A venisse registrato prima del passo 1 e il saldo del conto B venisse registrato prima del passo 3. Di conseguenza, un sistema operativo distribuito non può usare tale stato per eseguire le sue funzioni di controllo.

In questo paragrafo descriveremo un algoritmo che permette di registrare un insieme di stati locali consistenti. Tale insieme non è un vero e proprio sostituto dello stato globale, tuttavia, esso possiede proprietà che facilitano alcune delle funzioni di controllo effettuate da un sistema operativo distribuito.

### Registrazione di uno stato consistente

L'insieme degli stati locali di tutte le entità di un sistema distribuito, calcolato tramite un qualunque algoritmo, è detto *state recording*. Uno *state recording consistente* è un insieme di stati locali in cui gli stati di ogni coppia di processi del sistema sono consistenti in base alla Definizione 17.1.

**Definizione 17.1 Stati locali mutuamente consistenti** Gli stati locali dei processi  $P_k$  e  $P_l$  sono mutuamente consistenti se:

1. ogni messaggio registrato come "ricevuto da  $P_l$ " nello stato di  $P_k$ , viene registrato come "invia a  $P_k$ " nello stato di  $P_l$ , e
2. ogni messaggio registrato come "ricevuto da  $P_k$ " nello stato di  $P_l$ , viene registrato come "invia a  $P_l$ " nello stato di  $P_k$ .

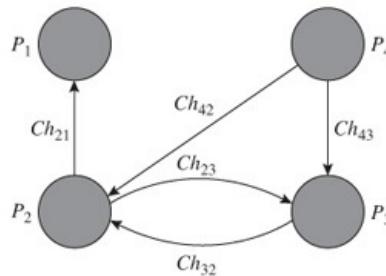
Nello state recording citato all'inizio di questo paragrafo, lo stato di  $P_2$  ha registrato che  $P_2$  ha ricevuto da  $P_1$  il messaggio relativo all'accordo di \$100 sul conto B, ma lo stato di  $P_1$  non ha registrato che  $P_1$  ha inviato tale messaggio. Quindi, lo state recording è incosistente. Un qualunque state recording che fosse contraddistinto da una delle seguenti situazioni sarebbe invece consistente:

1. il conto A contiene \$900 e il conto B contiene \$300;
2. il conto A contiene \$800 e il conto B contiene \$400;
3. il conto A contiene \$800 e il conto B contiene \$300.

Nel caso 1, nessuno dei processi ha ricevuto un messaggio prima che il proprio stato fosse registrato, pertanto gli stati dei processi sono consistenti. Nel caso 2, il messaggio inviato da  $P_1$  a  $P_2$  viene registrato come "ricevuto da  $P_1$ " nello stato di  $P_2$  e come "invia a  $P_2$ " nello stato di  $P_1$ . Nel caso 3, nessuno dei processi ha registrato la ricezione del messaggio. Esso è ancora in transito e verrà recapitato a  $P_2$  in qualche istante successivo, momento in cui  $P_2$  aggiungerà \$100 al conto B. Ciò spiega perché per garantire la mutua consistenza tra gli stati di due processi è richiesto che un messaggio registrato come spedito sia registrato anche come ricevuto, mentre il viceversa non è necessariamente richiesto.

#### 17.4.1 Proprietà di uno state recording consistente

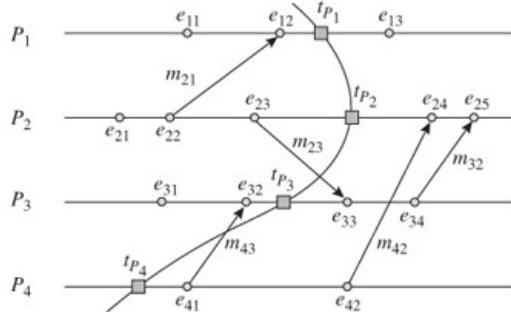
La [Figura 17.6](#) mostra il modello di un'elaborazione distribuita. L'elaborazione è costituita da quattro processi  $P_1 - P_4$  che comunicano tra loro attraverso scambio di messaggi. Un arco  $(P_i, P_j)$  rappresenta una *canale*  $Ch_{ij}$ , ovvero un cammino di comunicazione usato dal processo  $P_i$  per inviare messaggi al processo  $P_j$ . Si osservi che i canali sono unidirezionali - un processo può inviare messaggi su un canale oppure ricevere messaggi da un canale, ma non può fare entrambe le operazioni sullo stesso canale. I canali  $Ch_{23}$  e  $Ch_{32}$  permettono ai processi  $P_2$  e  $P_3$  di scambiarsi messaggi vicendevolmente.



**Figura 17.6** Esempio di elaborazione distribuita.

La [Figura 17.7](#) mostra il diagramma temporale dell'elaborazione. La [Tabella 17.2](#)

riporta invece gli stati dei processi  $P_1 - P_4$  registrati negli istanti di tempo  $t_{p1}, t_{p2}, t_{p3}$  e  $t_{p4}$ . Tali istanti di tempo sono evidenziati nella figura con il simbolo . Lo stato del processo  $P_1$  mostra che  $P_1$  ha ricevuto il messaggio  $m_{21}$ , ma che non ha ancora inviato alcun messaggio, mentre lo stato del processo  $P_2$  mostra che  $P_2$  ha inviato i messaggi  $m_{21}$  e  $m_{23}$  prima di  $t_{p2}$ , ma non ha ricevuto nessun messaggio. Questi stati sono mutuamente consistenti in base alla Definizione 17.1. Tuttavia, gli stati di  $P_3$  e  $P_4$  non sono mutuamente consistenti poiché lo stato del processo  $P_3$  ha registrato la ricezione del messaggio  $m_{43}$ , ma lo stato del processo  $P_4$  non ha registrato la spedizione dello stesso messaggio. Quindi lo state recording riportato nella [Tabella 17.2](#) non è consistente.



**Figura 17.7** Diagramma temporale per l'elaborazione distribuita di [Figura 17.6](#).

| Processi | Descrizione degli stati registrati                               |
|----------|------------------------------------------------------------------|
| $P_1$    | Nessun messaggio inviato. Messaggio $m_{21}$ ricevuto.           |
| $P_2$    | Messaggi $m_{21}$ e $m_{23}$ inviati. Nessun messaggio ricevuto. |
| $P_3$    | Nessun messaggio inviato. Messaggio $m_{43}$ ricevuto.           |
| $P_4$    | Nessun messaggio inviato. Nessun messaggio ricevuto.             |

**Tabella 17.2** Stati locali dei processi.

### Taglio di un sistema

La nozione di **taglio** di un sistema aiuta a determinare la consistenza di uno state recording. Supponiamo che  $t_{pi}$  sia l'istante in cui viene registrato lo stato del processo  $P_i$ .

**Definizione 17.2 Taglio di un sistema** Il taglio di un sistema è dato dalla curva che connette i punti di un diagramma temporale corrispondenti agli istanti di tempo in cui vengono registrati gli stati dei processi, seguendo un ordine crescente in base al numero di ogni processo.

Il taglio dell'elaborazione distribuita riportata in [Figura 17.7](#) corrisponde allo state recording descritto in [Tabella 17.2](#). La terminologia "effettuare un taglio" viene usata per indicare la registrazione dell'insieme di stati locali di un sistema. Un evento verificatosi in un processo prima della registrazione del suo stato si trova alla sinistra del taglio nel diagramma temporale. Tale evento appartiene al *passato del taglio*. Un evento che invece si verificasse in un processo dopo la registrazione del suo stato si troverà alla destra del taglio nel diagramma temporale. Tale evento appartiene quindi al *futuro del taglio*. Un taglio rappresenta uno state recording consistente di un sistema se gli stati di ogni coppia di processi soddisfano la Definizione 17.1.

### Stato di un canale

Lo stato di un canale  $Ch_{ij}$  è rappresentato dall'insieme dei messaggi contenuti in  $Ch_{ij}$ , ovvero dai messaggi inviati dal processo  $P_i$  che non sono ancora stati ricevuti dal

processo  $P_j$ . Useremo la seguente notazione per indicare lo stato di  $Ch_{ij}$ :

|                                   |                                                                                             |
|-----------------------------------|---------------------------------------------------------------------------------------------|
| <i>Recorded_sent<sub>ij</sub></i> | l'insieme dei messaggi registrati nello stato di $P_i$ come inviati sul canale $Ch_{ij}$    |
| <i>Recorded_recd<sub>ij</sub></i> | l'insieme dei messaggi registrati nello stato di $P_j$ come ricevuti sul canale $Ch_{ij}$ . |

Se  $Recorded\_sent_{ij} = Recorded\_recd_{ij}$ , allora tutti i messaggi inviati da  $P_i$  sono stati ricevuti da  $P_j$ . Quindi il canale è vuoto. Se  $Recorded\_sent_{ij} - Recorded\_recd_{ij} \neq \phi$ , dove “-” rappresenta l'operatore che esegue la differenza tra due insiemi, allora alcuni messaggi inviati da  $P_i$  non sono ancora stati ricevuti da  $P_j$ . Tali messaggi sono ancora in transito sul canale  $Ch_{ij}$ . Se  $Recorded\_recd_{ij} - Recorded\_sent_{ij} \neq \phi$ , allora  $P_j$  ha registrato come ricevuto almeno un messaggio che non è ancora stato registrato come inviato da  $P_i$ . In base alla Definizione 17.1, tale situazione indica che gli stati locali di  $P_i$  e  $P_j$  non sono consistenti.

In alcuni casi, un taglio in un diagramma temporale può intersecare un messaggio  $m_k$  inviato dal processo  $P_i$  al processo  $P_j$  tramite il canale  $Ch_{ij}$ . Il tipo di taglio indica se gli stati registrati per  $P_i$  e  $P_j$  sono consistenti rispetto all'invio e alla ricezione del messaggio. Inoltre, esso indica lo stato del canale. Possiamo distinguere tre casi:

- *nessuna intersezione tra taglio e messaggio*: gli eventi corrispondenti all'invio e alla ricezione del messaggio sono ambedue alla sinistra o alla destra del taglio. In entrambi i casi, il messaggio non era presente nel canale  $Ch_{ij}$  al momento del taglio;
- *intersezione anteriore*, l'evento corrispondente all'invio del messaggio è alla sinistra del taglio mentre l'evento corrispondente alla ricezione del messaggio è alla destra del taglio. In tal caso, il messaggio era ancora nel canale  $Ch_{ij}$  quando il taglio è stato effettuato. Il taglio nel diagramma temporale di [Figura 17.7](#) ha una intersezione anteriore con il messaggio  $m_{23}$ ;
- *intersezione posteriore*: l'evento corrispondente all'invio del messaggio è alla destra del taglio mentre l'evento corrispondente alla ricezione del messaggio è alla sinistra del taglio. In tal caso, al momento del taglio, il messaggio era già stato ricevuto, ma non era ancora stato inviato. Un tale messaggio evidenzia una inconsistenza tra gli stati registrati. Il taglio nel diagramma temporale di [Figura 17.7](#) ha una intersezione posteriore con il messaggio  $m_{43}$ .

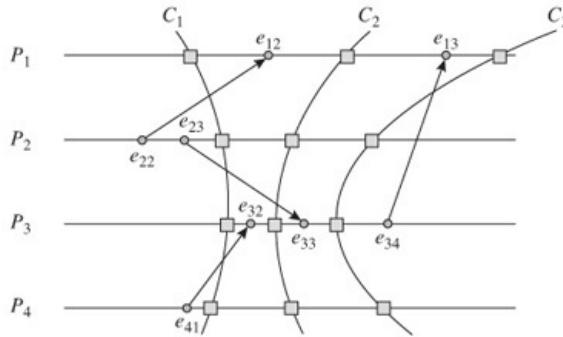
Dalle precedenti osservazioni, possiamo formulare la seguente condizione di consistenza per un taglio:

CC un taglio  $C$  rappresenta uno state recording consistente di un sistema distribuito se il futuro del taglio è chiuso rispetto alla relazione di precedenza tra eventi, ovvero se è chiuso rispetto a “ $\rightarrow$ ”.

La condizione CC può essere utilizzata come segue: un insieme di oggetti  $I$  è detto *chiuso rispetto alla relazione R*, se usando la relazione su ogni oggetto di  $I$  otteniamo ancora un oggetto contenuto in  $I$ . Supponiamo quindi che  $I$  sia l'insieme degli eventi appartenenti al futuro di un taglio. Applicando la relazione “ $\rightarrow$ ” a un evento  $e_i \in I$  otteniamo un evento  $e_j$  tale che  $e_i \rightarrow e_j$ , ovvero  $e_i$  precede  $e_j$ . Se  $I$  è chiuso rispetto alla relazione “-”, allora anche l'evento  $e_j$  appartiene a  $I$ . Ciò significa che  $e_j$  non appartiene al passato del taglio. Questa condizione equivale a imporre una restrizione al taglio in modo che esso non abbia intersezioni posteriori con alcun messaggio. Al contrario, una intersezione anteriore non viola la condizione CC, in quanto l'invio del messaggio apparterrebbe al passato del taglio mentre la ricezione avverrebbe nel futuro.

#### Esempio 17.4 - Consistenza di un taglio

Nella [Figura 17.8](#), i tagli  $C_1$  e  $C_2$  sono consistenti poiché non vi sono eventi appartenenti al passato dei tagli che seguono eventi appartenenti al loro futuro. Il taglio  $C_3$ , invece, è inconsistente poiché pur essendo l'evento  $e_{13}$  successivo all'evento  $e_{34}$  (in base alla relazione di causa-effetto esistente tra l'invio e la ricezione di un messaggio),  $e_{34}$  appartiene al futuro di  $C_3$  mentre  $e_{13}$  appartiene al suo passato.



**Figura 17.8** Consistenza dei tagli - I tagli  $C_1, C_2$  sono consistenti mentre  $C_3$  è inconsistente.

### 17.4.2 Un algoritmo per la registrazione di state recording consistenti

Questo paragrafo descrive un algoritmo definito da Chandy e Lamport (1985) per la determinazione di uno state recording consistente. L'algoritmo si basa sulle seguenti assunzioni:

1. i canali sono unidirezionali;
2. i canali hanno capacità illimitata;
3. i canali sono gestiti in modo FIFO.

L'assunzione relativa alla gestione FIFO dei canali implica che i messaggi vengano ricevuti dal processo destinatario nello stesso ordine in cui sono stati inviati dal processo mittente. Per iniziare la registrazione di uno state recording, un processo iniziatore registra il suo stato e invia un messaggio speciale, chiamato *marker*, su ogni suo canale di uscita richiedendo la determinazione dello state recording ai processi collegati. Quando un processo riceve il marker, esso registra lo stato del canale su cui ha ricevuto il marker. Alla prima ricezione del marker, il processo registra anche il suo stato e invia il marker su ogni suo canale di uscita. Useremo la seguente notazione per definire come viene determinato lo stato di un canale:

- |                       |                                                                                                                    |
|-----------------------|--------------------------------------------------------------------------------------------------------------------|
| $Received_{ij}$       | l'insieme dei messaggi ricevuti dal processo $P_j$ sul canale $Ch_{ij}$ prima di ricevere il marker su $Ch_{ij}$ ; |
| $Recorded\_recd_{ij}$ | l'insieme dei messaggi registrati sullo stato di $P_j$ come ricevuti sul canale $Ch_{ij}$ .                        |

Le regole dell'Algoritmo 17.2 sono eseguite atomicamente, ovvero, come operazioni indivisibili. L'operazione effettuata dall'algoritmo per registrare lo stato di un canale può essere spiegata come segue: supponiamo che il processo  $P_i$  invii i messaggi  $m_i, m_{i2}, \dots, m_{in}$  sul canale  $Ch_{ij}$  prima di registrare il suo stato e prima di inviare il marker su  $Ch_{ij}$ . Supponiamo, inoltre, che il processo  $P_j$  abbia due canali di ingresso  $Ch_{ij}$  e  $Ch_{kj}$ . Se il marker che arriva sul canale  $Ch_{ij}$  è il primo marker ricevuto da  $P_j$ , il processo registra nel suo stato l'insieme dei messaggi ricevuti, ovvero  $Recorded\_recd_{ij}$  e  $Recorded\_recd_{kj}$ . Inoltre,  $P_j$  registra lo stato del canale  $Ch_{ij}$  come vuoto, in quanto, essendo i canali gestiti in modo FIFO, il processo  $P_j$  deve necessariamente aver ricevuto il marker su  $Ch_{ij}$  dopo la ricezione dei messaggi  $m_{i1}, m_{i2}, \dots, m_{in}$ .

#### Algoritmo 17.2 Algoritmo di Chandy-Lamport

1. Quando un processo  $P_i$  inizia a registrare lo stato:  $P_i$  registra il suo stato e invia un marker su ogni suo canale di uscita.
2. Quando il processo  $P_j$  riceve un marker su un canale di ingresso  $Ch_{ij}$ :  $P_j$  esegue le seguenti operazioni:

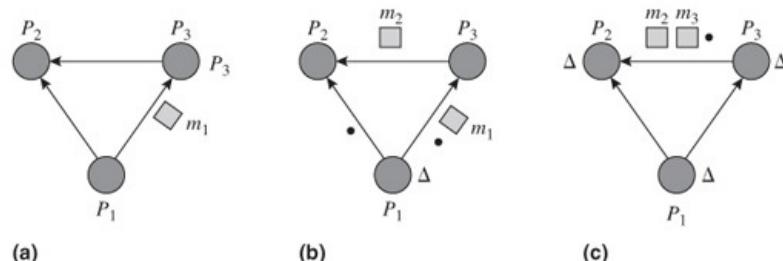
- a. se  $P_j$  non ha ancora ricevuto nessun marker, allora
  - i. registra il suo stato;
  - ii. registra lo stato del canale  $Ch_{ij}$  come vuoto;
  - iii. invia un marker su ogni suo canale di uscita;
- b. altrimenti, registra lo stato del canale  $Ch_{ij}$  come l'insieme dei messaggi appartenenti a  $Received_{ij} - Recorded\_recd_{ij}$ .

Supponiamo ora che  $P_j$  riceva due ulteriori messaggi  $m_{k1}$  e  $m_{k2}$  su  $Ch_{kj}$  prima di ricevere il marker su  $Ch_{kj}$ . In tal caso,  $Received_{kj} = Recorded\_recd_{kj} \cup \{m_{k1}, m_{k2}\}$ , e lo stato del canale  $Ch_{kj}$  viene registrato come l'insieme dei messaggi  $Received_{kj} - Recorded\_recd_{kj}$  ovvero,  $\{m_{k1}, m_{k2}\}$ . Infatti, il processo  $P_k$  deve necessariamente aver inviato i messaggi  $m_{k1}, m_{k2}$  prima di registrare il suo stato e prima di inviare il marker sul canale  $Ch_{kj}$ , pertanto se questi messaggi non fossero stati ricevuti da  $P_j$  prima che quest'ultimo registrasse il suo stato, essi avrebbero dovuto trovarsi nel canale.

L'Esempio 17.5 illustra l'esecuzione dell'algoritmo di Chandy-Lamport.

#### Esempio 17.5 - Esecuzione dell'algoritmo di Chandy-Lamport

La Figura 17.9(a) mostra un sistema distribuito al tempo 0. Il processo  $P_1$  ha inviato il messaggio  $m_1$  al processo  $P_3$ . Il messaggio è attualmente sul canale  $Ch_{13}$ . Al tempo 1, il processo  $P_3$  invia il messaggio  $m_2$  al processo  $P_2$ . Al tempo 2,  $P_1$  decide di registrare lo stato del sistema, quindi esso registra il suo stato e invia il marker sui suoi canali di uscita.



**Figura 17.9** Esempio di esecuzione dell'algoritmo di Chandy-Lamport: sistema al tempo 0, 2+, e 5+.

La Figura 17.9(b) mostra la situazione al tempo  $2^+$ . Il messaggio  $m_1$  è ancora nel canale  $Ch_{13}$  e il messaggio  $m_2$  è nel canale  $Ch_{32}$ . I pallini indicano i marker. Il simbolo  $\Delta$  indica che lo stato di un processo è già stato registrato. Il processo  $P_2$  riceve il marker su  $Ch_{12}$  al tempo 3, registra il suo stato, e registra lo stato del canale  $Ch_{12}$  come vuoto. Il processo  $P_3$  invia il messaggio  $m_3$  al processo  $P_2$  al tempo 4 e riceve il marker su  $Ch_3$  al tempo 5.  $P_3$  quindi registra il suo stato, registra lo stato del canale  $Ch_{13}$  come vuoto, e invia il marker su  $Ch_{32}$ . La Figura 17.9(c) mostra la situazione al tempo  $5^+$ . Gli stati di tutti i processi sono stati registrati, così come gli stati dei canali  $Ch_{12}$  e  $Ch_{13}$ . Al contrario, lo stato di  $Ch_{32}$  non è ancora stato registrato.

Quando il marker che si trova su  $Ch_{32}$  raggiunge il processo  $P_2$ , quest'ultimo registra lo stato di  $Ch_{32}$  in base al passo 2(b) dell'Algoritmo 17.2. Lo stato di  $Ch_{32}$  sarà quindi  $\{m_2, m_3\}$  poiché questi messaggi sono contenuti in  $Received_{32}$  ma non in  $Recorded\_recd_{32}$ . La Tabella 17.3 mostra lo state recording del sistema.

| Entità    | Descrizione dello stato registrato                        |
|-----------|-----------------------------------------------------------|
| $P_1$     | Messaggio $m_1$ inviato. Nessun messaggio ricevuto.       |
| $P_2$     | Nessun messaggio inviato né ricevuto.                     |
| $P_3$     | Messaggi $m_2$ e $m_3$ inviati. Messaggio $m_1$ ricevuto. |
| $Ch_{12}$ | Vuoto.                                                    |
| $Ch_{13}$ | Vuoto.                                                    |
| $Ch_{23}$ | Contiene i messaggi $\{m_2, m_3\}$ .                      |

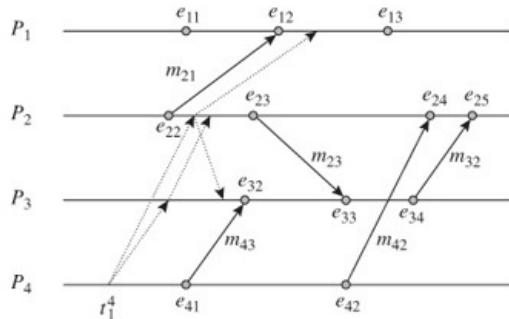
**Tabella 17.3** Recording dei processi e dei canali di [Figura 17.9](#).

#### Proprietà di uno state recording

Supponiamo che  $t_b$  e  $t_e$  siano gli istanti di tempo in cui inizia e finisce la determinazione dello stato di un sistema  $S$ . Supponiamo inoltre che  $RS$  sia lo stato finale registrato in  $S$ . Sembrerebbe ragionevole pensare che  $S$  si debba trovare nello stato  $RS$  in qualche istante di tempo  $t_i$  tale che  $t_b < t_i < t_e$ . Invece, ciò può anche non essere vero! Infatti, può succedere che lo stato  $RS$  non sia uguale a nessuno stato globale del sistema. Tale eventualità è evidenziata nell'Esempio 17.6.

#### Esempio 17.6 - Confronto tra stato registrato e stato globale

La [Figura 17.7](#) mostra un diagramma temporale del sistema distribuito di [Figura 17.6](#). Supponiamo che  $P_4$  inizi l'esecuzione dell'algoritmo di Chandy-Lamport al tempo  $t_1$ . Il diagramma temporale di [Figura 17.10](#) mostra come vengono inviati e ricevuti i marker durante la registrazione dello stato. I marker sono rappresentati con frecce tratteggiate.



**Figura 17.10** State recording del sistema delle [Figure 17.6](#) e [17.7](#).

La [Tabella 17.4](#) mostra lo stato dei canali e dei processi registrato in base all'algoritmo di Chandy-Lamport. Il messaggio  $m_{21}$  viene registrato come inviato da  $P_2$  e ricevuto da  $P_1$ . Nessun altro messaggio viene registrato. Tuttavia, poiché il diagramma temporale di [Figura 17.7](#) è riportato in scala, è evidente che il sistema non si trova mai in uno stato in cui solo il messaggio  $m_{21}$  è stato inviato e ricevuto - gli eventi  $e_{23}$ ,  $e_{32}$  ed  $e_{41}$ , corrispondenti all'invio e alla ricezione di alcuni messaggi, sono avvenuti prima dell'evento  $e_{12}$ , corrispondente alla ricezione del messaggio  $m_{21}$ . Pertanto, qualunque stato globale che abbia registrato la ricezione del messaggio  $m_{21}$  dovrebbe aver registrato anche l'invio e la ricezione del messaggio  $m_{43}$  e l'invio del messaggio  $m_{23}$ .

| Entità | Descrizione dello stato registrato                     |
|--------|--------------------------------------------------------|
| $P_1$  | Nessun messaggio inviato. Messaggio $m_{21}$ ricevuto. |
| $P_2$  | Messaggio $m_{21}$ inviato. Nessun messaggio ricevuto. |
| $P_3$  | Nessun messaggio inviato né ricevuto.                  |
| $p_4$  | Nessun messaggio inviato né ricevuto.                  |

\* Gli stati di tutti i canali sono stati registrati come vuoti.

**Tabella 17.4** Uno state recording che non corrisponde a nessuno stato globale.

Sebbene il sistema non si sia mai trovato nello stato registrato dall'algoritmo di Chandy-Lamport, quest'ultimo è utile per molte applicazioni che richiedono solamente la mutua consistenza degli stati locali. Si consideri, per esempio, di dover calcolare a quanto ammonta il fondo totale di un sistema bancario. In un simile contesto, quando vengono trasferiti \$100 dal conto A al conto B, è irrilevante che lo stato registrato mostri che tale quantità di denaro si trova nel conto A, oppure nel conto B, oppure nel canale che li collega, ciò che importa invece è che il sistema si trova in una di queste tre possibilità. Lo stato registrato dall'Algoritmo 17.2 soddisfa proprio questo requisito.

Chandy e Lamport (1985) hanno dimostrato che l'Algoritmo 17.2 può anche essere usato per provare se un sistema soddisfa proprietà stabili. Una proprietà è stabile se, divenuta vera in uno stato globale del sistema al tempo  $t_i$  continua a rimanere vera per tutti gli istanti di tempo  $t_j > t_i$ . È quindi possibile analizzare lo stato registrato periodicamente dall'Algoritmo 17.2 per identificare la presenza di proprietà stabili. Talvolta, l'analisi del primo stato registrato dopo l'istante di tempo  $t_i$ , potrebbe non riuscire a dimostrare la presenza di una proprietà stabile. Tuttavia, questa eventualità non rappresenta un grave problema, in quanto una proprietà stabile in un certo istante continua a esserlo anche in futuro, e quindi una successiva esecuzione dell'algoritmo sarà in grado di identifierla. Una proprietà stabile molto utile da rilevare è rappresentata dall'esistenza di un ciclo nel WFG o nel RRAG usato per determinare la presenza di deadlock in un sistema (Capitolo 8). Un'altra proprietà stabile di particolare interesse è la *condizione di terminazione distribuita* definita come la situazione in cui tutti i processi che partecipano all'elaborazione distribuita sono passivi e non vi sono messaggi in transito tra di essi (Paragrafo 18.6).

## Riepilogo

Il sistema operativo usa le nozioni di tempo e stato sia per controllare l'attività dei processi utente che per organizzare le sue funzionalità. Tuttavia, ogni nodo di un sistema distribuito ha il suo clock e la sua memoria privati, e quindi le nozioni di tempo e stato non possono essere utilizzate semplicemente come avviene in un sistema operativo convenzionale. In questo capitolo, abbiamo definito nozioni di tempo e stato alternative adatte a essere usate in un sistema distribuito.

Un sistema operativo usa la nozione di tempo per conoscere l'ordine in cui si verificano gli eventi, in modo che esso possa servirli usando meccanismi dipendenti dal tempo, come FCFS o LIFO. La nozione di *precedenza* tra eventi, che indica quale evento avviene prima dell'altro, è *transitiva*. Pertanto, se l'evento  $e_i$  si verifica prima dell'evento  $e_j$  ed  $e_j$  prima dell'evento  $e_k$ , allora anche  $e_i$  si verifica prima di  $e_k$ . Il sistema operativo può determinare la precedenza tra gli eventi come segue: Se due eventi occorrono nello stesso nodo, il sistema operativo conosce quale occorre prima. Per eventi che avvengono in nodi differenti, il sistema operativo usa la proprietà di transitività tra gli eventi e la nozione di *relazione casuale* tra eventi - ovvero la relazione di causa-effetto - per capire quale evento si verifica per primo. Per esempio, nel caso di comunicazione inter-processo basata su messaggi, l'invio di un messaggio è la causa mentre la sua ricezione è l'effetto. Quindi, l'evento corrispondente alla ricezione di un messaggio si verifica *dopo* l'evento corrispondente al suo invio. Usando la transitività, ne consegue che un evento che precede l'invio di un messaggio accade prima di un evento che si verifica dopo la ricezione dello stesso messaggio. Tuttavia, per alcune coppie di eventi, non è possibile conoscere quale di essi avvenga prima. Tali eventi sono detti *eventi concorrenti*.

Dedurre la precedenza degli eventi usando la transitività risulta essere laborioso. Quindi, il sistema operativo associa un *timestamp*, cioè un numero legato allo scorrere del tempo, a ogni evento e confronta i timestamp di due eventi per conoscere quale si verifica prima. Per facilitare l'assegnazione dei timestamp agli eventi, il sistema operativo mantiene un *clock* per ogni processo, detto *clock locale* del processo, e cerca di tenere ben sincronizzati i *clock* locali di tutti i processi. La sincronizzazione dei *clock* viene effettuata sfruttando la relazione di causalità che regola lo scambio di messaggi tra i processi come segue: ogni messaggio contiene il timestamp associato all'evento corrispondente al suo invio. Il *clock* locale del processo ricevente il messaggio deve avere un valore più grande del timestamp contenuto nel messaggio. In caso contrario, il *clock* locale del processo ricevente viene incrementato in modo che il suo valore risulti maggiore del timestamp del messaggio. Poiché i *clock* locali non memorizzano il tempo reale, essi vengono chiamati *clock logici*. I timestamp che usano i *clock logici* hanno la seguente proprietà: se  $t_i$  e  $t_j$  sono i timestamp degli eventi  $e_i$  ed  $e_j$ , allora  $t_i < t_j$  se  $e_i$  precede  $e_j$ . Esiste anche un diverso sistema di sincronizzazione dei *clock*, chiamato *vettori di clock*, per il quale oltre a valere la proprietà precedente, vale anche la seguente: se  $t_i < t_j$  allora l'evento  $e_i$  si verifica prima dell'evento  $e_j$ . Nel caso di eventi concorrenti, non è importante come essi vengono ordinati. Per ottenere un ordine totale tra gli eventi è quindi necessario usare una coppia (*id del processo, timestamp*).

Lo stato di un nodo è detto *stato locale*. Lo *stato globale* di un sistema è un insieme di stati locali dei suoi nodi ottenuti nello stesso istante di tempo. Non è possibile registrare lo stato globale chiedendo a ogni nodo di registrare il suo stato locale in un preciso istante di tempo, poiché i *clock* dei nodi non sono perfettamente sincronizzati. Un insieme arbitrario di stati locali può essere inconsistente. Per esempio, se i nodi registrano i loro stati spontaneamente, il nodo  $N_i$  potrebbe registrare il suo stato locale prima di inviare un messaggio  $m$  al nodo  $N_j$  mentre il nodo  $N_j$  potrebbe registrare il suo stato locale dopo aver ricevuto il messaggio  $m$ . Questo problema è stato risolto da Chandy e Lamport (1985) usando un messaggio speciale, chiamato *marker* per forzare i nodi a registrare i loro stati locali. Essi mostrarono che se la comunicazione tra i processi è basata su un meccanismo FIFO, gli stati locali dei processi registrati usando il loro algoritmo sono mutuamente consistenti. L'insieme di tali stati locali può essere usato per identificare *proprietà stabili*, ovvero proprietà che non cambiano nel tempo, come per esempio la presenza di un ciclo in un grafo di attesa.

## Domande

- 17.1. Indicare se le seguenti affermazioni sono vere o false.
- Gli eventi  $e_i$  ed  $e_j$  sono concorrenti solo se  $ts(e_i) = ts(e_j)$ , dove  $ts(e_i)$ ,  $ts(e_j)$  sono i timestamp di  $e_i$  ed  $e_j$  usando *clock logici*.
  - Anche se  $ts(e_i) > ts(e_j)$ , l'evento  $e_i$  potrebbe essere avvenuto prima dell'evento  $e_j$ .
  - Anche se  $vts(e_i) > vts(e_j)$ , l'evento  $e_i$  potrebbe essere avvenuto prima dell'evento  $e_j$ .
  - Un messaggio inviato da  $P_i$  a  $P_j$  che interseca un taglio è stato registrato, negli stati dei processi  $P_i$  e  $P_j$ , come inviatoda  $P_i$  ma non ancora ricevuto da  $P_j$ .
  - In uno stato registrato secondo l'algoritmo di Chandy-Lamport, è probabile che lo stato di un canale  $Ch_{ij}$  sia non vuoto solo se il processo  $P_j$  riceve il marker su qualche altro canale prima di riceverlo su  $Ch_{ij}$ .
- 17.2. Selezionare l'alternativa appropriata per ciascuna delle seguenti domande.
- Se il processo  $P_i$  invia un messaggio al processo  $P_j$ , ma il processo  $P_j$  non invia un messaggio al processo  $P_i$ , gli stati di  $P_i$  e  $P_j$  sono mutuamente consistenti solo se:
    - tutti i messaggi inviati dal processo  $P_i$  al processo  $P_j$  sono stati ricevuti dal processo  $P_j$ ;
    - alcuni messaggi inviati da  $P_i$  a  $P_j$  non sono stati ricevuti da  $P_j$ ;
    - tutti i messaggi ricevuti da  $P_j$  da parte di  $P_i$  sono stati inviati da  $P_j$ ;
    - Nessuna delle alternative (i)-(iii).

- b. Se l'evento  $e_i$  nel processo  $P_i$  è nel passato di un taglio  $C_k$
- tutti gli eventi del sistema che precedono l'evento  $e_i$  sono nel passato del taglio  $C_k$ ;
  - alcuni degli eventi che precedono  $e_i$  possono essere nel passato del taglio  $C_k$ ;
  - tutti gli eventi che occorrono dopo  $e_i$  sono nel futuro del taglio  $C_k$ ;
  - nessuna delle alternative (i)-(iii).

## Problemi

17.1. Nell'Esempio 17.2, il tempo di  $P_3$  è maggiore di quello di  $P_1$  e  $P_2$ . Elencare tutte le condizioni incui ciò può accadere.

17.2. In un sistema costituito da tre processi si verificano i seguenti eventi:

| processo $P_1$            | processo $P_2$              | processo $P_3$              |
|---------------------------|-----------------------------|-----------------------------|
| evento $e_1$ ;            | evento $e_3$ ;              | evento $e_5$ ;              |
| –                         | –                           | –                           |
| Invia messaggio a $P_2$ ; | Riceve messaggio da $P_3$ ; | Invia messaggio a $P_2$ ;   |
| evento $e_2$ ;            | Riceve messaggio da $P_1$ ; | evento $e_6$ ;              |
| –                         | –                           | Riceve messaggio da $P_2$ ; |
| –                         | evento $e_4$ ;              | –                           |
| –                         | Invia messaggio a $P_3$ ;   | evento $e_7$ ;              |

- disegnare il diagramma temporale del sistema;
  - mostrare le precedenze tra gli eventi del sistema;
  - elencare gli eventi concorrenti.
- 17.3.  $Synch(P_i, P_j, t_k) = \text{vero}$  se i clock logici di  $P_i$  e  $P_j$  sono ragionevolmente consistenti all'istante di tempo  $t_k$ ; ovvero, se la differenza tra i loro valori è  $< \delta$ , con  $\delta$  piccolo. Se  $RP(i, k)$  è l'insieme di processi da cui  $P_i$  ha ricevuto un messaggio prima di  $t_k$  e  $SP(i, k)$  è l'insieme di processi a cui  $P_i$  ha inviato messaggi prima di  $t_k$ , determinare se  $Synch(P_i, P_j, t_k)$  sarebbe *vero* nelle seguenti situazioni.
- $RP(i, k) \cap RP(j, k) \neq \emptyset$
  - Esiste  $P_g \in SP(i, k)$  tale che  $P_j \in SP(g, k)$
  - $P_j \in SP(i, k)$
  - $P_j \in RP(i, k)$  ma  $P_i \notin RP(i, k - 1)$
  - $P_j \in SP(i, k)$  e  $P_i \in SP(j, k)$
  - $P_j \in RP(i, k)$  ma  $P_j \notin RP(i, k - 1)$  e  $P_i$  non ha ricevuto messaggi da nessun processo dopo l'istante in cui ha inviato un messaggio a  $P_j$ .
- 17.4. La relazione (17.1) impone un ordine totale anche se gli eventi possono essere solo parzialmente ordinati usando la relazione di causalità. Fornire un esempio di un sistema che mostri tali eventi. Commentare vantaggi e svantaggi derivanti dall'uso della relazione (17.1).
- 17.5. Invece di usare la relazione (17.2) per ottenere un ordine totale per mezzo dei vettori di timestamp, si immagini di usare la relazione seguente:  $e_i$  precede  $e_j$  se:
- $\text{ppts}(e_i). \text{tempo locale} < \text{ppts}(e_j). \text{tempo locale}$ , o
  - $\text{ppts}(e_i). \text{tempo locale} = \text{ppts}(e_j). \text{tempo locale}$  e

$pvts(e_i). id \text{ del processo} < pvts(e_j). id \text{ del processo}$

Commentare la correttezza della precedente relazione.

- 17.6. Siano  $t_i$  e  $t_j$  i timestamp degli eventi  $e_i$  ed  $e_j$ .
  - a. Fornire un esempio di un sistema in cui  $t_i < t_j$  quando vengono usati i clock logici, ma  $t_i \not\prec t_j$  quando vengono usati i vettori di clock.
  - b. Se  $t_i < t_j$  usando i vettori di clock, mostrare che  $t_i < t_j$  quando vengono usati i clock logici.
  - c. Se  $t_i < t_j$  usando i clock logici, mostrare che  $t_i \not\prec t_j$  quando vengono usati i vettori di clock.
- 17.7. I vettori di clock di due eventi concorrenti  $e_i$  ed  $e_j$  sono tali che  $vts(e_i)[k] < vts(e_j)[k]$ . Dimostrare che gli eventi  $e_i$  ed  $e_j$  sono concorrenti se  $vts(e_i)[g] = vts(e_j)[g]$  per ogni  $g \neq k$  e  $vts(e_i)[k] > vts(e_j)[k]$ .
- 17.8. Spiegare, con l'aiuto di un esempio, perché l'algoritmo di Chandy-Lamport richiede che i canali siano di tipo FIFO.
- 17.9. Uno stato di un sistema è transitless se non ci sono messaggi in transito. (Si veda la [Tabella 17.4](#) per un esempio.) Fornire un esempio di un sistema in cui tutti gli stati registrati usando l'algoritmo di Chandy-Lamport sono necessariamente transitless.
- 17.10. Un sistema è costituito dai processi  $P_i$ ,  $P_j$  e dai canali  $Ch_{ij}$  e  $Ch_{ji}$ . Ogni processo invia un messaggio all'altro processo a intervalli regolari di  $\delta$  secondi. Ogni messaggio richiede  $\sigma$  secondi per raggiungere  $P_j$ . Dimostrare che se  $\delta < \sigma$ , la registrazione dello stato iniziata da  $P_j$  usando l'algoritmo di Chandy-Lamport non può essere transitless.
- 17.11. Fornire un esempio di un sistema in cui lo stato registrato dall'algoritmo di Chandy-Lamport è uno degli stati in cui il sistema si trova effettivamente durante l'esecuzione dell'algoritmo.
- 17.12. Quale sarà lo state recording nell'E-sempio 17.6, se la richiesta di eseguire uno state recording nel canale  $Ch_{42}$  viene ritardata e inviata al processo  $P_2$  immediatamente dopo l'occorrenza dell'evento  $e_{23}$ ?
- 17.13. L'algoritmo di Chandy-Lamport funziona correttamente anche se più nodi iniziano spontaneamente a registrare lo stato del sistema. Descrivere il funzionamento dell'algoritmo se i processi  $P_2$  e  $P_4$  di [Figura 17.6](#) iniziano a registrare lo stato (a) prima di inviare qualunque messaggio, (b) dopo avere inviato un solo messaggio su ognuno dei canali  $Ch_{21}$ ,  $Ch_{32}$  e  $Ch_{43}$  e senza che successivamente venga inviato nessun altro messaggio.
- 17.14. L'assunzione effettuata nell'Algoritmo 17.2, in base alla quale i canali devono essere di tipo FIFO, può essere rimossa inserendo un campo aggiuntivo in ogni messaggio. Tale campo conterrà i valori *before token* oppure *after token* a seconda che il messaggio sia stato inviato prima o dopo l'invio di un token nello stesso canale. Se un processo riceve un messaggio con il campo *after token* prima di ricevere un token nello stesso canale, esso effettua la stessa azione che farebbe alla ricezione del token, e ignora il successivo token che arriva. Formulare un insieme di regole per registrare lo stato di un canale usando lo schema precedente.

## Note bibliografiche

Lamport (1978) ha presentato un modo per ordinare gli eventi di un sistema distribuito definendo un ordine parziale. Mattern (1989), Garg (2002), Attiya e Welch (2004) hanno presentato i concetti di vettore di clock e di taglio consistente. La consistenza dei tagli è stata discussa anche da Chandy, Lamport (1985) e da Knapp (1987).

Chandy e Lamport (1985) hanno sviluppato l'algoritmo dei distributed snapshot descritto nel Paragrafo 17.2 che permette di registrare lo stato di un sistema in cui i processi comunicano tramite canali FIFO. Li, Radhakrishnan, Venkatesh (1987), Lai, Yang (1987), e Mattern (1989) hanno invece proposto algoritmi che non richiedono la presenza di canali FIFO. Anche Lynch (1996) e Tel (2000) hanno proposto alcuni algoritmi per determinare snapshot globali.

1. Attiya, H., and J. Welch (2004): *Distributed Computing: Fundamentals, Simulations*

*and Advanced Topics*, John Wiley, New York.

2. Chandy K.M., and L. Lamport (1985): "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, **3**, 1 (Feb. 1985), 63-75.
3. Garg, V.K. (2002): *Elements of Distributed Computing*, Wiley-IEEE, New York.
4. Knapp, E. (1987): "Distributed deadlock Detection," *Computing Surveys*, **19**, 4 (Dee. 1987), 303-328.
5. Lai, T.H., and T.H. Yang (1987): "On distributed snapshots," *Information Processing Letters*, **25**, 153-158.
6. Lamport L. (1978): "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, **21**, 7 (July 1978), 558-565.
7. Li, H.F., T. Radhakrishnan, and K. Venkatesh (1987): "Global state detection in non-FIFO networks," *Proceedings of the 7th International Conference on Distributed Computing Systems*, 364-370.
8. Lynch, N. (1996): *Distributed Algorithms*, Morgan Kaufmann.
9. Mattern, F. (1989): "Virtual time and global states of distributed systems," M. Cosnard et al. (eds.), *Parallel and Distributed Algorithms*, Elsevier Science, North Holland.
10. Spezialetti, M., and P. Kearns (1986): "Efficient distributed snapshots," *Proceedings of the 6th International Conference on Distributed Computing Systems*, 382-388.
11. Tel, G. (2000): *Introduction to Distributed Algorithms*, 2nd ed. Cambridge University Press, Cambridge.

---

# CAPITOLO 18

## Algoritmi di controllo distribuiti

---

### Obiettivi di apprendimento

- Concetto di algoritmo di controllo distribuito
- Correttezza di un algoritmo di controllo distribuito
- Mutua esclusione distribuita
- Gestione dei deadlock distribuita
- Schedulazione distribuita
- Rilevazione della terminazione distribuita
- Algoritmi di elezione

Un sistema operativo distribuito esegue numerose operazioni di controllo. Alcune di esse, quali le operazioni relative alla *gestione della mutua esclusione* e alla *gestione dei deadlock*, sono simili a quelle effettuate in un sistema operativo convenzionale. Altre sono invece specifiche per i sistemi distribuiti. Tra queste possiamo ricordare la gestione della *schedulazione dei processi*, che esegue operazioni per *bilanciare il carico* su tutti i nodi del sistema in modo equo; gli algoritmi di *elezione* usati per eleggere i processi coordinatori che dovranno gestire attività condivise tra gruppi di processi; l'algoritmo per il *controllo della terminazione* necessario per verificare se tutti i processi appartenenti a una computazione distribuita, in esecuzione su nodi differenti del sistema, hanno completato le loro attività.

Per rispondere velocemente e in modo affidabile agli eventi che si verificano nel sistema, le azioni effettuate dagli algoritmi di controllo di un sistema operativo distribuito vengono eseguite su diversi nodi del sistema. Si parla pertanto di *algoritmi di controllo distribuiti*. Essi non hanno bisogno di conoscere lo stato globale del sistema, bensì dipendono dagli stati locali dei differenti nodi, e utilizzano meccanismi basati sullo scambio di messaggi tra processi per conoscere lo stato dei nodi e conseguentemente prendere le decisioni necessarie. La correttezza di tali algoritmi dipende, quindi, da come essi usano sia le informazioni relative agli stati locali che la comunicazione inter-processo. In particolare, possiamo dire che un algoritmo di controllo è corretto se è in grado di garantire due aspetti fondamentali: la *liveness*, ovvero il fatto che prima o poi l'algoritmo prenderà la giusta decisione, e la *safety*, ovvero il fatto che l'algoritmo non prenderà mai decisioni errate.

In questo capitolo, presenteremo gli algoritmi di controllo distribuiti che gestiscono le varie funzioni di un sistema operativo distribuito e descriveremo le loro proprietà in relazione all'effetto che hanno sulle prestazioni del sistema.

### 18.1 Funzioni degli algoritmo di controllo distribuiti

I sistemi operativi distribuiti implementano funzioni di controllo tramite *algoritmi di controllo distribuiti*. Pertanto, le attività eseguite e i dati elaborati da tali algoritmi sono distribuiti sui vari nodi del sistema. Un simile approccio, rispetto a una implementazione centralizzata, presenta i seguenti vantaggi:

- si evitano i ritardi e il sovraccarico dovuti all'acquisizione dello stato globale del sistema necessario nel caso di gestione centralizzata;
- la funzione di controllo può rispondere velocemente a eventi che occorrono su nodi differenti del sistema;
- il fallimento di un singolo nodo non influenza negativamente l'esecuzione della funzione di controllo.

Un algoritmo di controllo distribuito fornisce servizi sia alle applicazioni utente che al kernel del sistema operativo. La [Tabella 18.1](#) descrive le principali funzioni di controllo di

un sistema operativo distribuito. *Mutua esclusione* ed *elezione* sono servizi forniti ai processi utente, *gestione dei deadlock* e *schedulazione* sono utilizzati dal kernel, mentre la *rilevazione della terminazione* è un servizio usato sia dai processi utente che dal kernel. In letteratura, i nomi delle precedenti funzioni sono generalmente associati al termine “distribuita” per enfatizzare la loro natura.

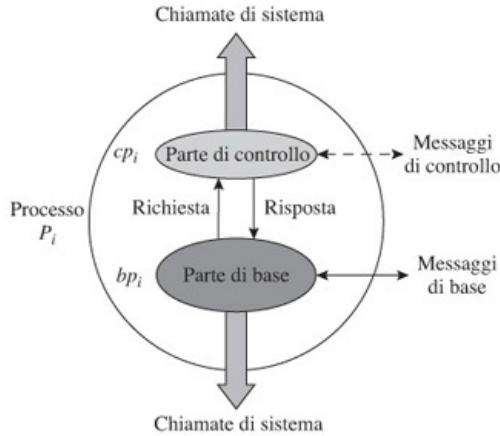
| Funzione                       | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Mutua esclusione               | Implementa la sezione critica (CS) per l’accesso a un dato $d_s$ da parte dei processi di un sistema distribuito. Ciò garantisce la sincronizzazione dei processi che operano su nodi differenti del sistema in modo che, in un dato istante di tempo, solo uno di essi possa trovarsi nella sezione critica di $d_s$ .                                                                                        |
| Gestione dei deadlock          | Previene o rileva deadlock dovuti alla condivisione di risorse all’interno di un nodo o tra i nodi di un sistema distribuito.                                                                                                                                                                                                                                                                                  |
| Schedulazione                  | Effettua operazioni di <i>bilanciamento del carico</i> per garantire che il carico computazione dei diversi nodi del sistema sia comparabile. Ciò richiede che i processi vengano trasferiti da nodi sovraccaricati verso nodi scarichi.                                                                                                                                                                       |
| Rilevazione della terminazione | I processi di una computazione distribuita possono agire su parecchi nodi del sistema. La rilevazione della terminazione consiste nel determinare se tale computazione ha completato la sua elaborazione. Ciò richiede di verificare se ciascun processo è ancora attivo e se vi sono messaggi in transito tra i vari processi.                                                                                |
| Elezione                       | Il processo <i>coordinatore</i> o <i>leader</i> è l’unico abilitato all’esecuzione di alcune operazioni privilegiate, come per esempio l’allocazione delle risorse. Nel caso in cui il coordinatore smetta di funzionare, è necessario eleggere un nuovo coordinatore. L’algoritmo di elezione seleziona come coordinatore uno dei processi attivi, e informa gli altri della nuova identità del coordinatore. |

**Tabella 18.1** Funzioni di controllo di un sistema operativo distribuito.

Un algoritmo di controllo distribuito opera in parallelo con i suoi client, in modo che esso possa rispondere prontamente agli eventi relativi al servizio fornito. Per distinguere tra le azioni di un client e quelle del corrispondente algoritmo di controllo useremo la seguente terminologia.

- *Elaborazione di base*: un’operazione di un client costituisce un’*elaborazione di base*. Essa può coinvolgere processi in uno o più nodi del sistema. I messaggi scambiati da tali processi sono chiamati *messaggi di base*.
- *Elaborazione di controllo*: un’operazione di un algoritmo di controllo costituisce un’*elaborazione di controllo*. I messaggi scambiati dai processi di un’elaborazione di controllo sono chiamati *messaggi di controllo*.

Per capire il modo di operare di un algoritmo di controllo distribuito, possiamo rappresentare ogni processo come se fosse costituito da due parti – una parte di base e una di controllo. La [Figura 18.1](#) illustra le due parti di un processo  $P_i$ . La *parte di base* si occupa di gestire l’elaborazione di base e scambia messaggi di base con le parti di base degli altri processi. Quando è richiesto un servizio fornito da un algoritmo di controllo, la parte di base invia una richiesta alla parte di controllo del processo. Tutte le altre richieste sono invece inviate direttamente al kernel. Al contrario, la *parte di controllo* di un processo si occupa di gestire l’elaborazione relativa al controllo e scambia messaggi di controllo con le parti di controllo degli altri processi. Essa può inoltre interagire con il kernel per implementare la sua funzione di controllo. Mentre la parte di base di un processo può bloccarsi quando effettua la richiesta di una risorsa, la parte di controllo non si blocca mai – questa caratteristica gli permette di rispondere agli eventi in modo tempestivo.



**Figura 18.1** Parti di un processo  $P_i$ .

### Esempio 18.1 - Parti di base e di controllo di un processo

Si consideri un'applicazione distribuita costituita da quattro processi  $P_1 - P_4$ , e si assuma che  $P_2$  si trovi nella sezione critica (CS) relativa a un dato  $d_s$ , condiviso con gli altri processi. Quando  $P_1$  vuole entrare nella sezione critica di  $d_s$ , la sua parte di base,  $bp_1$ , invia una richiesta alla parte di controllo,  $cp_1$ . Quest'ultima interagisce con uno degli algoritmi per la gestione della mutua esclusione descritti nel Paragrafo 18.3. Per decidere se  $P_1$  può entrare nella CS di  $d_s$ ,  $cp_1$  scambia messaggi con  $cp_2$ ,  $cp_3$  e  $cp_4$ . Dalle relative risposte,  $cp_1$  viene a conoscenza che qualche altro processo si trova nella CS di  $d_s$ , e quindi effettua una chiamata di sistema per bloccare  $bp_1$ . Si noti che  $cp_2$  partecipa a questa decisione anche se  $bp_2$  si trova in esecuzione all'interno di CS. Successivamente, quando  $P_2$  decide di uscire da CS,  $bp_2$  invia una richiesta a  $cp_2$ , che interagisce con la parte di controllo degli altri processi e decide che  $P_1$  può entrare nella CS di  $d_s$ . Di conseguenza,  $cp_1$  invia una richiesta al kernel per attivare  $bp_1$ .

## 18.2 Correttezza degli algoritmi di controllo distribuiti

I processi di un algoritmo di controllo distribuito scambiano dati e coordinano le loro azioni attraverso messaggi di controllo. Tuttavia, tale comunicazione è soggetta a ritardi. Pertanto, i dati usati dall'algoritmo possono diventare vecchi e inconsistenti e, in tal caso, l'algoritmo può fallire la sua missione mancando l'esecuzione di alcune azioni necessarie a una corretta prestazione oppure effettuare delle azioni dannose. Di conseguenza, la correttezza di un algoritmo di controllo distribuito presenta una doppia sfaccettatura.

- *Live ness*: l'algoritmo deve effettuare correttamente le azioni richieste, ovvero le compirà senza ritardi indefiniti.
- *Safety*: l'algoritmo non deve eseguire azioni sbagliate.

La mancanza di liveness implica che l'algoritmo potrebbe non riuscire a eseguire alcune azioni importanti. Per esempio, un algoritmo per la gestione della mutua esclusione potrebbe non soddisfare le proprietà di *progresso* e *attesa limitata* descritte nel Paragrafo 6.3.1, ovvero un algoritmo di gestione dei deadlock potrebbe non essere in grado di identificare la presenza di un deadlock nel sistema. Si noti che il tempo richiesto per effettuare l'azione necessaria è irrilevante per le proprietà di liveness; l'azione deve essere effettuata *prima o dopo*. La mancanza di sicurezza implica, invece, che l'algoritmo potrebbe eseguire azioni sbagliate, come per esempio consentire a più di un processo di accedere a una sezione critica nello stesso istante. La Tabella 18.2 riassume le proprietà di liveness e safety per alcuni algoritmi di controllo distribuiti.

| Algoritmo                      | Liveness                                                                                                                                                                                              | Safety                                                                                                           |
|--------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| Mutua esclusione               | (1) Se una CS è libera e alcuni processi hanno richiesto di potervi entrare, uno di essi vi accederà in tempo finito.<br>(2) Un processo che chiede di entrare in una CS vi accederà in tempo finito. | In ogni istante di tempo entrerà nella CS al massimo un processo.                                                |
| Gestione dei deadlock          | Se si origina un deadlock, esso verrà rilevato in tempo finito.                                                                                                                                       | Non verrà rilevato alcun deadlock a meno che non ne esista effettivamente uno.                                   |
| Rilevazione della terminazione | La terminazione di un'elaborazione distribuita verrà rilevata in tempo finito.                                                                                                                        | Non verrà rilevata la terminazione di un'elaborazione distribuita a meno che essa non abbia veramente terminato. |
| Elezione                       | Il nuovo coordinatore verrà eletto in tempo finito.                                                                                                                                                   | Verrà eletto coordinatore esattamente un solo processo.                                                          |

**Tabella 18.2** Proprietà di Liveness e Safety degli algoritmi di controllo distribuiti.

Assumendo che un algoritmo di controllo distribuito sia costituito da un insieme di azioni e condizioni distinte, possiamo rappresentarlo come un insieme di regole del tipo  $\langle\text{condizione}\rangle : \langle\text{azione}\rangle$ . Ogni regola specifica che l'algoritmo può eseguire una  $\langle\text{azione}\rangle$  se e solo se la corrispondente  $\langle\text{condizione}\rangle$  è vera. Usando la notazione " $\rightarrow$ " per indicare la frase "prima o poi conduce a," definiamo la nozione di correttezza per un algoritmo di controllo distribuito come segue.

- *Liveness*: per tutte le regole è vero che  $\langle\text{condizione}\rangle \rightarrow \langle\text{azione}\rangle$ , ovvero prima o poi  $\langle\text{azione}\rangle$  viene eseguita se  $\langle\text{condizione}\rangle$  diventa vera.
- *Mancanza di safety*: esiste almeno una regola per cui  $\neg \langle\text{condizione}\rangle \rightarrow \langle\text{azione}\rangle$ , ovvero  $\langle\text{azione}\rangle$  può essere eseguita anche se  $\langle\text{condizione}\rangle$  non si verifica mai.

Dimostrare la correttezza di un algoritmo distribuito non è semplice.  $\langle\text{condizione}\rangle$  e  $\langle\text{azione}\rangle$  devono essere specificate in modo da rappresentare correttamente l'algoritmo e, inoltre, è necessario utilizzare tecniche formali per dimostrare che l'algoritmo possiede le proprietà di liveness e safety. Fino ai primi anni '90 non esistevano i fondamenti teorici necessari per dimostrare formalmente la correttezza degli algoritmi distribuiti. Per questo motivo molti algoritmi sviluppati prima degli anni '90 contengono errori.

Va osservato che le proprietà di liveness e safety sono in relazione solo alla correttezza dell'algoritmo. Tuttavia, esistono altre proprietà rilevanti che dovrebbero essere definite e dimostrate per un algoritmo distribuito, come, per esempio, il fatto che le richieste inviate a un algoritmo che gestisce la mutua esclusione distribuita vengano servite secondo una politica FCFS.

## 18.3 Mutua esclusione distribuita

### 18.3.1 L'algoritmo di Ricart e Agrawala

L'algoritmo di Ricart e Agrawala (1981) è un algoritmo completamente distribuito che permette l'accesso alla sezione critica secondo una politica FCFS. Tutti i processi interessati a entrare nella sezione critica partecipano in modo paritario alla decisione su chi debba essere il prossimo ad accedervi. Un processo che desidera entrare nella sezione critica effettua una richiesta a tutti gli altri processi, inviando un messaggio con associato il proprio timestamp, e si mette in attesa di ricevere da questi ultimi il messaggio "OK". Se il sistema contiene  $n$  processi, l'algoritmo richiede lo scambio di  $2 \times (n - 1)$  messaggi prima che un processo possa entrare nella sezione critica. Le proprietà di safety e mutua esclusione derivano dal fatto che, in un certo istante di tempo, al massimo un solo processo può ottenere  $(n - 1)$  messaggi "OK". La proprietà di liveness è invece garantita dal fatto che l'accesso alla sezione critica avviene secondo una politica FCFS, quindi ogni processo è certo di poter accedervi in tempo finito.

### Algoritmo 18.1 Ricart-Agrawala

1. Quando un processo  $P_i$  desidera entrare nella CS:  $P_i$  invia un messaggio di richiesta del tipo ("richiesta",  $P_i$ ,  $<\text{timestamp}>$ ) a tutti i processi del sistema, e si blocca.
2. Quando un processo  $P_i$  riceve un messaggio di richiesta dal processo  $P_r$ :
  - a. se  $P_i$  non è interessato ad accedere a CS, esso invia immediatamente il messaggio "OK" a  $P_r$ ;
  - b. se anche  $P_i$  desidera entrare in CS, esso invia il messaggio "OK" a  $P_r$  se il timestamp contenuto nel messaggio ricevuto da  $P_r$  è minore del timestamp della propria richiesta; altrimenti, esso aggiunge l'id del processo  $P_r$  in una coda di attesa;
  - c. se  $P_i$  è in CS, esso aggiunge l'id del processo  $P_r$  nella coda di attesa.
3. Quando un processo  $P_i$  riceve  $n - 1$  messaggi "OK": il processo diventa attivo ed entra in CS.
4. Quando un processo  $P_i$  esce dalla CS: il processo invia il messaggio "OK" a ogni processo che aveva precedentemente inserito nella sua coda di attesa.

La [Tabella 18.3](#) mostra come sono implementati i vari passi dell'Algoritmo 18.1 nella parte di controllo di un processo. La prima colonna mostra i passi eseguiti da un processo durante la computazione di base. Quest'ultima è costituita da un ciclo in cui il processo (1) chiede di entrare nella sezione critica, (2) esegue codice all'interno della sezione critica, e (3) esce dalla sezione critica. Le altre colonne della tabella mostrano invece le azioni eseguite dalla parte di controllo dell'algoritmo. Il numero di messaggi richiesti per entrare nella sezione critica può essere ridotto se si consente a un processo  $P_i$  di attendere l'acquisizione del permesso di accesso da parte di un sottoinsieme  $R_i$  di tutti i processi del sistema. In tal caso  $R_i$  è detto *request set* di  $P_i$ . Per preservare le proprietà di safety è necessario definire attentamente i componenti di un request set. L'algoritmo proposto da Maekawa (1985) utilizza request set di dimensione  $\sqrt{n}$ , e adotta le seguenti regole per garantire la safety (Problema 18.3):

| Parte di base                                          | Passi dell'algoritmo eseguiti dalla parte di controllo |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|--------------------------------------------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                        | Passi                                                  | Dettagli                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| Ripeti continuamente<br>{Richiesta di accesso alla CS} | 1,2(b),3                                               | <ol style="list-style-type: none"> <li>i. Invia il messaggio ("richiesta", <math>P_i</math>, <math>&lt;\text{timestamp}&gt;</math>) a tutti gli altri processi e richiede al kernel di bloccare la parte di base.</li> <li>ii. Alla ricezione di una richiesta di accesso alla sezione critica da parte di un altro processo, invia il messaggio "OK" se la richiesta ha un timestamp più piccolo del proprio; altrimenti, aggiungi nella coda di attesa l'id del processo che ha effettuato la richiesta.</li> <li>iii. Conta le risposte "OK" ricevute. Attiva la parte di base del processo dopo aver ricevuto <math>(n - 1)</math> risposte.</li> </ol> |
| {Sezione critica}                                      | 2(c)                                                   | Inserisci tutte le richieste ricevute nella coda di attesa.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| {Uscita dalla sezione critica}                         | 4                                                      | Invia il messaggio "OK" a tutti i processi precedentemente inseriti nella coda di attesa.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| {Rimanente parte del ciclo}                            | 2(a)                                                   | Alla ricezione di una richiesta di accesso alla sezione critica da parte di un processo, invia immediatamente il messaggio "OK".                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

**Tabella 18.3** Azioni di base e di controllo effettuate da un processo in un algoritmo di mutua esclusione completamente distribuito.

1. per ogni  $P_i : P_i \in R_i$ ;
2. per ogni  $P_i, P_j : R_i \cap R_j \neq \emptyset$ .

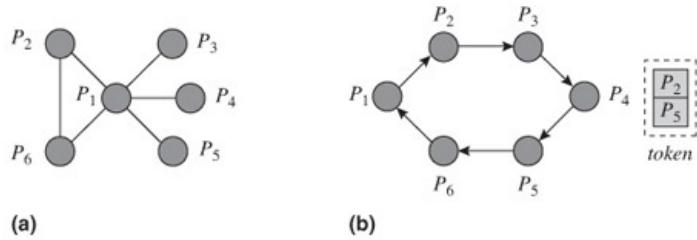
### 18.3.2 Algoritmi di mutua esclusione che utilizzano un token

Il *token* rappresenta un lasciapassare per poter usare la sezione critica; infatti, solo un

processo in possesso del token può entrare nella sezione critica. Tale regola garantisce che gli algoritmi basati su token siano corretti rispetto alla sicurezza. La correttezza rispetto alla liveness è invece garantita dal fatto che quando un processo invia la richiesta per poter entrare in una sezione critica, l'algoritmo di mutua esclusione si assicura che la richiesta arrivi al processo in possesso del token e che quest'ultimo venga inviato al processo richiedente. La complessità e il costo dell' algoritmo di mutua esclusione dipendono dalle proprietà del modello del sistema. Pertanto, gli algoritmi basati su token usano *modelli astratti del sistema* basati su grafi, i cui archi rappresentano i cammini usati per scambiare i messaggi di controllo. Nel seguito presenteremo due di tali algoritmi: uno basato sul concetto di *anello astratto*, l'altro invece su concetto di *albero*.

### Un algoritmo basato sulla topologia ad anello

La [Figura 18.2](#) mostra il modello logico di una computazione distribuita e il corrispondente modello astratto basato su una topologia ad anello unidirezionale. Il token è rappresentato da un oggetto, ovvero una struttura dati, che contiene la coda delle richieste di accesso alla sezione critica pendenti. In [Figura 18.2\(b\)](#), il token è posseduto dal processo  $P_4$  che si trova nella sezione critica, mentre la coda delle richieste del token contiene  $P_2$  e  $P_5$ . L'algoritmo opera come segue: un processo che desidera entrare nella sezione critica invia un messaggio contenente la sua richiesta e si blocca. Il messaggio viene inoltrato a tutti i nodi dell'anello finché non raggiunge il possessore del token. Se quest'ultimo si trova nella sezione critica, la sua parte di controllo inserisce l'id del processo richiedente nella coda delle richieste contenuta nel token. Quando il possessore del token esce dalla sezione critica, si estrae il primo processo dalla coda delle richieste e invia nell'anello un messaggio contenente il token e l'id di tale processo. Quando il messaggio raggiunge il processo indicato nel token, la parte di controllo di quest'ultimo estrae il token e attiva la corrispondente parte di base, che può quindi iniziare l'accesso alla sezione critica. Per esempio, in [Figura 18.2\(b\)](#), il processo  $P_2$  riceve il token quando  $P_4$  esce dalla sua sezione critica. Il numero di messaggi che i processi devono scambiarsi affinché uno di loro possa entrare nella sezione è pari a  $n$ , dove  $n$  rappresenta il numero di processi nel sistema.



**Figura 18.2** (a) Modello di sistema; (b) modello di sistema astratto.

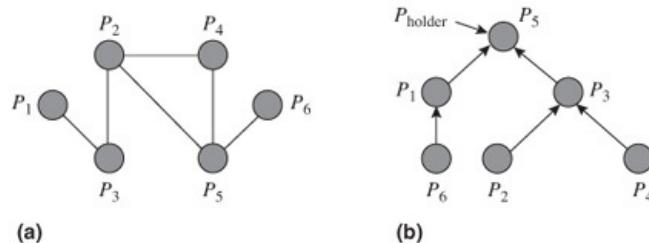
#### Algoritmo 18.2 Mutua esclusione basata su token per reti con topologia ad anello

1. Quando un processo  $P_i$  desidera accedere alla sezione critica: il processo invia un messaggio di tipo ("richiesta",  $P_i$ ) sul suo canale di uscita e si blocca.
2. Quando un processo  $P_i$  riceve una richiesta da un processo  $P_r$ : se  $P_i$  non possiede il token,  $P_i$  inoltra la richiesta sul suo canale di uscita. Se  $P_i$  possiede il token e non si trova nella sezione critica,  $P_i$  invia un messaggio del tipo ("token",  $P_r$ ) sul suo canale di uscita. Se  $P_i$  si trova nella sezione critica,  $P_i$  inserisce  $P_r$  nella coda di attesa del token.
3. Quando un processo  $P_i$  completa la sua esecuzione nella sezione critica:  $P_i$  verifica lo stato della coda delle richieste del token. Se non è vuota,  $P_i$  estrae l'id del primo processo in coda (supponiamo si tratti di  $P_r$ ) e invia un messaggio del tipo ("token",  $P_r$ ) sul suo canale di uscita.
4. Quando un processo  $P_i$  riceve un messaggio del tipo ("token",  $P_j$ ):  $P_i$  verifica se  $P_i = P_j$ . Se il confronto ha esito positivo,  $P_i$  crea una struttura dati locale per memorizzare

il token ed entra nella sezione critica. Al contrario, se il confronto ha esito negativo,  $P_i$  inoltra il messaggio sul suo canale di uscita.

### Algoritmo di Raymond

L'algoritmo di Raymond si avvale di un *albero astratto inverso* come modello del sistema. Un albero inverso si differenzia da un albero convenzionale per il fatto che la direzione di ogni arco va dal nodo figlio verso il nodo padre. Il processo in possesso del token rappresenta la radice dell'albero che, nel seguito, indicheremo con la notazione  $P_{\text{holder}}$ . L'algoritmo si basa su quattro caratteristiche fondamentali – uso di proprietà invarianti per garantire che la richiesta raggiunga  $P_{\text{holder}}$ , memorizzazione di un coda locale in ogni nodo per tenere traccia dei processi che chiedono di accedere alla sezione critica, impiego di tecniche ad hoc per ridurre il numero di messaggi di richiesta, utilizzo di particolari accorgimenti per garantire le proprietà di liveness.



**Figura 18.3** (a) Modello di un sistema; (b) modello astratto dello stesso sistema.

La [Figura 18.3](#) mostra il modello di un'elaborazione distribuita e il corrispondente albero astratto inverso. Il processo  $P_5$  detiene il token, e pertanto rappresenta la radice dell'albero. I processi  $P_1$  e  $P_3$ , che hanno archi uscenti diretti verso  $P_5$ , sono figli di quest'ultimo. Similmente, gli archi  $(P_6, P_1)$ ,  $(P_2, P_3)$  e  $(P_4, P_3)$  vanno dal processo figlio verso il corrispondente padre. L'algoritmo preserva tre proprietà invarianti sull'albero astratto inverso:

1. il processo  $P_{\text{holder}}$  è la radice dell'albero;
2. ogni processo del sistema appartiene all'albero;
3. ogni processo  $P_i \neq P_{\text{holder}}$  ha esattamente un arco di uscita  $(P_i, P_j)$ , dove  $P_j$  è il suo nodo padre.

Le tre proprietà invarianti assicurano che il modello astratto del sistema contenga un cammino da ogni processo  $P_i \vdash P_{\text{holder}}$  verso  $P_{\text{holder}}$ . Ciò garantisce che una eventuale richiesta di accesso alla sezione critica eseguita da  $P_i$  raggiunga sempre il processo  $P_{\text{holder}}$ . Quando un processo  $P_k$  invia il token a un altro processo  $P_j$ , le proprietà precedenti vengono preservate cambiando la direzione del corrispondente arco, ovvero  $(P_j, P_k)$  viene invertito. In tal modo si inverte la direzione lungo cui il token verrà successivamente spedito, e viene stabilito un cammino tra il vecchio e il nuovo possessore del token. Per esempio, l'arco  $(P_3, P_5)$  in [Figura 18.3\(b\)](#) verrebbe invertito nel momento in cui  $P_5$  inviasse il token a  $P_3$ .

Ogni processo mantiene localmente una coda delle richieste di accesso alla sezione critica ricevute. I messaggi di richiesta hanno un solo campo contenente l'id del processo richiedente, ovvero *id\_richiedente*. Un processo che desidera accedere alla sezione critica inserisce il proprio id nella coda locale e invia il messaggio di richiesta sul suo canale di uscita. Quando un processo  $P_i$  riceve una richiesta, esso inserisce l'id del processo richiedente nella sua coda locale e invia nel suo canale di uscita un nuovo messaggio di richiesta con il proprio id, cioè  $P_i$ . In tal modo, prima o poi  $P_{\text{holder}}$  riceve (lungo un qualche cammino, la cui esistenza è garantita dalla proprietà invariante 3) una richiesta di accesso alla sezione critica. Tuttavia, via via che la richiesta risale l'albero verso la radice, l'id del processo richiedente viene modificato nel passaggio da un arco al successivo. Per ridurre il numero totale di messaggi in transito nell'albero, un processo non invia una richiesta finché una sua eventuale precedente richiesta non è stata

esaudita (il processo è in grado di riconoscere una tale situazione poiché se la richiesta fosse stata esaudita esso avrebbe già ricevuto il token). Durante la sua esecuzione all'interno della sezione critica,  $P_{\text{holder}}$  inserisce tutte le richieste che riceve nella coda locale. Quindi, quando esce dalla sezione critica,  $P_{\text{holder}}$  estrae l'id del primo processo della coda e gli invia il token. A sua volta, il processo ricevente inoltra il token al primo processo presente nella sua coda locale, e così via. Tale procedura si arresta quando il processo che riceve il token si accorge che l'id in cima alla coda corrisponde al proprio id. In tal caso, la parte di controllo di tale processo acquisisce il token mentre la corrispondente parte di base si attiva ed entra nella sezione critica.

Per preservare la liveness è necessario che ciascun processo che richiede di poter entrare nella sezione critica riesca nel proprio intento in un tempo finito. Per garantire questa proprietà, un processo  $P_i$ , prima di inoltrare il token a un altro processo  $P_j$ , verifica se la sua coda locale è vuota. Se la coda contiene almeno una richiesta,  $P_i$  invia a  $P_j$ , oltre al token, anche un messaggio di richiesta inserendo nel campo *requester\_id* il proprio id. Così facendo,  $P_i$  si garantisce di ricevere il token in un istante successivo, in modo da poter servire le altre richieste ancora pendenti nella sua coda locale.

### Algoritmo 18.3 (Raymond)

1. *Quando un processo  $P_i$  desidera entrare nella sezione critica:*  $P_i$  inserisce il suo id nella coda locale e invia un messaggio di richiesta contenente il proprio id sul suo canale di uscita, a meno che non abbia già inviato una simile richiesta senza aver ancora ricevuto la corrispondente risposta.
2. *Quando un processo  $P_i$  riceve un messaggio di richiesta da un processo  $P_r$ :*  $P_i$  esegue le seguenti operazioni:
  - a. inserisce  $P_r$  nella sua coda locale;
  - b. se  $P_i \neq P_{\text{holder}}$ , invia un messaggio di richiesta contenente il suo id, ovvero  $P_i$ , sul suo canale di uscita a meno che non abbia già inviato una simile richiesta senza aver ancora ricevuto la corrispondente risposta.
3. *Quando un processo  $P_i$  termina l'esecuzione all'interno della sezione critica:*  $P_i$  effettua le seguenti operazioni:
  - a. rimuove l'id del processo in cima alla sua coda locale. Supponiamo si tratti di  $P_j$ ;
  - b. invia il token a  $P_j$ ;
  - c. inverte l'arco  $(P_j, P_i)$ ;
  - d. se la coda locale non è vuota, invia un messaggio di richiesta contenente il suo id, ovvero  $P_i$ , a  $P_j$ .
4. *Quando un processo  $P_i$  riceve il token:*
  - a. se il suo id è sulla cima della coda locale,  $P_i$  rimuove la richiesta dalla coda, attiva la sua parte di base ed entra nella sezione critica;
  - b. in caso contrario,  $P_i$  esegue i passi 3(a)-(d).

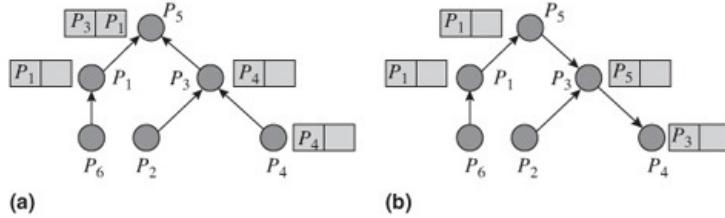
L'algoritmo necessita che per ogni richiesta vengano spediti  $\log n$  messaggi. Inoltre, esso non garantisce che l'accesso alla sezione critica avvenga secondo una politica FIFO [passo 2(b)]. Il funzionamento dell'algoritmo è illustrato nell'Esempio 18.2.

### Esempio 18.2 - Algoritmo di Raymond

La Figura 18.4(a) mostra lo stato del sistema rappresentato in Figura 18.3 dopo che le richieste effettuate da  $P_4$  e  $P_1$  hanno raggiunto il processo  $P_5$  possessore del token (passi 1 e 2 dell'Algoritmo 18.3). Quando il processo  $P_5$  esce dalla sezione critica, esso rimuove  $P_3$  dalla sua coda locale, invia il token a quest'ultimo, e inverte l'arco  $(P_3, P_5)$ . A questo punto,  $P_5$  invia una richiesta a  $P_3$  in quanto la sua coda locale non è vuota [si veda il passo 3(d)]. Similmente (passo 4), il processo  $P_3$  invia il token al processo  $P_4$ , inverte l'arco  $(P_4, P_3)$ , e invia una richiesta a  $P_4$ .

La Figura 18.4(b) mostra l'albero astratto inverso risultante dalle precedenti operazioni. Dopo l'uscita di  $P_4$  dalla sezione critica, in modo analogo a quanto precedentemente descritto, il token viene inviato a  $P_1$  attraverso  $P_3$  e  $P_5$ . In tal modo

$P_1$  può accedere a sua volta alla sezione critica. Va osservato come ciò non sarebbe possibile se l'algoritmo non prevedesse il passo 3(d).



**Figura 18.4** Esempio di esecuzione dell'algoritmo di Raymond.

## 18.4 Gestione dei deadlock distribuiti

Le tecniche descritte nel Paragrafo 8.3 volte a rilevare, prevenire ed evitare deadlock, utilizzano informazioni sullo stato del sistema. Questo paragrafo illustra innanzitutto i problemi che emergono quando si tenta di estendere tali tecniche a un sistema distribuito e, in secondo luogo, descrive alcuni approcci distribuiti per la rilevazione e la prevenzione dei deadlock. Al contrario, non verranno presentate tecniche distribuite per evitare il formarsi di deadlock, in quanto non ne esistono in letteratura. Per semplificare la trattazione dei paragrafi successivi, considereremo solo modelli di allocazione delle risorse di tipo SISR, ovvero singola istanza, singola richiesta (Paragrafo 8.3).

### 18.4.1 Problemi delle tecniche centralizzate per la rilevazione dei deadlock

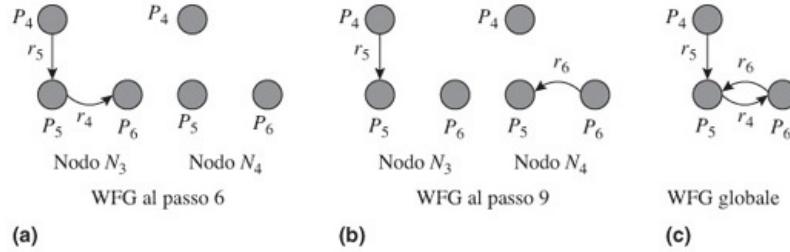
Le applicazioni distribuite possono usare risorse posizionate in vari nodi del sistema. I deadlock causati da tali applicazioni possono essere rilevati raccogliendo e unendo, in un singolo nodo, i grafi di attesa (WFG, ovvero *wait-for graphs*) di tutti i nodi del sistema. In tal modo è possibile applicare sul WFG globale uno qualunque degli algoritmi di rilevazione usati nei sistemi convenzionali, non distribuiti. Tuttavia, tale approccio presenta un inconveniente. I WFG raccolti dal nodo coordinatore potrebbero riferirsi a istanti di tempo differenti, e quindi il WFG globale potrebbe mostrare una visione fuorviante delle relazioni di attesa realmente esistenti tra i processi del sistema. Una simile eventualità può portare alla rilevazione di falsi deadlock, detti *deadlock fantasma*, violando così la proprietà di safety. L'Esempio 18.3 mostra una tale situazione.

#### Esempio 18.3 - Deadlock fantasma

Si consideri la seguente sequenza di eventi in un sistema distribuito composto da tre processi  $P_4$ ,  $P_5$  e  $P_6$ :

1.  $P_5$  richiede e ottiene la risorsa  $r_5$  posizionata nel nodo  $N_3$ ;
2.  $P_6$  richiede e ottiene la risorsa  $r_4$  posizionata nel nodo  $N_3$ ;
3.  $P_5$  richiede e ottiene la risorsa  $r_6$  posizionata nel nodo  $N_4$ ;
4.  $P_4$  richiede la risorsa  $r_5$  posizionata nel nodo  $N_3$ ;
5.  $P_5$  richiede la risorsa  $r_4$  posizionata nel nodo  $N_3$ ;
6. il nodo  $N_3$  invia il suo WFG locale al coordinatore;
7.  $P_6$  rilascia la risorsa  $r_4$  posizionata nel nodo  $N_3$ ;
8.  $P_6$  richiede la risorsa  $r_6$  posizionata nel nodo  $N_4$ ;
9. il nodo  $N_4$  invia il suo WFG locale al coordinatore.

Le Figure 18.5(a) e (b) mostrano i WFG dei vari nodi, rispettivamente, dopo il passo 6 e il passo 9. È possibile osservare che in entrambi i casi non esistono deadlock nel sistema. Tuttavia, il WFG globale, costruito unendo il WFG del nodo  $N_3$  al passo 6 e il WFG del nodo  $N_4$  al passo 9 [Figura 18.5(c)], contiene un ciclo  $\{P_5, P_6\}$ . In una tale situazione, il coordinatore rileva la presenza di un deadlock che non esiste realmente. Siamo pertanto in presenza di un deadlock fantasma.



**Figura 18.5** Esempio di deadlock fantasma: WFG dei nodi al passo 6 e al passo 9, e WFG globale.

### 18.4.2 Rilevazione dei deadlock distribuiti

Come visto nel [Capitolo 8](#), la presenza di un ciclo o di un nodo in un WFG è condizione necessaria e sufficiente affinché si origini un deadlock, rispettivamente, nei sistemi SISR e MISR. Nell'approccio distribuito, cicli e nodi sono identificati grazie alla cooperazione dei vari siti del sistema, in modo tale che ognuno di essi sia in grado di rilevare e dichiarare la presenza di deadlock. Nel paragrafi seguenti illustreremo due algoritmi per la rilevazione dei deadlock basati su tecniche distribuite.

#### Algoritmo basato su computazione per propagazione

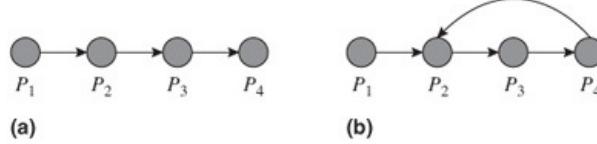
L'elaborazione per propagazione (*diffusion computation*) è stata proposta da Dijkstra e Scholten (1980) per identificare la terminazione dell'elaborazione. La computazione per propagazione è costituita da due fasi - una di propagazione e una di raccolta delle informazioni. Nella fase di propagazione, l'elaborazione inizia in un nodo e si diffonde verso gli altri nodi tramite messaggi di controllo detti *query* che vengono inviati su tutti gli archi del sistema. Un nodo può ricevere più di una query se ha più di un canale di ingresso. La prima query ricevuta da un nodo è detta *engaging query*, letteralmente query di ingaggio, mentre le successive sono chiamate *nonengaging query*. Quando un nodo riceve una engaging query, esso invia una query su ogni suo canale di uscita. Al contrario, alla ricezione di una nonengaging query, il nodo non invia nulla.

Nella fase di raccolta delle informazioni, ogni nodo del sistema invia una risposta a ogni query ricevuta. La risposta a una query di tipo engaging contiene informazioni riguardanti sia il nodo a cui quest'ultima era diretta che ai nodi a esso collegati. Al contrario, la risposta a una nonengaging query tipicamente non contiene alcuna informazione. Per questo motivo essa viene chiamata *dummy replay*, ovvero *risposta falsa*. Se il nodo che per primo ha dato origine alla fase di propagazione riceve la sua stessa query, esso invia immediatamente una dummy replay. L'algoritmo di Chandy-Lamport per la registrazione di uno stato consistente di un sistema distribuito, descritto nel [Paragrafo 17.4.2](#), fa uso della prima fase della computazione per propagazione (Problema 18.5).

L'Algoritmo 18.4 utilizza, invece, l'elaborazione per propagazione per rilevare la presenza di deadlock. Tale algoritmo, proposto da Chandy, Misra e Haas (1983), funziona sia per sistemi SISR che MISR. L'elaborazione si propaga attraverso tutti gli archi del WFG, e tutti i passi dell'algoritmo sono eseguiti in modo atomico, pertanto se un processo riceve due messaggi nello stesso istante di tempo, essi vengono elaborati uno di seguito all'altro. L'algoritmo assume che computazioni iniziate da processi distinti siano rappresentate da identificatori distinti. Gli identificatori vengono utilizzati per etichettare le query e le corrispondenti reply, e in tal modo, non si ha interferenza tra computazioni differenti.

Si consideri un sistema SISR costituito da quattro processi  $P_1 - P_4$ . Il WFG di [Figura 18.6\(a\)](#) mostra lo stato del sistema dopo che il processo  $P_1$  ha richiesto una risorsa correntemente allocata a  $P_2$ .  $P_1$ ,  $P_2$  e  $P_3$  sono *bloccati*, mentre  $P_4$  può proseguire la sua esecuzione.  $P_1$  inizia, quindi, un'elaborazione per propagazione. Quando  $P_2$  riceve la query di  $P_1$ , invia una query a  $P_3$ , che a sua volta invia una query a  $P_4$ . Tuttavia,  $P_4$  non è bloccato, e quindi, non risponde a  $P_3$ . Conseguentemente,  $P_1$  non riceverà mai una reply alla sua query e non dichiarerà alcun deadlock. Supponiamo ora che  $P_4$  richieda la risorsa allocata a  $P_2$  e si blocchi di conseguenza [WFG di [Figura 18.6\(b\)](#)]. In una tale

situazione,  $P_4$  inizia un'elaborazione per propagazione che si diffonde verso i processi  $P_2$  e  $P_3$ . Dal momento che anche questi processi sono bloccati,  $P_4$  riceverà una reply per la sua query e dichiarerà di essere coinvolto in un deadlock.



**Figura 18.6** Esempio di rilevazione di deadlock distribuita basata su computazione per propagazione.

**Algoritmo 18.4 Rilevazione dei deadlock distribuita mediante elaborazione per propagazione**

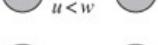
1. *Quando un processo si blocca su una risorsa:* il processo inizia una computazione per propagazione eseguendo le seguenti azioni:
  - a. invia query su tutti i suoi archi di uscita nel WFG;
  - b. ricorda le query inviate e attende le corrispondenti reply;
  - c. se riceve una reply per tutte le query inviate ed è rimasto continuamente bloccato dal momento in cui ha iniziato la computazione per propagazione, dichiara la presenza di un deadlock.
2. *Quando un processo riceve una engaging query:* se il processo è bloccato, esso esegue le seguenti azioni:
  - a. invia query su tutti i suoi archi di uscita nel WFG;
  - b. ricorda le query inviate e attende le corrispondenti reply;
  - c. se riceve una reply per tutte le query inviate ed è rimasto continuamente bloccato dal momento in cui ha ricevuto la engaging query, invia una reply al nodo da cui ha ricevuto la engaging query.
3. *Quando un processo riceve una nonengaging query:* se il processo è rimasto bloccato continuamente dal momento in cui ha ricevuto la engaging query, invia una dummy reply al nodo da cui ha ricevuto la nonengaging query.

L'algoritmo di Chandy, Misra e Haas non rileva mai deadlock fantasma grazie alla condizione che richiede ai processi di essere sempre rimasti bloccati, dal momento in cui essi iniziano l'elaborazione per propagazione, o dal momento in cui ricevono la engaging query, per poter, rispettivamente, dichiarare un deadlock o inviare una reply.

**L'algoritmo edge chasing**

L'algoritmo *edge chasing*, proposto da Mitchell e Merritt (1982), si basa sull'invio di messaggi di controllo attraverso gli archi di attesa del WFG per facilitare la rilevazione dei cicli. Da questa caratteristica deriva il nome dell'algoritmo (*to chase* significa letteralmente "cacciare"). Ogni processo è etichettato con una coppia di numeri detti *etichetta pubblica* ed *etichetta privata*. Al momento della creazione del processo tali etichette sono identiche. Il loro valore cambia quando un processo si blocca su una risorsa. L'etichetta pubblica di un processo cambia anche quando esso si mette in attesa di un processo con etichetta pubblica più grande. L'esistenza di un deadlock viene segnalata quando esiste una particolare relazione tra le etichette pubbliche e private dei processi sorgente e destinatario di un arco di attesa.

La [Figura 18.7](#) illustra le regole dell'algoritmo di Mitchell-Merritt. Un processo è rappresentato come  $\frac{u}{v}$  dove  $u$  e  $v$  sono, rispettivamente, la sua etichetta pubblica e la sua etichetta privata. Ciascuna regola viene applicata quando le etichette dei processi sorgente e destinazione di un arco di attesa soddisfano la pre-condizione. La regola cambia le etichette dei processi come indicato alla destra del simbolo  $\Rightarrow$ . I dettagli delle quattro regole sono i seguenti.

| Nome della regola | Pre-condizione                                                                    | Dopo l'applicazione della regola                                                                 |
|-------------------|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Blocco            |  | $\Rightarrow$  |
| Attivazione       |  | $\Rightarrow$  |
| Trasmissione      |  | $\Rightarrow$  |
| Rilevazione       |  | $\Rightarrow$  |

**Figura 18.7** Regole dell'algoritmo di Mitchell-Merritt.

1. *Blocco*: l'etichetta pubblica e l'etichetta privata di un processo vengono impostate al valore  $z$  quando esso si blocca su una risorsa. Il valore  $z$  viene generato tramite l'istruzione  $z := inc(u, x)$ , dove  $u$  è l'etichetta pubblica del processo in attesa,  $x$  è l'etichetta pubblica del processo che causa l'attesa, e la funzione  $inc$  genera un valore unico maggiore sia di  $u$  che di  $x$ .
2. *Attivazione*: l'arco di uscita di un processo viene rimosso dal WFG quando il processo viene attivato in seguito all'allocazione di una risorsa. Le sue etichette rimangono invariate.
3. *Trasmissione*: se l'etichetta pubblica  $u$  del processo sorgente di un arco di attesa è più piccola dell'etichetta pubblica  $w$  del processo destinazione, allora  $u$  viene sostituito con  $w$ .
4. *Rilevazione*: viene dichiarata la presenza di un deadlock se l'etichetta pubblica e l'etichetta privata del processo sorgente di un arco di attesa sono uguali all'etichetta pubblica del processo destinazione dello stesso arco.

Il modo di operare dell'algoritmo può essere descritto come segue: si consideri un cammino nel WFG dal processo  $P_i$  al processo  $P_k$ . Inoltre, si supponga che le etichette di  $P_i$  e  $P_k$  siano rispettivamente  $\frac{u_i}{v_i}$  e  $\frac{u_k}{v_k}$ . Applicando la regola di *trasmissione* a tutti gli archi del cammino da  $P_i$  a  $P_k$ , si può notare come  $u_i$  sia maggiore o uguale a tutte le etichette pubbliche dei processi inclusi nel cammino. Si assuma, quindi, che  $P_k$  effettui una richiesta che dà origine all'arco di attesa  $(P_k, P_i)$ . In base alla regola del *blocco*, le etichette di  $P_k$  assumono un valore dipendente da  $inc(u_k, u_i)$ . Immaginando che tale valore sia  $n$ , si avrà che  $n > u_i$ . Successivamente, in base alla regola di *trasmissione*,  $n$  si propaga verso  $P_i$  attraverso i processi inclusi nel cammino da  $P_i$  a  $P_k$ . In seguito a ciò, l'arco  $(P_k, P_i)$  soddisfa la regola di *rilevazione*. Come esempio, si consideri il sistema mostrato in [Figura 18.6](#). Nel momento in cui il processo  $P_4$  si blocca, esso riceve una nuova coppia di valori per le sue etichette. La sua etichetta pubblica sarà maggiore dell'etichetta pubblica di  $P_2$  e  $P_3$ , e quindi essa si propagherà verso  $P_2$  passando per  $P_3$ . Di conseguenza, il processo  $P_4$  rileverà un deadlock.

La correttezza dell'algoritmo deriva dal fatto che l'etichetta pubblica di un processo  $P_i$  che si trova dal lato sorgente di un arco di attesa si propaga verso un processo  $P_j$  solo se esiste un cammino da  $P_j$  a  $P_i$  (si veda il passo relativo alla regola di trasmissione).

Pertanto, se l'arco di attesa da  $P_i$  a  $P_j$  soddisfa la regola di rilevazione, esso è responsabile della presenza di un ciclo nel WFG, e quindi di un deadlock. La safety è facilmente garantita se non si consente ai processi di ritirare le proprie richieste spontaneamente.

### 18.4.3 Prevenzione dei deadlock distribuiti

Gli approcci per la prevenzione dei deadlock presentati nel Paragrafo 8.5 evitano il verificarsi di cicli nei grafi di richiesta e allocazione delle risorse (RRAG) e nei grafi di attesa (WFG) ponendo vincoli alle richieste delle risorse. In un sistema distribuito, i deadlock possono essere prevenuti in modo analogo. Ogni processo, al momento della sua creazione, viene etichettato con un timestamp (marchio temporale) formato dalla

coppia (tempo locale, id del nodo). Le attese circolari su RRAG e WFG vengono quindi evitate impedendo il formarsi di alcuni tipi di relazioni di attesa. Ciò si ottiene confrontando i timestamp dei vari processi per mezzo della relazione 17.1. Due schemi basati su tale approccio sono i seguenti.

- *Attendi-o-muori (wait-or-die)*: quando un processo  $P_{\text{req}}$  effettua una richiesta di una risorsa posseduta da  $P_{\text{holder}}$ ,  $P_{\text{req}}$  può mettersi in attesa della risorsa solo se è più vecchio di  $P_{\text{holder}}$ ; in caso contrario,  $P_{\text{req}}$  termina. Con questo schema, non possono verificarsi attese circolari in quanto un processo più vecchio può attendere su un processo più giovane, ma il contrario non è ammesso.
- *Ferisci-o-attendi (wound-or-wait)*: se  $P_{\text{req}}$  è più giovane di  $P_{\text{holder}}$ ,  $P_{\text{req}}$  può mettersi in attesa di una risorsa posseduta da  $P_{\text{holder}}$ ; in caso contrario,  $P_{\text{holder}}$  termina e la risorsa viene allocata a  $P_{\text{req}}$ . In tal modo, un processo giovane può attendere che un processo più vecchio termini, ma il contrario non è ammesso.

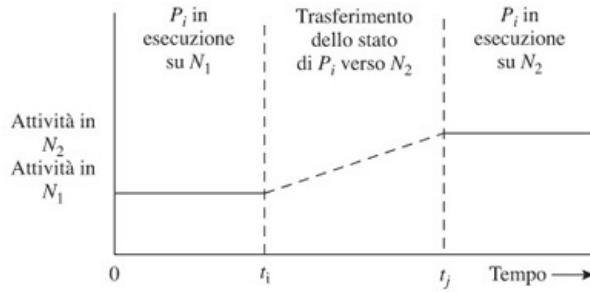
In entrambi gli approcci, il processo più giovane termina e deve ricominciare la sua esecuzione. Per evitare fenomeni di starvation dovuti a continue terminazioni, un processo può preservare il suo vecchio timestamp quando viene fatto ripartire. In pratica, lo schema attendi-o-muori è preferibile poiché, al contrario dello schema ferisci-o-attendi, non richiede prelazione delle risorse.

## 18.5 Algoritmi di schedulazione distribuiti

Sia le prestazioni del sistema che la velocità di calcolo delle applicazioni sarebbero negativamente influenzate se il carico computazionale dei nodi di un sistema distribuito fosse non uniforme. Gli algoritmi di schedulazione distribuiti bilanciano il carico computazione nei nodi del sistema trasferendo processi dai nodi sovraccarichi verso i nodi meno impegnati. Tale tecnica, nota con il nome di *migrazione dei processi*, è illustrata in [Figura 18.1](#). Al tempo  $t = 0$ , viene creato il processo  $P_i$  nel nodo  $N_1$ . Al tempo  $t_i$ , la funzione di schedulazione decide di trasferire il processo verso il nodo  $N_2$ , pertanto, l'esecuzione del processo nel nodo  $N_1$  viene interrotta, e il kernel inizia il trasferimento dello stato del processo verso il nodo  $N_2$ . Al tempo  $t_j$ , il trasferimento viene completato e il processo inizia la sua esecuzione sul nodo  $N_2$ .

Per effettuare il bilanciamento del carico tramite migrazione dei processi, l'algoritmo di schedulazione misura il carico computazionale dei vari nodi, e applica un valore *soglia* per decidere quali nodi sono troppo carichi e quali lo sono troppo poco. Di conseguenza, a tempo opportuno, l'algoritmo trasferisce processi dai primi, detti *nodi mittenti* verso i secondi, detti *nodi riceventi*. L'utilizzo della CPU è un indicatore diretto del carico computazionale di un nodo; tuttavia, monitorare continuamente tale parametro porta a un elevato sovraccarico della macchina. Pertanto, è preferibile usare, come misura del carico computazionale, il numero di processi in esecuzione su un nodo o la lunghezza della sua *ready queue*. Tali parametri possiedono una buona correlazione con il tempo di risposta medio di un nodo, e il loro utilizzo richiede un sovraccarico ridotto.

La migrazione di un processo può essere effettuata in due modi. La *migrazione con prelazione* richiede la sospensione di un processo, la registrazione del suo stato, il trasferimento del processo verso un altro nodo e la riattivazione dell'esecuzione di quest'ultimo ([Figura 18.8](#)); ciò richiede un notevole supporto da parte del kernel. Nella *migrazione senza prelazione*, la decisione sul bilanciamento del carico viene presa al momento della creazione di un nuovo processo. Se il nodo in cui il processo sta per essere creato è molto carico, il processo viene creato in un nodo remoto. Tale tipo di migrazione non richiede alcun supporto particolare da parte del kernel.



**Figura 18.8** Migrazione del processo  $P_i$  dal nodo  $N_1$  al nodo  $N_2$ .

La *stabilità* è un aspetto importante nella progettazione di un algoritmo di schedulazione distribuito. Un algoritmo è instabile se, sotto alcune condizioni di carico, il suo sovraccarico non è limitato. Si consideri un algoritmo di schedulazione distribuito che trasferisce un processo da un nodo molto carico verso un altro nodo scelto in modo casuale. Se anche il destinatario è molto carico, il processo potrebbe essere costretto a migrare nuovamente. Sotto condizioni di carico particolarmente elevate, un tale algoritmo potrebbe portare a una situazione simile al fenomeno di *trashing* - il sovraccarico dovuto alla schedulazione sarebbe particolarmente elevato a causa delle frequenti migrazioni, ma i processi trasferiti non ne trarrebbero alcun vantaggio.

Un algoritmo *iniziatto dal mittente* trasferisce un processo senza prelazione da un nodo mittente a un nodo ricevente. Al momento della creazione di un processo in un nodo molto carico, quest'ultimo interroga gli altri nodi per trovarne uno meno carico su cui trasferire il processo. Questo meccanismo rende l'algoritmo di schedulazione instabile su sistemi con carico di elaborazione elevato, poiché un mittente che non riesce a trovare un nodo non molto carico, continua a interrogare gli altri nodi sprecando una parte consistente del suo tempo di CPU. L'instabilità può essere prevenuta limitando il numero di tentativi effettuati dal mittente per trovare un nodo ricevente. Quando tale numero viene superato, il mittente deve rinunciare alla migrazione e il processo viene creato in locale. L'instabilità può nascere anche a causa dell'invio di molti processi verso lo stesso nodo ricevente. In tale contesto, il nodo ricevente diventa a sua volta un nodo mittente che dovrà trasferire alcuni dei processi ricevuti. Questa situazione può essere evitata usando un opportuno protocollo in base al quale un nodo accetta un processo solo finché rimane un ricevente (Problema 18.10).

Se l'algoritmo di migrazione viene, invece, *iniziatto dal ricevente*, ciascun nodo verifica se esso stesso può essere un ricevente ogni volta che uno dei suoi processi termina. Quindi, in caso di esito positivo, il nodo interroga gli altri nodi del sistema per trovarne uno che non desidera diventare un ricevente, e trasferisce un processo da tale nodo verso se stesso. Così facendo, la migrazione dei processi risulta essere necessariamente con prelazione. Con carichi di sistema elevati, il sovraccarico richiesto dalle interrogazioni dei nodi riceventi è limitato, in quanto ciascun ricevente sarà in grado di trovare velocemente un nodo mittente. D'altra parte, con carichi bassi, le continue interrogazioni effettuate dai riceventi non rappresentano un problema, in quanto nel sistema esisteranno sicuramente cicli in cui la CPU è inattiva. È comunque possibile limitare il tempo e le risorse spese per il bilanciamento del carico rinunciando al trasferimento di un processo se non si riesce a trovare un mittente in un numero prefissato di tentativi; tuttavia, va osservato che, per garantire la proprietà di liveness, un ricevente deve ritentare il bilanciamento del carico del sistema a intervalli di tempo regolari.

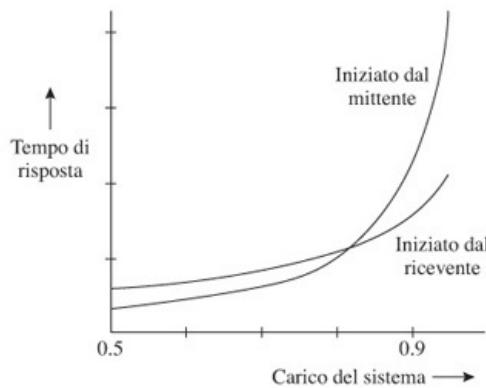
Un ulteriore approccio alla schedulazione distribuita è rappresentato dall'*algoritmo simmetrico*, che include caratteristiche di entrambe le tecniche descritte precedentemente. Esso si comporta come un algoritmo iniziato dal mittente in sistemi con carico basso, mentre agisce come un algoritmo iniziato dal ricevente in sistemi con carico elevato. Ogni nodo mantiene un flag di stato per indicare, in ogni istante, se il nodo debba essere considerato un mittente, un ricevente, oppure nessuno dei due (in questo caso parleremo di nodo OK). Inoltre, i nodi mantengono tre liste, una lista dei mittenti, una dei riceventi e una dei nodi OK, in cui vengono memorizzati, rispettivamente, gli id dei nodi mittenti, dei nodi riceventi e dei nodi OK.

### Algoritmo 18.5 Simmetrico per il bilanciamento del carico

1. Quando un nodo diventa mittente in seguito alla creazione di un processo: imposta il flag di stato al valore "mittente". Se la lista dei riceventi non è vuota, interroga i nodi inclusi in essa, in accordo con il numero massimo di nodi che possono essere interrogati.
  - a. Se il nodo interrogato risponde dicendo di essere un nodo ricevente, trasferisci il processo verso tale nodo. Esamina il carico locale e conseguentemente imposta il flag di stato.
  - b. Altrimenti sposta il nodo interrogato nella lista appropriata in base alla risposta ricevuta.
2. Quando un nodo diventa ricevente in seguito al completamento di un suo processo: imposta il flag di stato al valore "ricevente". Interroga prima i nodi inclusi nella lista dei mittenti, poi quelli inclusi nella lista dei nodi OK, e infine quelli inclusi nella lista dei riceventi, in accordo con il numero massimo di nodi che possono essere interrogati.
  - a. Se il nodo interrogato risponde dicendo di essere un mittente, trasferisci un processo verso di esso. Esamina il carico locale e conseguentemente imposta il flag di stato.
  - b. Altrimenti, sposta il nodo nella lista appropriata in base alla risposta ricevuta.
3. Quando un nodo viene interrogato da un ricevente: sposta il nodo che ha eseguito l'interrogazione nella lista dei riceventi. Invia una risposta contenente il proprio stato corrente.
4. Quando un nodo viene interrogato da un mittente: sposta il nodo che ha eseguito l'interrogazione nella lista dei mittenti. Invia una risposta contenente il proprio stato corrente.
5. Quando un processo viene trasferito dal nodo verso un altro nodo o viceversa: esamina il carico locale e imposta conseguentemente il flag di stato.

L'algoritmo precedente può essere soggetto a instabilità se un elevato numero di processi viene trasferito contemporaneamente verso un nodo ricevente. Per prevenire l'instabilità, un nodo ricevente dovrebbe cambiare il suo flag di stato nel passo 3, anticipando un trasferimento, invece che nel passo 5.

La [Figura 18.9](#) confronta le prestazioni degli algoritmi di schedulazione distribuiti precedentemente descritti. L'algoritmo iniziato dal mittente induce un sovraccarico limitato in un sistema con carico basso, poiché in esso vi sono pochi mittenti. Quindi, un tale sistema è in grado di fornire tempi di risposta buoni ai vari processi. All'aumentare del carico, cresce anche il numero di mittenti e con esso il sovraccarico dovuto all'algoritmo. Su sistemi con carico particolarmente elevato, l'algoritmo iniziato dal mittente diventa instabile a causa del grande numero di mittenti e del ridotto (se non nullo) numero di riceventi. Conseguentemente, il tempo di risposta cresce rapidamente.



**Figura 18.9** Prestazioni degli algoritmi di schedulazione distribuiti.

L'algoritmo iniziato dal ricevente presenta, invece, un sovraccarico maggiore nei sistemi con carico basso, in quanto in essi il numero di riceventi è elevato. Quindi, il

tempo di risposta per tali sistemi è peggiore rispetto a quello dell'algoritmo iniziato dal mittente. Al contrario, in sistemi con carico elevato, l'algoritmo iniziato dal ricevente ha un comportamento nettamente migliore rispetto all'algoritmo iniziato dal mittente. Le prestazioni dell'algoritmo simmetrico assomigliano a quelle dell'algoritmo iniziato dal mittente in sistemi con carico basso, e a quelle dell'algoritmo iniziato dal ricevente in sistemi con carico elevato.

## 18.6 Rilevazione distribuita della conclusione

Un processo mette in relazione varie risorse di un sistema, come per esempio strutture dati del kernel, memoria, eccetera che vengono rilasciate dal kernel quando il processo termina. La conclusione può avvenire a seguito di una chiamata di sistema con cui il processo stesso chiede di essere "ucciso", oppure può essere causata dall'azione di un altro processo. Questi metodi non sono però adeguati per processi appartenenti a un'elaborazione distribuita poiché, in tal caso, i processi potrebbero non essere in grado di decidere né quando dovrebbero terminare né quando dovrebbero uccidere altri processi. Per esempio, si consideri un'elaborazione distribuita i cui processi presentano una relazione di tipo client-server. Il server non è a conoscenza delle richieste che gli potrebbero arrivare, poiché non sa né quali sono i client né se essi hanno già terminato la loro esecuzione o meno. In tali casi, il kernel utilizza tecniche di rilevazione distribuite per verificare se l'intera computazione ha completato la sua esecuzione e, in caso di esito positivo, termina i processi coinvolti e rilascia le risorse che gli erano state allocate.

Per rilevare più facilmente quando la computazione termina, è possibile aggiungere due stati ai processi del nostro modello di sistema. Un processo è in uno *stato passivo* quando non ha alcuna attività da svolgere; un simile processo è dormiente e attende che qualche altro processo gli richieda di eseguire un'attività tramite l'invio di un messaggio. Un processo è invece in uno *stato attivo* quando gli viene richiesto di eseguire qualcosa, come per esempio, effettuare operazioni di I/O, mettersi in attesa di una risorsa o della CPU, o eseguire istruzioni. Lo stato di un processo cambia parecchie volte durante la sua esecuzione. Un processo passivo diventa attivo immediatamente dopo la ricezione di un messaggio, in seguito alla quale invia un acknowledgment al mittente e inizia a processare il messaggio. Un processo attivo, invece, invia subito un acknowledgment in seguito alla ricezione di un messaggio, ma può posticipare a un momento più opportuno la fase in cui il messaggio sarà processato. Un processo attivo diventa passivo quando, dopo aver terminato l'esecuzione del lavoro corrente, non deve effettuare nessun'altra operazione. Nelle considerazioni precedenti, è stato assunto che i messaggi di controllo e i messaggi di base vengano inviati sullo stesso canale.

Un'elaborazione distribuita è considerata conclusa se essa soddisfa la *condizione di terminazione distribuita* (DTC). La DTC è composta dalle seguenti due parti.

1. Tutti i processi della computazione distribuita sono passivi.
2. Non vi sono messaggi di base in transito. (18.1)

La seconda parte della transizione è necessaria poiché un messaggio in transito attiva il corrispondente processo destinatario al momento del suo arrivo. Nel seguito verranno presentati due metodi per determinare se un'elaborazione distribuita soddisfa la condizione DTC.

### **Rilevamento della conclusione di un processo mediante distribuzione del credito**

In questo approccio, proposto da Mattern (1989), a ogni attività reale o potenziale di un'elaborazione distribuita viene associato un valore numerico chiamato *credito*. All'avvio, ogni computazione distribuita viene fornita di un credito  $C$  noto a priori. Il credito viene quindi distribuito tra tutti i processi della computazione. Il meccanismo con cui il credito viene distribuito non è rilevante, l'importante è che ogni processo riceva un credito  $c_i$  maggiore di zero. Successivamente, quando un processo invia un messaggio di base a un altro processo, esso inserisce nel messaggio una parte del suo credito. Anche in questo caso, la quantità di credito inviata è un fattore irrilevante; ciò che conta è che esso abbia un valore diverso sia da zero che dal credito totale del processo mittente. Alla ricezione di un messaggio, il processo ricevente aggiunge il credito contenuto in quest'ultimo prima di iniziare a processarlo. Quando un processo diventa passivo, esso invia tutto il suo credito a uno speciale processo di sistema detto *processo raccoglitore* che accumula tutti i crediti che gli vengono inviati. La terminazione dell'elaborazione distribuita viene rilevata quando il credito accumulato dal processo raccoglitore diventa

uguale al credito iniziale  $C$ . L'algoritmo appena presentato è semplice ed elegante, ma poiché il credito può essere distribuito indefinitamente, è necessario utilizzare per esso una rappresentazione opportuna.

### **Rilevamento della conclusione di un processo mediante elaborazione per propagazione**

Ogni processo che diventa passivo inizia un'elaborazione per propagazione per determinare se la condizione DTC è soddisfatta. In tal modo, ogni processo ha la capacità di rilevare l'eventuale conclusione. L'algoritmo si basa sulle seguenti tre regole.

1. I processi non vengono né creati né distrutti dinamicamente durante l'esecuzione della computazione; ovvero tutti i processi vengono creati nel momento in cui l'elaborazione ha inizio, e rimangono vivi finché essa non termina.
2. I canali di comunicazione inter-processo sono di tipo FIFO.
3. I processi comunicano l'un l'altro in modo sincrono, ovvero il mittente di un messaggio rimane bloccato finché non riceve il corrispondente acknowledgment.

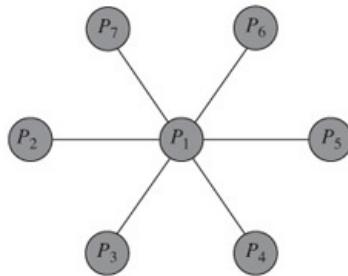
La regola 3 è necessaria per semplificare la verifica della condizione DTC. Infatti, poiché il mittente di un messaggio di base si blocca fino alla ricezione dell'acknowledgment, esso può entrare nello stato di passività solo dopo aver finito il suo lavoro. Pertanto, il messaggio di base inviato da un processo non può essere in transito quando quest'ultimo diventa passivo; inoltre non può esserci nessun messaggio in transito nel sistema quando tutti i processi sono diventati passivi. In tal modo, è sufficiente verificare solo la prima parte della condizione DTC, ovvero basta verificare se tutti i processi sono passivi. L'Algoritmo 18.6 esegue tale verifica utilizzando la computazione per propagazione su un grafo i cui nodi e archi rappresentano, rispettivamente, processi e canali di comunicazione. L'Esempio 18.4 mostra il modo di operare dell'Algoritmo 18.6.

#### **Algoritmo 18.6 Rilevamento distribuito della conclusione**

1. *Quando un processo diventa passivo:* il processo inizia una computazione per propagazione eseguendo le seguenti azioni.
  - a. Invia una query del tipo "devo dichiarare che l'elaborazione si è conclusa?" su tutti gli archi a cui è connesso.
  - b. Ricorda il numero delle query inviate e attende le corrispondenti risposte.
  - c. Dopo aver ricevuto tutte le risposte, nel caso siano tutte positive, il processo dichiara che l'elaborazione è terminata.
2. *Quando un processo riceve una engaging query:* se il processo è nello stato *attivo*, invia una risposta negativa; altrimenti effettua le seguenti azioni.
  - a. Invia query a tutti i suoi archi tranne all'arco da cui ha ricevuto la engaging query.
  - b. Ricorda il numero delle query inviate e attende le corrispondenti risposte.
  - c. Dopo aver ricevuto tutte le risposte, nel caso siano tutte positive, il processo invia una risposta positiva al mittente della engaging query, altrimenti gli invia una risposta negativa.
3. *Quando un processo riceve una nonengaging query.* il processo invia immediatamente una risposta positiva al mittente della query.

#### **Esempio 18.4 - Rilevamento distribuito della conclusione**

Si consideri l'elaborazione distribuita rappresentata in [Figura 18.10](#) e si assuma che solo i processi  $P_1$  e  $P_2$  siano attivi, mentre tutti gli altri siano passivi. A partire da tale situazione, immaginiamo che si verifichino i seguenti eventi.



**Figura 18.10** Esempio di rilevamento distribuito della conclusione.

1. Il processo  $P_2$  diventa passivo, inizia la rilevazione della terminazione e invia una query al processo  $P_1$ .
2. Il processo  $P_1$  invia un messaggio di base al processo  $P_5$  tramite l'arco  $(P_1, P_5)$  e diventa passivo appena possibile.

Gli eventi corrispondenti alla ricezione del messaggio di base da parte di  $P_5$  e all'invio delle query e delle rispettive reply da parte dei vari processi del sistema possono avvenire secondo diverse sequenze. Tra queste, le due riportate di seguito sono particolarmente interessanti.

- Se  $P_1$  ricevesse la query da  $P_2$  prima di diventare passivo, esso invierebbe una reply negativa a  $P_2$ , e quindi quest'ultimo non dichiarerebbe la conclusione dell'elaborazione.
- Se  $P_1$  ricevesse la query da  $P_2$  dopo esser diventato passivo, in base alla regola 3, esso avrebbe dovuto aver già ricevuto un acknowledgment per il messaggio di base inviato a  $P_5$  nel passo 2, e quindi  $P_5$  sarebbe dovuto diventare attivo dopo aver ricevuto il messaggio di  $P_1$  e prima che quest'ultimo diventasse passivo. Di conseguenza, alla ricezione della query di  $P_2$ ,  $P_1$  avrebbe dovuto inviare una query a ciascuno dei processi  $P_3 - P_7$ .  $P_5$  avrebbe inviato una reply negativa a  $P_1$ , che a sua volta avrebbe inviato una reply negativa a  $P_2$ . Di conseguenza,  $P_2$  non avrebbe dichiarato alcuna conclusione.

Se le regole 2 e 3 precedentemente descritte fossero rimosse, in alcune situazioni l'algoritmo presentato in questa sezione non sarebbe in grado di garantire la safety.

La rimozione delle regole 1-3 rende maggiormente complessi gli algoritmi necessari per effettuare il rilevamento distribuito della conclusione. Gli articoli citati nella bibliografia presentano dettagli relativi ad alcuni di tali algoritmi.

## 18.7 Algoritmi di elezione

Alcune funzioni critiche, come per esempio la reintroduzione del token necessario per gli algoritmi basati su token, vengono affidate a un singolo processo, detto *coordinatore*. Tipicamente, viene scelto come coordinatore di una funzione il processo con priorità più elevata all'interno del gruppo di processi interessati alla funzione stessa. Quando un processo si accorge che il coordinatore non ha risposto a una sua richiesta, esso assume che quest'ultimo abbia smesso di funzionare e avvia un *algoritmo di elezione*. Tale algoritmo sceglie come coordinatore il processo con la priorità più elevata tra quelli che sono ancora in esecuzione e comunica a essi la scelta effettuata.

### Algoritmo di elezione per reti con topologia ad anello unidirezionale

L'algoritmo presentato nei paragrafi seguenti assume che tutti i collegamenti dell'anello siano rappresentati da canali di tipo FIFO. Inoltre, si assume che la parte di controllo di un processo che smette di funzionare correttamente continui a inoltrare i messaggi ricevuti sul suo canale di uscita. L'elezione viene effettuata scegliendo il processo con id più alto tra tutti i processi del sistema che non hanno smesso di funzionare. L'algoritmo procede come segue: un processo  $P_i$  avvia l'elezione inviando un messaggio del tipo

(“eleggete me”,  $P_i$ ) sul suo canale di uscita. Ogni processo  $P_j$  che riceve tale messaggio effettua due operazioni - invia un messaggio del tipo (“eleggete me”,  $P_j$ ) e successivamente inoltra il messaggio (“eleggete me”,  $P_i$ ). Di conseguenza,  $P_i$ , dopo aver ricevuto i messaggi di tutti gli altri processi, esamina gli id contenuti in essi ed elegge come coordinatore il processo con id più alto (supponiamo si tratti del processo  $P_{high}$ ). In seguito,  $P_i$  invia nell’anello un messaggio del tipo (“nuovo coordinatore”,  $P_{high}$ ) per informare gli altri processi dell’avvenuta elezione. L’algoritmo assume che durante l’elezione non si verifichino fallimenti, in tal modo si garantisce che il risultato finale sia sempre lo stesso anche quando vengono indette più elezioni contemporaneamente.

A ogni esecuzione, l’algoritmo richiede  $n^2$  messaggi prima di eleggere il coordinatore. Tuttavia, il numero di messaggi può essere ridotto come segue: alla ricezione di un messaggio del tipo (“eleggete me”,  $P_i$ ),  $P_j$  invia solo il messaggio (“eleggete me”,  $P_j$ ), iniziando così una nuova elezione, se il suo id è maggiore di quello di  $P_i$ ; in caso contrario,  $P_j$  invia solo il messaggio (“eleggete me”,  $P_i$ ). In tal modo, solo il processo con id più elevato riceve di ritorno il messaggio “eleggete me” che lui stesso aveva precedentemente inviato, e successivamente inoltra il messaggio (“nuovo coordinatore”,  $P_{high}$ ) per annunciare la sua elezione. Di conseguenza, tutti gli altri processi abbandonano le loro elezioni nel momento in cui ricevono il messaggio (“nuovo coordinatore”,  $P_{high}$ ). Usando questo accorgimento, il numero di messaggi per elezione sarà al massimo  $3n - 1$ . Infatti, il messaggio (“eleggete me”,  $P_i$ ) inviato dal processo che per primo inizia l’algoritmo viene inoltrato al massimo  $n - 1$  volte prima di raggiungere il processo con id più elevato, l’elezione iniziata da quest’ultimo richiede  $n$  messaggi prima di venir completata, e altri  $n$  messaggi sono necessari per informare ogni processo del risultato dell’elezione. Il tempo impiegato per effettuare l’elezione potrebbe pertanto essere pari a  $(3n - 1) \times t_{wc}$ , dove  $t_{wc}$  è il tempo necessario per inviare un messaggio nel caso peggiore.

### **Algoritmo del prepotente**

In base all’algoritmo del prepotente, un processo  $P_i$  inizia una nuova elezione inviando un messaggio del tipo (“eleggete me”,  $P_i$ ) solo ai processi con una priorità maggiore della sua, e avvia un timeout  $T_1$ . Se alla scadenza del timeout,  $P_i$  non ha ricevuto nessuna risposta al suo messaggio, egli assume che i processi con priorità maggiore della sua abbiano smesso di funzionare, e quindi elegge se stesso come coordinatore inviando un messaggio del tipo (“nuovo coordinatore”,  $P_i$ ) a tutti i processi con priorità più bassa della sua. Al contrario, se un processo  $P_j$  a priorità più elevata di  $P_i$  riceve il messaggio (“eleggete me”,  $P_i$ ), egli risponde a  $P_i$  in modo negativo, e successivamente avvia una nuova elezione inviando il messaggio (“eleggete me”,  $P_j$ ) a tutti i processi con priorità maggiore della sua. In seguito a ciò,  $P_i$  abbandona il proposito di diventare il nuovo coordinatore, avvia un nuovo timeout e si mette in attesa di ricevere un messaggio che annuncia l’avvenuta elezione di un processo a priorità maggiore della sua. Se  $P_i$  non riceve tale messaggio entro la scadenza del timeout, egli assume che il processo che sarebbe dovuto diventare coordinatore abbia smesso di funzionare, e inizia una nuova elezione inviando nuovamente il messaggio (“eleggete me”,  $P_i$ ).

L’algoritmo richiede l’invio di un numero di messaggi dell’ordine di  $n^2$  prima di arrivare a eleggere il coordinatore. Inoltre, se il grafo del sistema è completamente connesso e non vi sono fallimenti tra i nodi, il tempo impiegato per ogni elezione può essere dato da  $T_1 + T_2$ , dove  $T_1$  e  $T_2$  rappresentano il valore dei due timeout. In particolare,  $T_1 \geq 2 \times t_{wc}$ , dove  $t_{wc}$  è il tempo richiesto nel caso peggiore per il trasferimento di un messaggio attraverso un arco del grafo, e  $T_2 \geq 3 \times t_{wc}$ . Tuttavia, sarebbe sufficiente considerare  $T_2 \geq 2 \times t_{wc}$  in quanto la trasmissione del messaggio “eleggete me” inviato da un processo a priorità elevata può fungere anche da risposta negativa nei confronti di un messaggio “eleggete me” inviato da un processo a priorità più bassa. Pertanto, possiamo considerare che il tempo impiegato dall’algoritmo sia minore o uguale a  $5 \times t_{wc}$ .

### **Esempio 18.5 - Algoritmi di elezione**

Si consideri un sistema che contiene dieci processi  $P_1, P_2, \dots, P_{10}$ , con priorità 1, ..., 10, dove 10 rappresenta il valore più alto di priorità. Si immagini inoltre che il processo

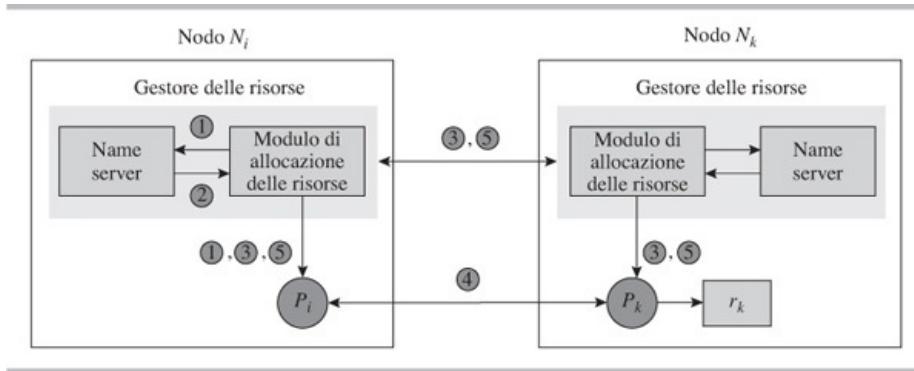
coordinatore,  $P_{10}$ , smetta di funzionare e che  $P_2$  rilevi il fallimento di  $P_{10}$ . In tal caso,  $P_2$  invierà il messaggio (“eleggete me”,  $P_2$ ) ai processi  $P_3 - P_{10}$ . In seguito a ciò, ciascuno dei processi  $P_3 - P_9$  risponderà negativamente alla richiesta di  $P_2$  e inizierà a sua volta una nuova elezione inviando il messaggio “eleggete me” ai processi di priorità più elevata. Se non si verificano altri fallimenti tra i nodi del sistema, i processi  $P_2 - P_8$  riceveranno risposte negative alla loro richiesta, mentre  $P_9$ , non ricevendo alcuna risposta entro il corrispondente timeout, eleggerà se stesso come coordinatore e invierà il messaggio (“nuovo coordinatore”,  $P_9$ ) ai processi  $P_1 - P_8$ . L’elezione richiederà, pertanto, l’invio di un totale di 36 messaggi di tipo “eleggete me”, 28 risposte negative e 8 messaggi di tipo “nuovo coordinatore”. Quindi, il numero totale di messaggi spediti sarà pari a 72.

Va osservato che, se lo stesso sistema fosse organizzato come un anello unidirezionale con gli archi  $(P_i, P_{i+1}) \forall i < 10$  e  $(P_{10}, P_1)$ , sarebbero necessari solo 27 messaggi per effettuare l’elezione del nuovo coordinatore.

## 18.8 Problemi pratici nell’uso degli algoritmi di controllo distribuiti

### 18.8.1 Gestione delle risorse

Quando un processo richiede l’accesso a una risorsa, il modulo che si occupa di allocare le risorse deve trovare la locazione in cui la risorsa risiede nel sistema, determinarne la disponibilità, ed eventualmente allocarne una istanza. La Figura 18.11 mostra schematicamente come avviene l’allocazione di una risorsa. Ogni nodo del sistema contiene un *gestore di risorse* costituito da un name server e da un modulo per l’allocazione delle risorse. Gli archi numerati corrispondono ai passi della seguente procedura.



**Figura 18.11** Allocazione di una risorsa in un sistema distribuito.

1. Quando un processo  $P_i$  desidera usare la risorsa  $ris_j$ , egli invia la coppia  $(ris_j, P_i)$  al gestore delle risorse del proprio nodo, il quale inoltra la richiesta al name server.
2. Il name server identifica la locazione di  $ris_j$  usando il suo nome e i suoi attributi, e inoltra al modulo per l’allocazione delle risorse la tripla  $(r_k, N_k, P_i)$ , dove  $ris_j$  rappresenta la risorsa  $r_k$  nel nodo  $N_k$ .
3. Il modulo per l’allocazione delle risorse verifica se la risorsa  $r_k$  del nodo  $N_k$  è disponibile. In caso affermativo, esso invia a  $P_i$  l’id  $P_k$  del processo che controlla la risorsa e a  $P_k$  un messaggio di richiesta di allocazione da parte di  $P_i$ . Al contrario, se la risorsa non è disponibile, la richiesta viene inserita nella coda delle richieste pendenti. La richiesta sarà quindi soddisfatta nel momento in cui la risorsa diventerà disponibile.
4. Il processo  $P_k$  interagisce con  $P_i$  per soddisfare il servizio richiesto da  $P_i$ .
5. Dopo aver usato la risorsa,  $P_i$  invia una richiesta di rilascio, in seguito alla quale il gestore delle risorse chiede il rilascio della risorsa a  $P_k$  ed, eventualmente, cerca di allocare nuovamente la risorsa per soddisfare una delle richieste pendenti.

L’aspetto rilevante del passo 3 consiste nel garantire che i moduli di allocazione delle

risorse, in esecuzione su nodi diversi del sistema, non interferiscono tra loro. Ciò può essere ottenuto tramite l'uso di algoritmi di mutua esclusione, oppure attraverso l'uso di algoritmi di elezione per definire un coordinatore che si occupi di gestire l'allocazione di tutte le risorse del sistema. L'utilizzo di un algoritmo di mutua esclusione induce un sovraccarico nel sistema per ogni allocazione, mentre l'approccio basato sull'elezione di un coordinatore non è affatto da tale problema. Tuttavia, l'utilizzo del coordinatore richiede l'impiego di un opportuno protocollo per garantire che le informazioni relative allo stato di tutte le risorse siano accessibili al nuovo coordinatore in seguito al fallimento del coordinatore corrente. Un semplice accorgimento per evitare il sovraccarico dovuto sia all'impiego di tecniche di mutua esclusione, sia di algoritmi di elezione, sia di strategie di tolleranza ai guasti, consiste nell'affidare l'allocazione delle risorse di un nodo direttamente al modulo di allocazione delle risorse del nodo stesso. Un tale approccio risulta essere anche particolarmente robusto, in quanto ogni risorsa può essere allocata a un processo per tutto il tempo in cui i nodi che contengono la risorsa e il processo, e il percorso di rete che li unisce, rimangono operativi.

Un ulteriore problema è rappresentato dal fatto che il name server di ogni nodo deve essere aggiornato ogni volta che vengono aggiunte risorse al sistema. Questo problema può essere, tuttavia, risolto in modo simile a quanto avviene per il *domain name service* (DNS) (Paragrafo 16.4.1), per cui solo il name server di un singolo dominio deve essere aggiornato quando viene aggiunta una risorsa.

### 18.8.2 Migrazione dei processi

Le tecniche di migrazione dei processi vengono utilizzate per spostare i processi tra i nodi di un sistema distribuito in modo da garantire il bilanciamento del carico e la riduzione del traffico di rete dovuto all'utilizzo di risorse remote. Le stesse tecniche permettono inoltre di garantire la disponibilità del servizio nel caso in cui un nodo debba essere disattivato per manutenzione. La [Figura 18.8](#) schematizza, in modo ingannevole, la migrazione di un processo come un'operazione semplice; al contrario, in realtà, essa risulta essere abbastanza complessa per svariate ragioni.

Un primo motivo che rende la migrazione più difficile di quanto possa sembrare è dovuto alla necessità di recuperare le informazioni sullo stato del processo. Quest'ultimo è tipicamente costituito dalle seguenti componenti:

- identificatore del processo e dei suoi figli;
- messaggi e segnali pendenti;
- directory di lavoro corrente e descrittori dei file aperti (Paragrafo 13.8).

Visto che le precedenti informazioni sono generalmente distribuite in varie strutture dati memorizzate dal kernel, risulta difficile estrarre lo stato di un processo. Inoltre, gli identificatori del processo e dei suoi figli e i descrittori dei file aperti devono assumere valori unici per il nodo in cui risiede il processo, pertanto, essi potrebbero dover essere modificati quando il processo migra in un nuovo nodo. Ciò crea difficoltà sia per la sincronizzazione tra processi che per le operazioni di I/O. In tale contesto, risulta estremamente importante essere in grado di fornire identificatori univoci a livello globale, come avviene nel Cluster Sun (Paragrafo 16.3), e non per il singolo nodo, e riuscire a mantenere la trasparenza sull'uso delle risorse e dei servizi del sistema (Paragrafo 16.8).

Un secondo problema riguarda gli eventuali messaggi pendenti. Quando viene inviato un messaggio a un processo, il DNS converte il nome del processo (*<host\_name>*, *<id del processo>*) nella coppia (indirizzo IP, *<id del processo>*). Tuttavia, il messaggio potrebbe essere ancora in transito quando il suo processo destinatario viene trasferito in un nodo diverso. In tal caso, è necessario fare in modo che il messaggio venga recapitato all'indirizzo del nuovo nodo. Per garantire ciò, ogni nodo può preservare lo *stato residuo* di tutti i processi che hanno subito una migrazione in uscita. Tale stato contiene l'identificatore del nodo su cui il processo è stato trasferito. In tal modo, i messaggi indirizzati ai processi che sono stati trasferiti su nodi remoti possono essere reindirizzati correttamente, indipendentemente dalla loro migrazione. Tuttavia, lo stato residuo risulta essere un punto critico per l'affidabilità del sistema. Infatti, la perdita dello stato residuo di un processo, o l'impossibilità di accedervi, a causa di un guasto, potrebbe impedire ad alcuni messaggi di essere recapitati. Un meccanismo alternativo all'uso dello stato residuo consiste nell'informare del cambio di locazione di un processo (così come dell'eventuale variazione del suo identificatore) tutti i processi con cui è in atto una

comunicazione. In tal modo, i messaggi possono essere inviati direttamente verso la nuova locazione. Se un messaggio, in transito quando il processo destinatario inizia la migrazione, viene recapitato nel vecchio nodo, quest'ultimo risponderà al mittente "il processo non è più qui". Di conseguenza il mittente inoltrerà il processo verso la nuova locazione.

## Riepilogo

Un *algoritmo di controllo distribuito* è costituito da un insieme di operazioni eseguite dal sistema operativo su vari nodi di un sistema distribuito. Il sistema operativo usa tali algoritmi per (i) evitare il sovraccarico necessario per raccogliere, in modo centralizzato, le informazioni di stato relative a tutte le entità del sistema, (ii) avere un maggior livello di reattività nei confronti degli eventi che si verificano nel sistema, e fornire affidabilità in caso di guasti sui nodi o sui collegamenti. In questo capitolo sono stati descritti algoritmi di controllo per gestire la mutua esclusione, i deadlock, la schedulazione, l'elezione dei coordinatori e il rilevamento della conclusione di applicazioni distribuite.

Le componenti di un algoritmo di controllo distribuito, in esecuzione su differenti nodi, giungono a una decisione condivisa interagendo tra loro tramite scambio di messaggi. Questo modo di operare può portare a ritardi nel raggiungere una decisione, ma l'importante è garantire che prima o poi venga presa una decisione. Inoltre, dal momento che gli algoritmi distribuiti non hanno accesso allo stato di tutte le entità di interesse nello stesso istante, è importante garantire anche che non vengano effettuate azioni errate. Questi due aspetti, relativi alla correttezza degli algoritmi, sono noti, rispettivamente, con i termini *liveness* e *safety*. Queste proprietà devono essere interpretate nel contesto della funzione eseguita dall'algoritmo. Per esempio, nel caso della mutua esclusione, la liveness implica che vengano soddisfatte le condizioni di progresso e attesa limitata descritte nel Paragrafo 6.3.1, mentre la safety richiede che in ogni istante di tempo possa accedere alla sezione critica un solo processo alla volta. Le prestazioni di un algoritmo di controllo distribuito sono misurate in termini del numero di messaggi scambiati dall'algoritmo, e del ritardo necessario per effettuare le azioni richieste.

Gli algoritmi di controllo distribuiti possono utilizzare un *modello fisico* del sistema, oppure un *modello logico* in cui i nodi modellano processi e gli archi i canali di comunicazione su cui vengono scambiati i messaggi. Ogni nodo del modello è a conoscenza del suo stato locale, e interagisce con gli altri nodi per trasmettere le relative informazioni. La correttezza degli algoritmi dipende da come tali informazioni vengono trasferite tra i nodi e da come vengono prese le decisioni, mentre le loro prestazioni dipendono dalla tipologia del modello di sistema utilizzato.

La mutua esclusione viene garantita usando un modello logico completamente connesso e un meccanismo basato su *timestamp*, oppure un approccio basato su un *token* il cui possesso rappresenta il privilegio di poter entrare nella sezione critica. La prima strategia garantisce tempi di decisione ridotti, mentre la seconda richiede un minor numero di messaggi.

Gli algoritmi per la rilevazione distribuita dei deadlock usano un modello logico in cui gli archi rappresentano relazioni di attesa tra i processi, e inviano messaggi speciali su tali archi per identificare l'eventuale presenza di deadlock. Tali algoritmi si basano sull'uso dell'elaborazione per propagazione per raccogliere informazioni sullo stato di tutti i processi di interesse, oppure sull'identificazione di cicli in seguito alla ricezione da parte di un processo dello stesso messaggio precedentemente inviato per rilevare un eventuale deadlock.

La schedulazione distribuita viene effettuata scambiando informazioni di stato tra i nodi del sistema per decidere se trasferire uno o più processi da un nodo all'altro in modo da bilanciare il carico di tutti i nodi.

Un'elaborazione distribuita termina solo quando tutti i processi coinvolti diventano inattivi e non vi sono messaggi a loro destinati in transito nella rete. Il rilevamento distribuito della conclusione può essere effettuato usando l'elaborazione per propagazione. In alternativa, è possibile utilizzare una strategia basata sulla distribuzione di crediti tra i processi. In tal caso, la conclusione viene rilevata quando il credito totale dei processi inattivi è pari alla quantità di credito inizialmente

distribuita.

Infine, gli algoritmi di elezione usano modelli logici e messaggi speciali per identificare ed eleggere come coordinatore il processo con priorità più elevata tra quelli attivi.

## Domande

- 18.1. Indicare se le seguenti affermazioni sono vere o false.
  - a. La parte di controllo di un processo non si blocca mai.
  - b. L'algoritmo di Ricart-Agrawala non porta mai a deadlock se i timestamp sono distinti.
  - c. In un algoritmo per la mutua esclusione basato su token un processo che desidera una risorsa invia la sua richiesta a tutti gli altri processi.
  - d. Nell'elaborazione per propagazione, i processi non inviano reply alle nonengaging query.
  - e. Un algoritmo per la rilevazione dei deadlock centralizzato può identificare deadlock fantasma.
  - f. Un algoritmo di schedulazione iniziato dal mittente diventa instabile con carichi di sistema elevati.
  - g. Un'elaborazione distribuita è terminata se tutti i suoi processi sono in uno stato passivo.
- 18.2. Selezionare l'alternativa appropriata per ciascuna delle seguenti domande:
  - a. Quale delle seguenti proprietà applicate per implementare una sezione critica garantisce la liveness in un algoritmo di mutua esclusione distribuito ([Tabella 6.1](#)).
    - i. La proprietà di progresso.
    - ii. La proprietà di attesa limitata.
    - iii. Le proprietà di progresso e attesa limitata assieme.
    - iv. Nessuna delle alternative (i)-(iii).
  - b. Un processo  $P_1$  inizia un'elaborazione per propagazione inviando query agli altri processi. Un altro processo  $P_k$  nello stesso sistema:
    - i. riceve la query inviata da  $P_1$  esattamente una sola volta;
    - ii. potrebbe non ricevere la query nemmeno una volta;
    - iii. riceve la query almeno una volta, ma può riceverla anche più volte;
    - iv. nessuna delle alternative (i)-(iii).

## Problemi

- 18.1. Descrivere e confrontare le proprietà di liveness di (a) un algoritmo distribuito di mutua esclusione e (b) un algoritmo di elezione.
- 18.2. Si immagini di modificare il passo 2 dell'algoritmo di Ricart-Agrawala in modo che un processo che desidera entrare nella sezione critica non invii risposte positive a nessun altro processo finché esso stesso non sia riuscito a usare la sezione critica. Dimostrare che modificando l'algoritmo in tal modo è possibile che si verifichino deadlock.
- 18.3. Dimostrare la proprietà di safety per l'algoritmo di Maekawa, che utilizza insiemi di richieste di dimensione  $\sqrt{n}$ .
- 18.4. Descrivere un esempio in cui l'algoritmo di Raymond non garantisce l'accesso alla sezione critica in modalità FCFS. (*Suggerimento:* considerare la seguente situazione per l'Esempio 18.2. Il processo  $P_2$  invia una richiesta per entrare nella sezione critica mentre  $P_5$  è ancora nella sua sezione critica.)
- 18.5. Identificare quali sono le query di tipo engaging e non nonengaging nell'algoritmo di Chandy-Lamport per la registrazione di uno stato consistente ([Algoritmo 17.2](#)). Estendere l'algoritmo affinché le informazioni relative allo stato registrato vengano raccolte nel nodo che ha dato inizio alla registrazione.
- 18.6. Dimostrare che il modulo di allocazione delle risorse degli schemi per il rilevamento dei deadlock aspetta-omuori e ferisci-e-aspetta non garantiscono la

- proprietà di liveness se a un processo ucciso viene assegnato un nuovo timestamp al momento della sua reinizializzazione.
- 18.7. Si consideri il seguente algoritmo di edge chasing per il rilevamento di deadlock che si verificano durante la comunicazione inter-processo. Quando un processo si blocca su una richiesta di tipo “ricevi un messaggio”, esso invia una query al processo da cui sta attendendo il messaggio. Se anche quest’ultimo è in attesa su una richiesta di tipo “ricevi un messaggio”, esso a sua volta inoltra la query verso il processo che causa la sua attesa, e così via. Un processo dichiara di avere rilevato un deadlock se riceve la query da lui stesso inviata. Commentare la validità di questo algoritmo nel caso di:
- comunicazione simmetrica;
  - comunicazione asimmetrica.
- 18.8. Mostrare che se nell’algoritmo di Mitchell-Merritt viene omesso l’uso della funzione *inc* nella regola *blocco*, l’algoritmo viola la proprietà di liveness.
- 18.9. Dimostrare la correttezza dell’algoritmo di rilevamento della conclusione basato sulla distribuzione del credito.
- 18.10. Si consideri un algoritmo di schedulazione iniziato dal mittente che utilizza il seguente protocollo per trasferire un processo da un nodo all’altro.
- Il mittente interroga tutti gli altri nodi del sistema alla ricerca di un nodo ricevente.
  - Seleziona un nodo come possibile ricevente, e gli invia un messaggio del tipo “prenota te stesso per il trasferimento di un processo”.
  - Il ricevente del messaggio invia una risposta negativa se nel frattempo ha modificato il suo stato in “non ricevente”. Altrimenti, incrementa di una unità la lunghezza della sua coda di schedulazione e invia una risposta positiva.
  - Il mittente trasferisce il processo quando riceve una risposta positiva.
  - Se invece il mittente riceve una risposta negativa, esso seleziona un altro nodo e ripete i passi 10(b)-10(e).
- Il protocollo precedentemente descritto è in grado di evitare l’instabilità nel caso di carichi di sistema elevati?
- 18.11. Definire le proprietà di liveness e safety di un algoritmo di schedulazione distribuito. (*Suggerimento*: è possibile che non si riesca a ottenere il bilanciamento del carico in un sistema in cui l’algoritmo di schedulazione possiede le proprietà di liveness e di safety?)

## Note bibliografiche

Nel contesto degli algoritmi di controllo distribuiti, Dijkstra e Scholten (1980) e Chang (1982) hanno trattato il modello dell’elaborazione per propagazione, mentre Andrews (1991) ha descritto algoritmi basati su broadcast e token.

Raymond (1989) e Ricart e Agrawala (1981) hanno presentato algoritmi per la mutua esclusione distribuita, mentre Dhamdhere e Kulkarni (1994) hanno proposto un algoritmo di mutua esclusione tollerante ai guasti.

Chandy et al. (1983) hanno adattato l’elaborazione per propagazione nell’ottica di un algoritmo per il rilevamento distribuito dei deadlock (Algoritmo 18.4). In questo contesto, anche Knapp (1987) ha presentato diversi algoritmi, mentre Sinha e Natarajan (1984) hanno proposto un algoritmo basato su edge chasing. Wu et al. (2002) hanno, infine, descritto un algoritmo per il rilevamento distribuito dei deadlock per il modello AND.

La conclusione dell’elaborazione distribuita è stata trattata, invece, da Dijkstra e Scholten (1980), Mattern (1989), e Dhamdhere et al. (1997). L’algoritmo del prepotente per l’elezione distribuita del processo coordinatore è stato proposto da Garcia-Molina (1982). Varie tecniche di migrazione sono state presentate da Smith (1988). Anche Tel (2000) e Garg (2002) hanno trattato algoritmi di elezione e conclusione dell’elaborazione, mentre Attiya e Welch (2004) si sono concentrati solo sulla definizione di algoritmi di elezione. Svariati algoritmi di controllo sono stati proposti anche da Singhal e Shivaratri (1994) e da Lynch (1996).

1. Andrews, G.R. (1991): “Paradigms for process interaction in distributed programs,” *Computing Surveys*, **23**, 1, 49-40.
2. Attiya, H. and J. Welch (2004): *Distributed Computing: Fundamentals, Simulations and Applications*, Springer.

and Advanced Topics, John Wiley, New York.

3. Chandy, K.M., J. Misra, and L.M. Haas (1983): "Distributed deadlock detection," *ACM Transactions on Computer Systems*, **1** (2), 144-152.
4. Chang, E. (1982): "Echo algorithms: depth parallel operations on general graphs," *IEEE Transactions on Software Engineering*, **8** (4), 391-401.
5. Dhamdhere, D.M., and S.S. Kulkarni (1994): "A token based k-resilient mutual exclusion algorithm for distributed systems," *Information Processing Letters*, **50** (1994), 151-157.
6. Dhamdhere, D.M., S.R. Iyer, and E.K.K. Reddy (1997): "Distributed termination detection of dynamic systems," *Parallel Computing*, **22** (14), 2025-2045.
7. Dijkstra, E.W., and C.S. Scholten (1980): "Termination detection for diffusing computations," *Information Processing Letters*, **11** (1).
8. Garg, V.K. (2002): *Elements of Distributed Computing*, Wiley-IEEE, New York.
9. Garcia-Molina, H. (1982): "Elections in distributed computing systems," *IEEE Transactions on Computers*, **31** (1).
10. Knapp, E. (1987): "Deadlock detection in distributed databases," *Computing Surveys*, **19**, (4), 303-328.
11. Lynch, N. (1996): *Distributed Algorithms*, Morgan Kaufmann.
12. Mattern, F. (1989): "Global quiescence detection based on credit distribution and recovery," *Information Processing Letters*, **30** (4), 195-200.
13. Mitchell, D.P., and M.J. Merritt (1982): "A distributed algorithm for deadlock detection and resolution," *Proceedings of the ACM Conference on Principles of Distributed Computing*, August 1984, 282-284.
14. Obermarck, R. (1982): "Distributed deadlock detection algorithm," *ACM Transactions on Database Systems*, **7** (2), 187-202.
15. Raymond, K. (1989): "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems*, **7**, 61-77.
16. Ricart, G., and A.K. Agrawala (1981): "An optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, **24** (1), 9-17.
17. Singhal, M., and N.G. Shivaratri (1994): *Advanced Concepts in Operating Systems*, McGraw-Hill, New York.
18. Sinha, M.K., and N. Natarajan (1984): "A priority based distributed deadlock detection algorithm," *IEEE Transactions on Software Engineering*, **11** (1), 67-80.
19. Smith, J.M. (1988): "A survey of process migration mechanisms," *Operating Systems Review*, **22** (3), 28-40.
20. Tel, G. (2000): *Introduction to Distributed Algorithms*, 2nd ed., Cambridge University Press, Cambridge.
21. Wu, H., W. Chin, and J. Jaffer (2002): "An efficient distributed deadlock avoidance algorithm for the AND model," *IEEE Transactions on Software Engineering*, **28**, 1, 18-29.

---

# CAPITOLO 19

## File system distribuiti

---

### Obiettivi di apprendimento

- Funzionalità di un file system distribuito
- Trasparenza
- Semantica della condivisione dei file
- Tolleranza ai guasti
- File server stateless e stateful
- Prestazioni di un file system distribuito

Gli utenti di un file system distribuito (FSD) richiedono gli stessi livelli di affidabilità e prestazioni e gli stessi vantaggi di un file system tradizionale. Il vantaggio di usare un file system distribuito si basa su due fattori chiave: *trasparenza* e *semantica della condivisione dei file*. La *trasparenza* di un FSD consente agli utenti di ignorare la posizione dei file sui nodi e sui dischi del sistema. La *semantica della condivisione dei file* specifica, invece, se e come gli effetti delle modifiche apportate da un processo a un file sono visibili a eventuali altri processi che stanno usando lo stesso file.

Un processo e il file a cui esso accede possono trovarsi in nodi diversi del sistema distribuito, pertanto, un guasto su uno dei due nodi o sul cammino che li unisce potrebbe influenzare l'accesso al file. I file system distribuiti garantiscono l'affidabilità tramite la *replicazione dei file*, e l'uso di *file server stateless* che, non preservando informazioni di stato relative all'accesso ai file, permettono di minimizzare l'impatto di un eventuale malfunzionamento del file server finché è in corso l'accesso a un file.

Il tempo di risposta delle operazioni eseguite sul file system è influenzato dalla latenza della rete durante l'accesso ai file remoti. Pertanto, per ridurre il traffico di rete durante l'accesso a un file, vengono usate tecniche di *file caching*. Un altro aspetto relativo alle prestazioni è rappresentato dalla *scalabilità* - i tempi di risposta non dovrebbero degradarsi all'aumentare della dimensione del sistema distribuito. La scalabilità viene affrontata attraverso tecniche che permettono di localizzare su un *cluster* le operazioni su un file. Un cluster è costituito da un gruppo di calcolatori collegati da una LAN ad alta velocità.

In questo capitolo tratteremo le tecniche per l'implementazione di un FSD che permettano di garantire facilità d'uso, affidabilità e buone prestazioni. Le operazioni relative a un FSD verranno illustrate mediante alcuni casi di studio.

### 19.1 Problematiche di progetto in un file system distribuito

Un file system distribuito (FSD) memorizza i file degli utenti in vari nodi del sistema. Pertanto, accade spesso che un processo e il file a cui esso fa accesso si trovino su nodi diversi. Tale situazione ha tre probabili conseguenze:

- un utente potrebbe aver bisogno di conoscere la topologia del sistema per poter aprire e accedere a file posizionati nei vari nodi del sistema;
- l'accesso a un file da parte di un processo potrebbe essere corrotto se si manifesta un guasto nel nodo del processo, nel nodo del file o nel cammino che li unisce;
- le prestazioni del file system possono essere scarse a causa del traffico di rete richiesto per l'accesso ai file.

La [Tabella 19.1](#) riassume tre problematiche di progetto, discusse nei paragrafi successivi, che devono essere affrontate per evitare le precedenti conseguenze.

| Problematica         | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Trasparenza          | Un file system con elevata trasparenza permette agli utenti di non dover conoscere la locazione dei file nel sistema per potervi accedere. Il concetto di trasparenza è contraddistinto da due aspetti. La <i>trasparenza della locazione</i> implica che il nome di un file non dovrebbe rilevare la sua locazione all'interno del file system. L' <i>indipendenza dalla locazione</i> implica invece che dovrebbe essere possibile cambiare la posizione di un file senza dover cambiare il suo nome. |
| Tolleranza ai guasti | Un guasto su un nodo o su un canale di comunicazione può corrompere eventuali operazioni di accesso a file. Un guasto può influenzare sia la disponibilità del servizio offerto dal file system che la consistenza dei dati e dei metadati, cioè i dati di controllo, contenuti in esso. Un FSD dovrebbe adottare tecniche speciali per evitare le precedenti situazioni.                                                                                                                               |
| Prestazioni          | La latenza della rete è un fattore dominante per il tempo di accesso a file in un FSD. Essa influenza sia l'efficienza che la scalabilità di un FSD. Quindi, un FSD dovrebbe usare opportune tecniche per ridurre il traffico di rete generato durante l'accesso a un file.                                                                                                                                                                                                                             |

**Tabella 19.1** Problematiche di progetto di un file system distribuito.

### Trasparenza

Come visto nel Paragrafo 13.9.1, per identificare la posizione di un file, il file system esegue un'operazione nota come *pathname resolution*, ovvero risoluzione del percorso. Due problematiche rilevanti nel contesto di un file system distribuito sono: quante informazioni sulla locazione di un file dovrebbero essere rivelate dal suo pathname, e se un FSD può cambiare la locazione di un file per ottimizzare le prestazioni relative al suo accesso. Queste problematiche si legano a due aspetti fondamentali del concetto di trasparenza.

- *Trasparenza della locazione*: il nome di un file non dovrebbe rilevare informazioni sulla sua posizione.
- *Indipendenza dalla locazione*: il file system dovrebbe essere in grado di cambiare la posizione di un file senza dover cambiare il suo nome.

La trasparenza della locazione fornisce agli utenti, e alle elaborazioni, il vantaggio di non dover conoscere la posizione del file. L'indipendenza dalla locazione permette invece al file system di ottimizzare le sue prestazioni. Per esempio, se l'accesso ai file memorizzati in un singolo nodo causa congestione della rete, degradando così le prestazioni del sistema, il FSD può spostarne alcuni su altri nodi. Questa operazione è detta *migrazione di file*. L'indipendenza dalla locazione può anche essere usata per migliorare l'utilizzo dei dispositivi di memorizzazione del sistema. Discuteremo tali aspetti in dettaglio nel Paragrafo 19.2.

### Tolleranza ai guasti

Un guasto può interrompere a metà le operazioni di accesso a un file, e a causa di ciò è possibile perdere la consistenza dei dati del file e dei metadati (ovvero i dati di controllo) del file system. Per proteggere la consistenza dei dati, un FSD può utilizzare tecniche di *journaling* come fatto nei file system convenzionali, oppure può usare un modello di file server *stateless*; in tal caso, diventa superfluo proteggere la consistenza di dati e metadati. La protezione dei dati viene fornita tramite una *semantica delle transazioni*, utile per implementare *transazioni atomiche*, in modo tale che la tolleranza ai guasti sia garantita dall'applicazione stessa. Gli aspetti relativi alla tolleranza ai guasti verranno discussi nel Paragrafo 19.4.

### Prestazioni

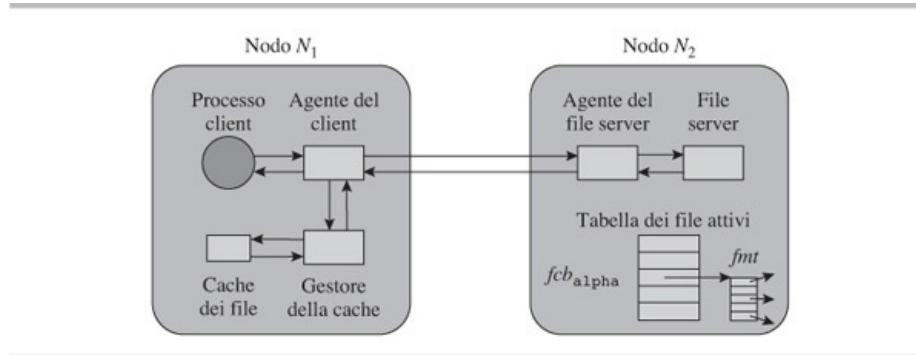
Le prestazioni di un FSD vanno misurate su due aspetti: efficienza e scalabilità. In un sistema distribuito, la *latenza della rete* risulta essere un fattore dominante per l'efficienza delle operazioni effettuate sul file system. La latenza di rete, tipicamente è

maggiore del tempo richiesto per accedere a un record del file, pertanto, a differenza della latenza del dispositivo di I/O, non può essere mascherata dal blocking o dal buffering dei record (Paragrafi 14.8 e 14.9). Un FSD utilizza *il caching dei file*, per tenere una copia del file remoto nel nodo del processo che vi sta accedendo. In tal modo, gli accessi al file non causano traffico di rete, sebbene sia necessario prevenire l'invecchiamento dei dati presenti nella cache attraverso tecniche di *coerenza della cache*.

La *scalabilità* di un FSD richiede che i tempi di risposta non degradino quando la dimensione del sistema aumenta a causa dell'aggiunta di nuovi nodi e utenti. Un sistema distribuito è composto da vari *cluster*, ovvero gruppi di calcolatori connessi da una LAN ad alta velocità (Paragrafo 16.2), pertanto, memorizzare una copia di un file in un cluster permette l'accesso a tale file da parte dei calcolatori appartenenti al cluster in modo indipendente dalla dimensione del sistema. La stessa strategia permette di ridurre il traffico di rete. Entrambi questi aspetti aiutano a migliorare la scalabilità delle prestazioni di un FSD. Quando molti processi accedono in parallelo allo stesso file, è necessario, inoltre, usare tecniche di *locking distribuito* per garantire che le operazioni di sincronizzazione degli accessi siano scalabili all'aumentare della dimensione del sistema. Tratteremo le tecniche per migliorare le prestazioni di un FSD nel Paragrafo 19.6.

### 19.1.1 Funzionalità di un file system distribuito

La [Figura 19.1](#) mostra lo schema semplificato del meccanismo di accesso ai file in un FSD, noto come modello di *accesso a file remoto*. Nella figura, un processo, detto *processo client* o semplicemente *client*, in esecuzione nel nodo  $N_1$ , detto *nodo client*, apre il file con pathname ... alpha. A fronte della risoluzione del pathname, l'FSD rileva che il file è posizionato nel nodo  $N_2$ , e quindi utilizza il meccanismo mostrato in [Figura 19.1](#). Il componente del file system nel nodo  $N_2$  è detto *file server*, mentre  $N_2$  è noto come *nodo server*. Altri nodi coinvolti nella risoluzione del pathname o che potrebbero essere coinvolti nel trasferimento dei dati da  $N_1$  a  $N_2$  verranno chiamati *nodi intermedi*.



**Figura 19.1** Operazioni di base per l'accesso a un file in un FSD.

L'accesso al file è fornito tramite processi stub detti *agente del file server (file server agent)* e *agente del client (client agent)*, che implementano un approccio simile a quanto avviene per RPC (Paragrafo 16.5.2). Quando il client desidera aprire il file, la richiesta viene inoltrata all'agente del client che la comunica all'agente del file server del nodo  $N_2$ .

Quest'ultimo, a sua volta, informa il file server che apre il file  $\alpha$  e inserisce un blocco  $fcb_{\alpha}$  per  $\alpha$  nella cache. Se il file caching non viene utilizzato, le operazioni di lettura e scrittura su  $\alpha$  vengono implementate tramite l'invio di messaggi tra l'agente del client e l'agente del file server. Il nodo  $N_2$  mantiene un buffer di I/O per il file, e al client viene passato un record alla volta. Quando, invece, viene utilizzato il *file caching*, le richieste di lettura e scrittura vengono inoltrate al *gestore della cache*, che verifica se i dati richiesti possono essere letti/scritti dalla/nella *cache*. Il gestore della cache interagisce con l'agente del server tramite messaggi quando ha bisogno di trasferire dati tra la cache e il file. Per motivi di efficienza, l'agente del client e il gestore della cache sono tipicamente inglobati in un unico componente.

## 19.2 Trasparenza

In un file system convenzionale, l'utente identifica un file tramite il suo pathname e, pertanto, è consapevole che il file appartiene a una specifica directory. Tuttavia, non conosce la posizione della directory all'interno del sistema. La posizione del file nel disco è memorizzata nel campo *location info* dell'elemento della directory che lo contiene. Tale approccio è adeguato anche per fornire la *trasparenza dalla locazione* in un FSD - un utente usa il pathname per accedere a un file, e l'FSD ottiene la posizione del file dell'elemento della directory che lo contiene. L'FSD può decidere di tenere tutti i file di una directory nello stesso nodo oppure su nodi diversi. Nel primo caso, i metadati sarebbero identici a quelli di un file system convenzionale. Nel secondo caso, il campo *location info* relativo all'elemento delle directory di un file contiene la coppia (id nodo, locazione).

Per garantire l'*indipendenza dalla locazione* è invece necessario che le informazioni contenute nel campo *location info* degli elementi di una directory possano essere modificate dinamicamente. In tal modo, è possibile per un FSD cambiare la posizione di un file subito dopo aver inserito le informazioni sulla nuova posizione nel campo *location info*. Analogamente, devono essere modificate le corrispondenti informazioni su tutti i link del file (Paragrafo 13.4.2). Per semplificare tali modifiche, l'FSD può usare il seguente approccio: ad ogni file viene assegnato un id unico in tutto il sistema che viene memorizzato nell'elemento della directory relativo al file. L'FSD mantiene una struttura dati specifica per memorizzare le coppie (id del file, locazione del file). In tal modo, l'FSD dovrà modificare solo una coppia di questa struttura dati nel caso in cui venga modificata la posizione del file, indipendentemente dal numero dei suoi link.

La maggior parte dei file system distribuiti garantisce la trasparenza della locazione, ma non l'indipendenza dalla locazione. Quindi, i file non possono essere spostati tra i nodi. Tale restrizione non consente all'FSD dell'opportunità di ottimizzare le prestazioni delle operazioni di accesso ai file.

## 19.3 Semantica della condivisione dei file

La semantica della condivisione dei file determina il modo in cui risulta visibile l'effetto delle modifiche effettuate concorrentemente su uno stesso file da più utenti. Si ricorda dal Paragrafo 13.10 che tutti i client che accedono in modo concorrente a un *file modificabile con singola immagine* hanno la stessa visione del suo contenuto. Pertanto, le modifiche fatte da un client sono visibili immediatamente agli altri client che accedono allo stesso file. Al contrario, i client che accedono a un *file modificabile con immagini multiple* possono avere viste diverse del suo contenuto. In tal caso, quando l'accesso viene completato, il file system può ricreare la consistenza tra le varie viste per creare un'immagine unica del file, oppure può supportare l'esistenza di più di una versione del file. In quest'ultimo caso, è necessario garantire che una successiva apertura del file da parte di un qualunque client fornisca accesso alla versione corretta del file. La [Tabella 19.2](#), riassume le caratteristiche principali di tre tipi di semantica di condivisione dei file - *semantica Unix*, *semantica di sessione* e *semantica delle transazioni*.

| Semantica                   | Descrizione                                                                                                                                                                                                                                                                                                                                                                |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Semantica Unix              | Permette di implementare il paradigma del <i>file modificabile con singola immagine</i> . L'effetto di una scrittura su un file da parte di un client è visibile immediatamente agli altri client. I client possono eventualmente condividere il puntatore al prossimo record del file.                                                                                    |
| Semantica di sessione       | Permette di implementare il paradigma del <i>file modificabile con immagini multiple</i> . Solo i client appartenenti a una <i>sessione</i> condividono la stessa immagine del file. Gli aggiornamenti effettuati da un client sono visibili immediatamente agli altri client della stessa sessione; sono visibili agli altri client solo dopo la chiusura della sessione. |
| Semantica delle transazioni | L'accesso ai file effettuato da un client viene implementato come una transazione atomica, pertanto o tutte le operazioni vengono eseguite oppure nessuna di esse va a buon fine. Questo approccio semplifica le tecniche utilizzate per garantire la tolleranza ai guasti.                                                                                                |

**Tabella 19.2** Semantica della condivisione dei file.

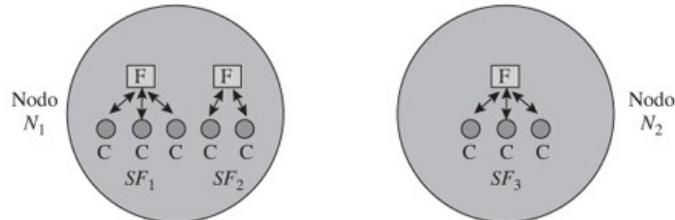
### Semantica Unix

Si ricorda dal Paragrafo 13.10 che la semantica Unix supporta il paradigma del file modificabile con singola immagine. Pertanto, gli aggiornamenti effettuati da un client sono immediatamente visibili agli altri client. Eventualmente, i client possono condividere il puntatore di accesso al file. Tale caratteristica è utile se i client accedono al file in modo congiunto. La semantica Unix è efficiente e semplice da implementare su un file system convenzionale; tuttavia, come trattato nel Paragrafo 19.5.2, essa richiede un notevole overhead per mantenere la coerenza della cache in un FSD che fa uso di file caching.

### Semantica di sessione

Con il termine *sessione* si intende un insieme di client collocati in uno stesso nodo del sistema. I client di una sessione condividono la stessa immagine immutabile di un file. Pertanto, il risultato di una scrittura effettuata da un processo client è visibile immediatamente agli altri client appartenenti alla stessa sessione, ma non a quelli appartenenti ad altre sessioni.

La creazione delle sessioni e la visibilità delle immagini dei file è governata dalle seguenti regole: sia  $SF_i$  la sessione relativa un insieme di client che accedono al file F. Quando un altro client dello stesso nodo apre F, l'FSD gli permette di unirsi alla sessione  $SF_i$  solo se nessuno dei client già presenti in  $SF_i$  ha chiuso F dopo aver eseguito un'operazione di scrittura. In caso contrario, l'FSD crea una nuova sessione. Quando, invece, il file F viene aperto da un client posizionato in un nodo differente, l'FSD inizia subito una nuova sessione. La [Figura 19.2](#) mostra tre sessioni  $SF_1$ ,  $SF_2$  e  $SF_3$  per il file F. Due di esse sono state aperte nel nodo  $N_1$  poiché l'FSD crea una nuova versione del file ogni volta che un client chiude il file dopo averlo modificato.



**Figura 19.2** Esempio di sessioni aperte in un file system che implementa la semantica di sessione.

Tuttavia, la semantica di sessione non specifica le regole con cui decidere quale versione

del file deve essere aperta al momento della creazione di una nuova sessione. Pertanto, ogni file system può implementare questo aspetto in modo diverso. Di conseguenza, le applicazioni che usano la semantica di sessione potrebbero non essere portabili. Va infine osservato che la semantica di sessione è semplice da implementare in un FSD che fa uso di file caching, poiché le modifiche effettuate in un file non sono visibili ai client degli altri nodi.

### ***Semantica delle transazioni***

Adottando la semantica delle transazioni, le operazioni di accesso ai file effettuate da processi client vengono eseguite come transazioni atomiche. Questo tipo di semantica può essere implementata adottando i seguenti accorgimenti: (i) le operazioni *open* e *close* per l'apertura e chiusura di un file vengono trattate come se fossero l'inizio e la fine di una transazione, (ii) l'accesso mutuamente esclusivo ai file viene garantito impostando un *lock* sul file stesso, (iii) gli aggiornamenti dei file vengono eseguiti. Pertanto, in ogni istante, l'accesso a un determinato file può essere effettuato da un solo client alla volta. Inoltre, l'aggiornamento di file da parte di un client deve rispettare la proprietà "tutto o niente", in base alla quale, se non è possibile apportare tutte le modifiche richieste dal client, il file rimarrà invariato. Tale proprietà garantisce in ogni momento la consistenza del file. Di conseguenza, di fronte al verificarsi di un guasto, un client può semplicemente rieseguire l'operazione di accesso al file che eventualmente fosse stata interrotta dal guasto. Inoltre, va osservato che il meccanismo di locking garantisce all'FSD di non dover gestire accessi concorrenti a un file da parte di più client.

## **19.4 Tolleranza ai guasti**

L'affidabilità di un file system è caratterizzata da vari aspetti. Innanzitutto, un file deve essere *robusto*, ovvero deve poter tollerare i guasti. In secondo luogo, esso deve essere *ripristinabile*, cioè deve essere possibile riportare il file nello stato precedente a un guasto. Infine, un file deve essere *disponibile* indipendentemente dal verificarsi di un guasto nel sistema, e quindi in ogni istante deve esistere una copia del file accessibile a eventuali client.

Robustezza e possibilità di ripristino dipendono, rispettivamente, da come vengono memorizzati i file, e da come viene eseguito il loro backup, mentre la disponibilità dipende dalla modalità con cui si effettua l'apertura e l'accesso ai file. Va osservato inoltre, che i precedenti aspetti sono indipendenti tra loro. Pertanto, per esempio, un file può essere ripristinabile a uno stato precedente senza possedere la caratteristica della robustezza o senza che la sua disponibilità sia costantemente garantita. La robustezza si ottiene usando tecniche di memorizzazione dei dati affidabili, per esempio, le strategie di mirroring usate nei RAID di livello 1 (Paragrafo 14.3.5). Ripristino e disponibilità di accesso si ottengono invece attraverso approcci particolari descritti in questo paragrafo.

Se si verifica un guasto nel server o nei nodi intermedi durante l'apertura di un file, l'operazione con cui si esegue la risoluzione del pathname potrebbe non essere effettuata correttamente. È possibile, tuttavia, adottare tecniche di tolleranza che garantiscono la disponibilità del file anche in caso di guasto. Per far ciò, l'FSD mantiene molte copie delle informazioni richieste per la risoluzione del pathname, e molte copie di ogni file. Se una di tali copie diventa inaccessibile a causa di un guasto, l'FSD ne utilizza semplicemente un'altra. Tuttavia, le tecniche che garantiscono la disponibilità dei file diventano molto complesse e dispendiose quando si desidera fornire tolleranza ai guasti durante l'accesso a un file. Pertanto, esistono pochi file system che gestiscono questa tipologia di guasti.

I guasti che avvengono su server o client durante le operazioni di accesso possono portare alla perdita delle informazioni di stato. Come vedremo nel Paragrafo 19.4.3, è possibile progettare un file server in modo tale che il suo funzionamento non venga compromesso nel caso in cui le informazioni di stato vadano perse a causa di un guasto. Tuttavia, se i client non usano tecniche particolari per proteggersi contro la perdita dello stato, un malfunzionamento, in seguito a un guasto, può portare a situazioni di inconsistenza. L'unica soluzione in caso di malfunzionamento di un nodo client è rappresentata dall'uso di file server che adottano la *semantica delle transazioni*. In tal modo, è possibile eventualmente ripristinare un file allo stato in cui si trovava prima che il client iniziasse l'accesso. Al contrario, un guasto in un nodo intermedio non influenza l'accesso al file se il sistema di comunicazione è sufficientemente flessibile. Quindi, generalmente, i file system non gestiscono tali guasti.

La [Tabella 19.3](#) riassume le tecniche di tolleranza ai guasti usate nei file system distribuiti. La replicazione dei file e il caching delle directory permettono di gestire guasti che si verificano su file server e nodi intermedi durante le operazioni di apertura di un file. I file server stateless (ovvero, privi di stato) permettono invece di gestire guasti che si verificano sul file server durante l'accesso a un file. I prossimi paragrafi descrivono queste tecniche.

| Tecnica                 | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Caching delle directory | Una <i>cached directory</i> è una copia locale di una directory memorizzata su un sito remoto. Il suo utilizzo permette all'FSD di tollerare guasti nei nodi intermedi coinvolti nell'operazione di risoluzione del pathname.                                                                                                                                                                                                                                                                                                                                             |
| Replicazione dei file   | Per garantire la disponibilità di servizio nel file system distribuito, vengono memorizzate molteplici copie di uno stesso file. La consistenza tra tali copie viene preservata usando tecniche speciali. Per esempio, la tecnica nota come <i>copia primaria</i> permette ai client di accedere in lettura a qualunque copia di un file, mentre l'accesso in scrittura è permesso solo sulla copia primaria. Le modifiche effettuate alla copia primaria si propagano verso tutte le altre copie. Questa tecnica semplifica notevolmente il controllo della concorrenza. |
| File server stateless   | Un file server convenzionale preserva informazioni di stato relative alle operazioni effettuate su un file all'interno del blocco di controllo. Al contrario, un <i>file server stateless</i> non mantiene alcuna informazione. Pertanto, non è influenzato da guasti che causano la perdita delle informazioni di stato.                                                                                                                                                                                                                                                 |

**Tabella 19.3** Tecniche di tolleranza ai guasti in un FSD.

#### 19.4.1 Disponibilità

Si dice che un file è disponibile, se è possibile accedere ad almeno una delle sue copie. La capacità di aprire un file dipende dalla possibilità di completare la risoluzione del suo pathname, ovvero, dal corretto funzionamento del nodo server e di tutti i nodi intermedi coinvolti nella risoluzione del pathname. Al contrario, la capacità di accedere a un file dipende esclusivamente dal corretto funzionamento del nodo client e del nodo server, in quanto l'esistenza di un percorso che li unisce è garantita dalla flessibilità della rete.

Si consideri il pathname `a/b/c/d`, dove le directory `a`, `b`, `c` e il file `d` sono memorizzati, rispettivamente, nei nodi A, B, C e D. È possibile utilizzare due approcci per risolvere tale pathname. Un primo approccio prevede che il FSD invii il suffisso `b/c/d` al nodo B quando rileva che la directory `b` è memorizzata su B. Analogamente, quando B rileva che `c` è contenuto nella directory `b` del nodo C, esso invia `c/d` a C, e così via fino alla completa risoluzione del pathname. L'approccio alternativo prevede, invece, che l'FSD effettui la risoluzione di tutte le componenti del pathname nel nodo client. Quando il client rileva che un componente del pathname è il nome di una directory contenuta in un nodo remoto, effettua una copia locale della directory e la usa per continuare la risoluzione del pathname. In tal modo, durante la risoluzione del pathname, tutte le directory vengono copiate nel nodo client. Come vedremo in seguito, questi due approcci hanno implicazioni diverse rispetto al livello di disponibilità garantito dal file system. In entrambi gli approcci, l'accesso ai dati contenuti in un file non coinvolge i nodi intermedi utilizzati durante la risoluzione del pathname. Pertanto, l'accesso a un file non è influenzato da eventuali guasti che si verificano su tali nodi dopo l'apertura del file.

##### Caching delle directory

Quando il pathname di un file a cui si desidera accedere si estende per molti nodi, può verificarsi una situazione anomala. Consideriamo l'esempio precedente, e supponiamo che il nodo `c` smetta di funzionare mentre è in corso l'accesso al file `d`, precedentemente aperto usando il pathname `a/b/c/d`. In tale situazione, un eventuale altro client in esecuzione nel nodo A non potrebbe aprire il file `z` contenuto in `d` tramite il pathname `a/b/c/z`, a causa del malfunzionamento del nodo di `c`. Pertanto, `z` non potrebbe essere utilizzato benché il suo accesso coinvolga gli stessi nodi client e server usati per il file `d`.

L'unico modo per evitare tale anomalia consiste nel memorizzare in una cache del nodo client gli accessi effettuati a directory remote durante la risoluzione del pathname. Nel caso del pathname `a/b/c/d`, ciò implica che l'FSD dovrebbe effettuare il caching delle directory `a/b` e `a/b/c` nel nodo A. In tal modo, durante la successiva risoluzione dei pathname che coinvolgono i prefissi `a/b` e `a/b/c`, l'FSD potrà utilizzare i dati delle directory precedentemente memorizzati nella cache. Pertanto, sarà in grado di risolvere il pathname `a/b/c/z` senza accedere ai nodi B e C. Tuttavia, le informazioni memorizzate nella cache possono diventare vecchie a causa della creazione o della cancellazione di directory in alcuni dei nodi intermedi, e quindi è necessario utilizzare un particolare protocollo per l'aggiornamento della cache. Tratteremo tale argomento nel prossimo paragrafo.

### **Replicazione dei file**

L'FSD esegue la replicazione di un file in modo trasparente ai suoi client. Tuttavia, la replicazione di un file, con elevata probabilità di essere aggiornato spesso, richiede un'accurata analisi del rapporto tra costo e complessità del protocollo di aggiornamento, e implicazioni relative all'uso efficiente del file. Il protocollo di commit a due fasi può essere utilizzato per aggiornare, nello stesso istante, tutte le copie di un file. In tal modo, non vi possono essere, contemporaneamente, copie vecchie e copie aggiornate e, pertanto, un client necessita di un'unica copia del file per poter effettuare un accesso in lettura. Tuttavia, un'operazione di aggiornamento può invece essere ritardata qualora alcune copie siano in uso da parte di altri processi o inaccessibili a causa di guasti. Pertanto, le tecniche alternative al commit a due fasi mirano a velocizzare le operazioni di aggiornamento riducendo il numero di copie che devono essere aggiornate.

Nell'approccio basato su *copia primaria*, gli aggiornamenti sono effettuati su una singola copia, detta appunto primaria. Nel momento in cui la copia primaria viene aggiornata, le altre copie vengono invalidate. Le modifiche apportate alla copia primaria vengono replicate sulle altre copie solo quando queste ultime vengono nuovamente referenziate. In alternativa, l'FSD può usare per i dati replicati una strategia simile al protocollo lettori-scrittori. In tal caso, per garantire efficienza e tolleranza ai guasti, i quorum di lettura e scrittura devono essere più bassi possibile. Per identificare la copia più recente nel quorum di lettura, è sufficiente confrontare i timestamp assegnati a ogni copia per indicare l'istante in cui è stata effettuata l'ultima modifica.

Le tecniche di replicazione dei file funzionano ancora meglio se l'eventuale utilizzo di una copia vecchia risulta essere comunque significativo. In tal caso, infatti, le modifiche non hanno bisogno di essere propagate a tutte le copie immediatamente. Tale assunzione risulta particolarmente utile nel caso della replicazione delle directory. Infatti, se si accede a un copia vecchia di una directory, possono verificarsi due tipi di problemi: (i) un file non possiede un elemento nella directory anche se è già stato creato, (ii) esiste ancora un elemento nella directory relativo a un file che è già stato cancellato. Nel primo caso, il file server può consultare immediatamente la copia primaria per verificare se il file esiste veramente, ed eventualmente terminare il processo. Nel secondo caso, invece, il problema emerge quando si cerca di eseguire un'operazione di lettura o scrittura sul file cancellato, ma anche in tal caso è possibile terminare il processo.

### **19.4.2 Guasti su client e server**

Come descritto nel Paragrafo 13.8, un file system convenzionale utilizza metadati, quali per esempio il file control block (FCB), per memorizzare le informazioni relative allo stato di un'operazione di accesso a file. Tali informazioni stabiliscono un contesto implicito tra il file system e il client, e il loro utilizzo rende efficienti le operazioni di lettura e scrittura su file. Per esempio, durante la lettura sequenziale di un file, il file system accede all'FBC di quest'ultimo per ottenere l'identificatore del prossimo byte da leggere, e accede alla file map table (FMT) per recuperare l'indirizzo del corrispondente blocco su disco; non è necessario accedere all'elemento della directory relativo al file per ottenere l'indirizzo della sua FMT. Un file system che mantiene informazioni sullo stato delle operazioni di accesso a file è detto *file system stateful*. Analogamente a quanto avviene su un file system convenzionale, per garantire prestazioni elevate, il nodo server di un FSD può salvare in memoria i file control block e la tabella dei file attivi (AFT). Tuttavia, l'utilizzo di un FSD stateful diventa problematico in caso di malfunzionamento del nodo server o di un client.

Nel caso di errore nel client, l'eventuale operazione di accesso al file in atto deve

essere arrestata e il file deve essere ripristinato al suo stato precedente. In tal modo, al riavvio, il client potrà rieseguire l'accesso al file partendo da una situazione di consistenza. D'altro canto, il server, per poter gestire la richiesta di accesso al file, deve effettuare il commit delle risorse quali l'FCB e i buffer di I/O. Tali risorse devono essere rilasciate nel caso in cui un'operazione di accesso al file venga terminata, altrimenti rimarranno occupate indefinitamente. Per gestire tali problematiche è possibile adottare una strategia basata sul concetto di circuito virtuale (Paragrafo 16.6.5). In base a tale strategia, client e file server condividono un circuito virtuale che "detiene" le azioni di accesso ai file e le risorse coinvolte (per esempio i metadati del file server). Nel caso di malfunzionamento del client o del server, il corrispondente circuito virtuale viene interrotto rendendo, così, orfane le azioni e le risorse detenute. Quindi, le azioni devono essere annullate, mentre le risorse devono essere rilasciate. Per far ciò è possibile usare un protocollo client-server che implementa la semantica delle transazioni. Se un FSD non supporta tale semantica, il client deve occuparsi personalmente di ripristinare il file in uno stato consistente.

Quando, in un file system stateful, un file server smette di funzionare, le informazioni di stato memorizzate nei metadati vanno perse. Pertanto è necessario annullare eventuali operazioni di accesso al file e riportare quest'ultimo nello stato precedente. Per evitare tali problemi, come descritto nel paragrafo seguente, è possibile utilizzare un *file system stateless*.

#### 19.4.3 File server stateless

Un file system stateless non preserva alcuna informazione di stato relativa alle operazioni di accesso ai file, e quindi non viene instaurato alcun legame tra client e file server. Di conseguenza, il client deve mantenere localmente le informazioni sullo stato delle operazioni di accesso ai file, e deve inserire tali informazioni nelle chiamate di funzione effettuate per accedere al file system. Per esempio, un client che legge un file in modo sequenziale deve tener traccia dell'identificatore del prossimo byte da leggere in modo da poter effettuare chiamate a funzione del tipo:

```
read ("alpha", <id del byte>, <indirizzo area_io>)
```

A seguito di tale chiamata, il file server apre il file *alpha*, identifica la corrispondente tabella di mapping e la usa per convertire *<id del byte>* nella coppia (*id blocco su disco, offset*) (Paragrafo 13.9.2). Quindi, il file server legge su disco il blocco individuato dal mapping e fornisce il byte richiesto al client. In tal modo, la maggior parte delle azioni che solitamente vengono effettuate solo al momento dell'apertura del file, devono essere ripetute per ogni operazione eseguita sul file. Se un file server smette di funzionare, il client adotta un protocollo basato su timeout e ritrasmissione. Di conseguenza, al momento del riavvio in seguito al malfunzionamento, il file server elabora le eventuali richieste ritrasmesse e fornisce le corrispondenti risposte al client. In tal modo, il client non si accorge del malfunzionamento del server, ma percepisce solo un ritardo nel tempo di risposta.

L'uso di file server stateless, da un lato, garantisce tolleranza ai guasti, ma dall'altro, causa un sostanziale peggioramento delle prestazioni dovuto principalmente a due ragioni. In primo luogo, il file server deve aprire il file e fornire informazioni sul suo stato per ogni operazione richiesta dal client. In secondo luogo, quando un client esegue un'operazione di scrittura, per garantire un certo grado di affidabilità, i dati devono essere immediatamente scritti nella copia del file residente sul disco. Di conseguenza, un file system stateless non può utilizzare buffering, file caching (Paragrafo 19.5.2), o disk caching (Paragrafo 14.12) per velocizzare le operazioni di accesso. Nel Paragrafo 19.5.1, verrà presentato un tipo di file server ibrido che permette di evitare la ripetizione della *open* per accessi effettuati sullo stesso file.

Un file server stateless non risente di eventuali guasti sul client poiché non preserva informazioni di stato né sul client né sulle operazioni di accesso ai file da esso effettuate. Se si verifica un guasto su un client, al momento del riavvio quest'ultimo invia nuovamente le richieste pendenti al file server che dovrà semplicemente rielaborarle. Per lo stesso motivo, il server non può identificare e scartare eventuali richieste duplicate, e quindi potrebbe servire una stessa richiesta più di una volta. Una singola operazione di lettura o scrittura è idempotente, e quindi la sua riesecuzione non crea problemi. Tuttavia, richieste relative alle directory, come la creazione o la cancellazione di un file, non sono idempotenti. Di conseguenza, il client può ricevere notifiche ambigue o

ingannevoli nel caso di malfunzionamento di un file server stateless, che successivamente viene ripristinato mentre è in atto un'operazione di accesso a una directory. Anche sequenze di letture e scritture possono non essere idempotenti. Per esempio, una sequenza di operazioni costituita dalla lettura di un record in un file, ricerca della stringa xyz nel record, inserimento di una stringa S prima di xyz, e scrittura del record modificato nel file, non è idempotente. Nel caso in cui un client smetta di funzionare durante l'esecuzione di una sequenza non idempotente, è necessario ripristinare il file a uno stato precedente prima di rieseguire la stessa sequenza di operazioni.

## 19.5 Prestazioni di un FSD

L'utilizzo di un meccanismo di accesso efficiente ai file permette a un file system distribuito di garantire prestazioni elevate sia dal punto di vista del throughput che del tempo medio di risposta in relazione alle richieste dei client. Un FSD è in grado di fornire prestazioni elevate quando tutti gli accessi sono locali ai nodi client, ovvero quando client e file server risiedono nello stesso nodo. Al contrario, la latenza della rete può annullare completamente l'efficienza dei meccanismi di accesso ai file utilizzati, anche qualora solo una piccola frazione degli accessi dia origine a traffico di rete. Ciò motiva la necessità di identificare strategie per ridurre il traffico di rete causato dall'accesso ai file.

Un FSD è *scalabile* se le sue prestazioni non degradano all'aumentare della dimensione del sistema distribuito. La scalabilità è una caratteristica importante per evitare situazioni in cui un FSD con buone prestazioni in un determinato contesto, diventi un collo di bottiglia qualora il contesto si allarghi. La scalabilità si ottiene tramite tecniche speciali che garantiscono che il traffico di rete non cresca all'aumentare della dimensione del sistema distribuito.

La [Tabella 19.4](#) riassume le tecniche usate per ottenere FSD a elevate prestazioni. Tali tecniche sono trattate in dettaglio nei paragrafi seguenti.

| Tecnica                           | Descrizione                                                                                                                                                                                                                                                                                                                                                             |
|-----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| File server multithread           | Ogni richiesta effettuata dai client viene gestita da un thread separato. La manipolazione di un file è un'attività I/O bound, quindi parecchi thread possono lavorare in parallelo contribuendo così a innalzare il throughput.                                                                                                                                        |
| File server basato su hint        | Un <i>hint</i> (letteralmente, suggerimento) è un'informazione relativa a un'operazione su file in corso di svolgimento, che potrebbe essere mantenuta nel file server. Quando un hint è disponibile, il file server si comporta come un file server stateful per poter eseguire l'operazione in modo efficiente; altrimenti si comporta come un file server stateless. |
| File caching                      | Alcune parti di un file residente su un nodo remoto vengono copiate nella <i>file cache</i> del client. Il file caching riduce il traffico di rete durante l'accesso ai file sostituendo il trasferimento di dati sulla rete con un trasferimento di dati locale al nodo client.                                                                                        |
| Cluster di nodi semi-indipendenti | Un <i>cluster di nodi</i> è un sottoinsieme del sistema distribuito che contiene un numero di risorse hardware e software tali che le operazioni eseguite dai processi appartenenti al cluster richiedano solo raramente risorse posizionate al di fuori di esso.                                                                                                       |

**Tabella 19.4** Tecniche per il miglioramento delle prestazioni di un FSD.

### 19.5.1 Accesso a file efficiente

L'efficienza del meccanismo di accesso ai file dipende da come viene strutturato il funzionamento del file server. Nei paragrafi seguenti verranno presentati due modi di strutturare il file server che garantiscono un accesso efficiente ai file.

#### **File server multithread**

Un file server multithread è costituito da molti thread, ognuno dei quali è in grado di

servire le richieste inviate dai client. Visto che le operazioni di accesso ai file sono attività di tipo I/O bound, è possibile sovrapporre l'esecuzione di più thread. In tal modo le richieste dei client vengono servite più velocemente e il throughput ne risente positivamente. Il numero di thread utilizzati può variare in base al numero di richieste effettuate dai client in ogni istante di tempo, e con la disponibilità di risorse del sistema operativo, quali per esempio, il numero di blocchi di controllo utilizzabili per creare nuovi thread.

#### **File server basati su hint**

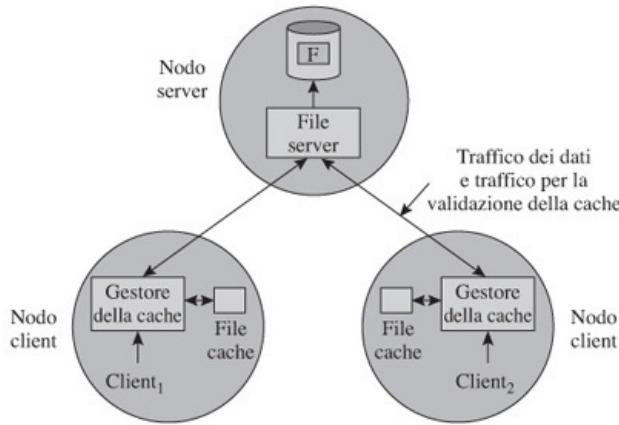
Un file server basato su hint ha una struttura ibrida, nel senso che possiede sia le caratteristiche di un file server stateful che quelle di un file server stateless. Per garantire un maggior livello di efficienza, agisce in modalità stateful ogniqualsiasi volta è possibile, mentre, in altri casi, si comporta come un file system stateless. Un *hint* (letteralmente, suggerimento) rappresenta un'informazione relativa a un'operazione di accesso al file in corso di svolgimento come, per esempio, l'identificatore del prossimo record da leggere quando si accede a un file in modo sequenziale (Paragrafo 13.8). Il file server mantiene un insieme di hint nella sua memoria volatile. Quando un client chiede di poter eseguire un'operazione su un file, il file server verifica se è presente un hint utile per velocizzare l'operazione richiesta, e in tal caso lo usa. Se l'hint non è disponibile, il file server agisce in modalità stateless - apre il file e usa l'identificatore del byte fornito dal client per accedere al byte richiesto. In entrambi i casi, dopo aver completato l'operazione, il file server memorizza parte dello stato relativo all'attività eseguita all'interno di un hint nella sua memoria volatile e lo restituisce al client. L'efficienza complessiva del file server dipende dal numero di operazioni che vengono eseguite con l'aiuto di hint.

Un file server basato su hint è tollerante ai guasti poiché la sua funzionalità non viene inficiata anche se, a causa di un guasto, tutti gli hint presenti nella sua memoria volatile venissero persi. In tal caso, infatti, gli utenti notano solo un degrado dei tempi di risposta finché il file server non ricostruisce un insieme di hint significativo.

### **19.5.2 File caching**

La tecnica nota come *file caching* velocizza le operazioni di un FSD riducendo il traffico di rete attraverso la memorizzazione delle informazioni relative al file remoto su un buffer locale del client detto *file cache*. La file cache e la copia del file sul disco del nodo server costituiscono una sorta di gerarchia di memoria (Paragrafo 2.2.3). Pertanto, il modo di operare della file cache e i benefici che ne derivano sono analoghi a quanto avviene per la cache della CPU. I blocchi dati di un file, detti *chunk*, vengono caricati dal file server nella file cache. Per beneficiare della località spaziale, ogni chunk è grande a sufficienza per soddisfare un certo numero di accessi a file effettuati da un client. Alcuni studi effettuati sulla distribuzione della dimensione dei file indicano che i file sono mediamente piccoli. Quindi è possibile copiare nella cache anche l'intero file. In tal caso, si parla di *whole-file caching*. Gli studi effettuati da Tanenbaum e da altri autori hanno osservato che il 79% dei file contenuti nei loro calcolatori erano più piccoli di 4 KB, mentre il 94% erano più piccoli di 16 KB. Nell'Andrew file system, la dimensione di un chunk, che varia a seconda del client, è mediamente pari a 8 KB ed è solitamente sufficiente per contenere un intero file. Sempre nell'Andrew file system, il tasso di successo della file cache supera il 98%. Un FSD, oltre alla file cache, può usare anche una cache separata per memorizzare le informazioni relative agli attributi dei file. Si parla in tal caso di *attribute cache*.

La [Figura 19.3](#) mostra uno schema del meccanismo basato su file caching. Il *gestore della cache* funge da intermediario tra client e file server. Carica i blocchi del file nella file cache; preleva i dati dalla cache e li fornisce ai client; mantiene la file cache per mezzo di un opportuno algoritmo di rimpiazzamento dei blocchi e aggiorna la copia del file posseduta dal file server con i blocchi modificati dai client.



**Figura 19.3** Schema del file caching.

Gli aspetti principali relativi alla progettazione di una file cache sono:

- posizione della file cache;
- politica di aggiornamento dei file;
- politica di validazione della cache;
- dimensione dei chunk.

La file cache può essere mantenuta nella memoria di un nodo client o su uno dei suoi dischi. Mantenere la file cache in memoria garantisce un accesso più veloce ai dati; tuttavia è sicuramente meno affidabile, poiché un guasto nel nodo client può portare alla perdita dei dati contenuti in essa, comprese eventuali modifiche non ancora riportate nella copia del file nel file server. Mantenere la cache nel disco, invece, rallenta le operazioni di accesso ai file, ma garantisce una maggior affidabilità, in quanto la file cache e i dati presenti sul server non vengono persi in caso di malfunzionamento del nodo client. Le tecniche basate sulla ridondanza, come il mirroring del disco, possono essere usate per migliorare ulteriormente l'affidabilità di una file cache mantenuta su disco.

Quando un client esegue un'operazione di scrittura su disco, i dati modificati devono essere scritti anche nella copia del file presente sul server. È quindi necessario decidere se tale copia deve essere aggiornata immediatamente dopo la modifica effettuata dal client oppure in un secondo momento. La decisione ha impatto sul compromesso tra il ritardo percepito dal client e il livello di affidabilità fornito dal DFS. Dal punto di vista della semplicità è preferibile usare la politica *write-through*, in base alla quale la file cache del client e la copia del server vengono aggiornate nello stesso istante. Questo approccio può essere implementato usando le transazioni e quindi garantisce un ottimo livello di affidabilità. D'altra parte, esso induce un ritardo nel client a ogni operazione di scrittura. Per evitare tali ritardi, è possibile usare la politica *delayed write*, in base alla quale l'aggiornamento della copia del server può essere effettuato in un secondo momento. Tuttavia, questa politica richiede l'utilizzo di opportuni meccanismi che garantiscono di non perdere i dati qualora il nodo client subisca un guasto prima di aver concluso l'aggiornamento del server. La scrittura sul server può essere effettuata in diversi istanti - quando il chunk modificato deve essere rimosso dalla file cache a causa dell'algoritmo di rimpiazzamento, oppure quando il client chiude il file.

Se un file viene modificato contemporaneamente da molti client, nello stesso istante di tempo i suoi dati saranno contenuti in varie file cache. Se un client esegue un'operazione di scrittura, le copie nelle cache degli altri client diventano *non valide*. La funzione di *validazione della cache* identifica e gestisce i dati non validi in base alla semantica della condivisione implementata nell'FSD. Per esempio, nella semantica Unix, gli aggiornamenti di un file effettuati da un client devono essere immediatamente visibili agli altri client, quindi la corrispondente funzione di validazione della cache deve aggiornare i dati non validi oppure impedire ai client di poterli usare.

La dimensione di un chunk nella file cache dovrebbe essere sufficientemente grande da far sì che la località spaziale dei dati garantisca un elevato tasso di successo della cache. Tuttavia, l'uso di chunk grandi implica una maggior probabilità che i dati

contenuti in una cache vengano invalidati a causa di modifiche effettuate sullo stesso chunk da altri client, causando maggiori ritardi e un maggiore overhead per la validazione della cache rispetto a quanto avviene usando chunk piccoli. È pertanto necessario individuare un compromesso sulla dimensione dei chunk in un FSD. Tuttavia, imporre una dimensione fissa per i chunk potrebbe non adattarsi alle esigenze di tutti i client del sistema. Quindi, alcuni FSD, come per esempio l'Andrew file system, permettono ai vari client di usare dimensioni diverse per i chunk.

### **Validazione della cache**

Un semplice metodo per identificare i dati non validi consiste nell'associare un timestamp a tutti i chunk di un file e a tutti i corrispondenti chunk nella cache. Il timestamp di un chunk indica l'ultima volta in cui i dati contenuti in esso sono stati modificati. Quando un chunk di un file viene copiato nella cache, viene memorizzato in quest'ultima anche il suo timestamp. È quindi possibile verificare, in ogni istante, la validità di un chunk contenuto nella cache confrontando il suo timestamp con quello del corrispondente chunk nel file originale. In tal modo un'operazione di scrittura effettuata da un client sul chunk  $x$  di un file è in grado di invalidare tutte le copie di  $x$  nelle cache degli altri client. I dati di un chunk invalidato verranno quindi ricaricati al successivo accesso.

Esistono principalmente due tecniche per validare la cache: *validazione iniziata dal client* e *validazione iniziata dal server*. La validazione iniziata dal client viene effettuata dal gestore della cache nel nodo del client. Ad ogni accesso al file effettuato dal client, il gestore controlla se i dati richiesti sono già presenti nella cache e, in tal caso, verifica la loro validità. Se la verifica ha successo, il gestore fornisce i dati al client prelevandoli dalla cache, altrimenti, prima di restituire i dati al client, viene effettuato l'aggiornamento della cache. Questo tipo di approccio crea traffico di rete per la validazione della cache a ogni accesso a file. Il traffico può però essere ridotto effettuando la validazione della cache periodicamente e non a ogni accesso, a patto che tale validazione sia consistente con la semantica della condivisione dei file implementata nell'FSD. Sun NFS utilizza tale approccio (Paragrafo 19.6.1).

Nella validazione iniziata dal server, quest'ultimo tiene traccia dei dati memorizzati nelle file cache dei vari client. Quando un client aggiorna i dati nel chunk  $x$  di un file, il server informa i gestori di tutti client che possiedono una copia di  $x$  nella loro cache di invalidare i dati locali relativi a  $x$ . Di conseguenza, i manager dei client possono decidere di cancellare o aggiornare immediatamente la copia di  $x$  contenuta nella cache, oppure di aggiornare quest'ultima solo nel momento in cui verrà effettuato un nuovo accesso a  $x$ .

Poiché la validazione della cache è un'operazione costosa, alcune semantiche di condivisione dei file, come per esempio la semantica di sessione, non richiedono che gli aggiornamenti effettuati da un client siano subito visibili ai client degli altri nodi. In tal modo non è necessario eseguire la validazione della cache. In alternativa, un'ulteriore strategia per evitare il sovraccarico richiesto dalla validazione della cache consiste nel disabilitare il file caching quando un client apre un file in scrittura. Così facendo, tutti gli accessi a tale file vengono direttamente implementati nel nodo server.

### **19.5.3 Scalabilità**

In un FSD, la scalabilità si ottiene cercando di localizzare all'interno di piccole porzioni del sistema distribuito, chiamate *cluster di nodi* o semplicemente *cluster* (Paragrafo 16.2), la maggior parte del traffico dati generato dalle operazioni di accesso ai file. Le ragioni che rendono efficace un simile approccio sono principalmente due. In primo luogo, i cluster costituiscono delle sottoreti che garantiscono un elevato tasso di trasferimento dei dati, come per esempio le LAN ad alta velocità. Pertanto, sia il tempo di risposta che il throughput traggono beneficio confinando il traffico dati su un singolo cluster. In secondo luogo, un aumento del numero dei cluster non porta a un degrado delle prestazioni poiché il traffico di rete aggiunto da un cluster, a livello di sistema distribuito, non è particolarmente elevato. Quando un client di un FSD che possiede sia la trasparenza che l'indipendenza dalla locazione accede a un file remoto, quest'ultimo può essere semplicemente spostato nel cluster dove è posizionato il client. Se, al contrario l'FSD non garantisce l'indipendenza dalla locazione, è possibile ottenere un effetto analogo per i file in sola lettura, tramite l'uso della replicazione o del file caching nel nodo client. D'altra parte, nel caso sia invece necessario effettuare operazioni di scrittura su file, l'uso della semantica di sessione elimina il traffico dati dovuto alla

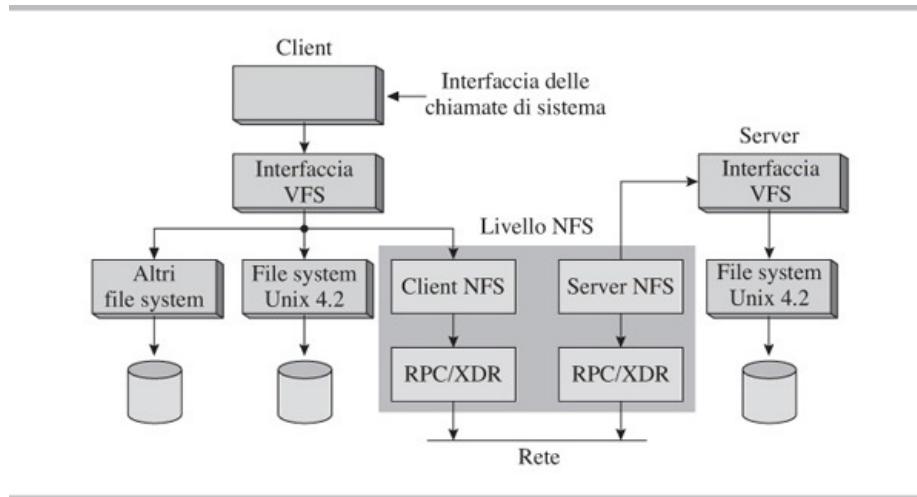
validazione della cache, e quindi per ridurre il traffico di rete è sufficiente salvare una copia del file nel nodo client.

## 19.6 Casi di studio

### 19.6.1 Il network file system di Sun

Il Network File System (NFS) di Sun permette la condivisione di file system tra nodi che utilizzano il sistema operativo SunOS, ovvero la versione di Unix della Sun.

La [Figura 19.4](#) mostra il diagramma schematico di NFS. L'architettura si compone di due livelli: il *virtual file system* (VFS) e il livello NFS. Il livello VFS implementa il *protocollo di mount* e crea un identificatore unico a livello di sistema per ogni file, detto *vnode*. Se il file su cui si desidera effettuare un'operazione si trova in un file system locale, il livello VFS si preoccupa di invocare tale file system. In caso contrario, il VFS invoca il livello NFS, che si occupa di interagire tramite il *protocollo NFS* con il nodo remoto contenente il file. Questa architettura permette a un nodo di funzionare sia da client che da server nello stesso istante.



**Figura 19.4** Architettura del Network File System (NFS) di Sun.

#### Protocollo di mount

Ogni nodo del sistema possiede una *export list* che contiene coppie del tipo *(<directory>, <lista\_dei\_nodi>)*. Ogni coppia indica che la *<directory>*, che risiede in un file system locale, può essere montata in remoto soltanto nei nodi contenuti nella *<lista\_dei\_nodi>*. Quando il superuser di un nodo richiede di montare in locale una directory remota, NFS verifica la validità della richiesta, monta la directory e restituisce al chiamante un *file handle*, ovvero una struttura dati che contiene l'identificatore del file system su cui risiede la directory remota, e il corrispondente inode di quest'ultima. Di conseguenza, la gerarchia di directory vista dagli utenti del nodo rifletterà l'effetto dell'operazione di mount.

NFS permette di effettuare operazioni di mount in cascata, ovvero è possibile montare un file system all'interno di un altro file system, che a sua volta è stato precedentemente montato in un altro file system, e così via. Tuttavia, NFS impedisce la transitività del meccanismo di mount. Per esempio, si consideri la seguente situazione:

1. il superuser del nodo  $N_1$  monta il file system  $C$  del nodo  $N_3$  nel punto di mount  $y$  del file system locale  $B$ ;
2. il superuser del nodo  $N_2$  monta il file system  $B$  del nodo  $N_1$  nel punto di mount  $x$  del file system locale  $A$ .

NFS non permette agli utenti del nodo  $N_2$  di accedere al file system  $C$  montato all'interno del file system  $B$ . In tal modo, la gerarchia di directory vista da ogni macchina deriva solamente dalle operazioni di mount effettuate dal proprio superuser, e il file server può operare in modalità stateless. Se non fosse imposta tale restrizione, ogni file

server dovrebbe essere a conoscenza di tutte le operazioni di mount effettuate da qualunque client sul suo file system, il che richiederebbe un file server stateful.

### **Protocollo NFS**

Il protocollo NFS utilizza il paradigma del servizio remoto (ovvero, l'accesso remoto ai file - Paragrafo 19.1.1) tramite un modello client-server che fa uso di chiamate a procedure remote (RPC). Il file server è stateless, quindi i parametri delle RPC identificano il file, la directory che lo contiene, e l'identificatore del record da leggere o i dati da scrivere. Le chiamate a funzione messe a disposizione da NFS permettono di trovare un file all'interno di una directory, leggere gli elementi di una directory, manipolare link e directory, accedere agli attributi di un file (ovvero le informazioni contenute nel suo inode), nonché effettuare operazioni di lettura e scrittura.

Dal momento che il file server è stateless, esso esegue implicitamente le operazioni di apertura e chiusura per ogni accesso al file e non usa il buffer cache di Unix (si veda il Paragrafo 14.13.1 per la descrizione del buffer cache di Unix). Inoltre, non fornisce il locking di file e record; gli utenti devono pertanto farsi carico dell'implementazione di meccanismi ad hoc per il controllo della concorrenza.

### **Risoluzione del pathname**

Supponiamo che l'utente U1 posizionato nel nodo  $N_1$  desideri utilizzare il pathname  $x/y/z/w$  dove  $y$  è la cartella radice di un file system montato su  $N_1$ . All'inizio,  $N_1$  crea il vnode  $vnode_x$  per  $x$ . Quindi, NFS, usando la tabella di mount di  $N_1$  per trovare il prossimo componente del pathname, si accorge che  $y$  è una directory montata. Esso crea, pertanto,  $vnode_y$  in base alle informazioni fornite dalla tabella di mount. Se  $vnode_y$  si riferisce a un file nel nodo  $N_2$ , NFS crea una copia della directory  $y$  nel nodo  $N_1$ . Quindi, cercando  $z$  nella copia di  $y$ , NFS utilizza nuovamente la tabella di mount di  $N_1$ . In tal modo, NFS è in grado di risolvere opportunamente  $z$  anche se si tratta di un file system montato dal superuser di  $N_1$  in qualche punto del file system remoto  $y$ . Inoltre, il file server del nodo  $N_2$ , che contiene  $y$ , non necessita di essere a conoscenza di tale eventuale mounting. Se al posto di usare questa procedura, il pathname  $y/z/w$  venisse trasmesso al file server del nodo  $N_2$ , esso dovrebbe conoscere tutti i mount effettuati da tutti i client del suo file system. Ciò richiederebbe che il file server fosse stateful.

Per velocizzare la risoluzione dei pathname, ogni client utilizza una cache speciale per memorizzare i nomi delle directory remote note e i relativi vnode. Ogni volta un client risolve il prefisso di un nuovo pathname, aggiunge un elemento corrispondente nella cache. Al contrario, quando la risoluzione di un pathname presente nella cache fallisce a causa di una incongruenza tra gli attributi restituiti dal file server e quelli contenuti nel vnode della cache, il corrispondente elemento della cache viene rimosso.

### **Operazioni su file e semantica della condivisione**

NFS usa due cache per velocizzare le operazioni su file. La *cache degli attributi dei file* memorizza informazioni relative agli inode. Il suo utilizzo è motivato dal fatto che una grande percentuale delle richieste fatte ai file server è relativa proprio agli attributi dei file. Gli attributi memorizzati nella cache vengono scartati dopo 3 secondi per i file e 30 secondi per le directory. I blocchi dei file sono invece contenuti nella file cache, detta appunto *cache dei blocchi dei file*. Il file server utilizza blocchi dati grandi (8 KB), e adotta le tecniche read-ahead e delayed-write (ovvero, tecniche di bufferizzazione, Paragrafo 14.8) per migliorare le prestazioni del file system. La validazione della cache viene effettuata tramite timestamp associati a ogni file e a ogni blocco della cache. Il contenuto di un blocco nella cache è ritenuto valido per un certo periodo di tempo. Per accessi successivi a tale periodo, viene usato il blocco della cache solo se il suo timestamp è maggiore di quello del file sul server. I blocchi modificati dai client vengono inviati al file server affinché aggiorni la propria copia in istanti di tempo non precisati, oppure alla chiusura del file. Questa politica viene usata anche se più client accedono concorrentemente allo stesso blocco dati. Pertanto, in base alla politica precedente e allo schema di validazione della cache adottata, la semantica della condivisione dei file di NFS non è classificabile né come semantica Unix né come semantica di sessione.

## **19.6.2 I file system Andrew e Coda**

Andrew, l'ambiente di calcolo distribuito sviluppato alla Carnegie Mellon University, è

stato progettato per supportare sistemi distribuiti di grandi dimensioni contenenti fino a 5000 workstation. Ogni workstation è equipaggiata con un disco fisso che viene utilizzato per organizzare il name space locale. Quest'ultimo contiene i programmi di sistema per il boot e il funzionamento della macchina, e file temporanei memorizzati per ragioni di efficienza. I client condividono inoltre un name space, trasparente alla posizione, implementato con server dedicati chiamati *Vice*.

La scalabilità delle prestazioni è ottenuta grazie all'uso di cluster che delimitano il più possibile le operazioni di accesso a file, in modo da non generare traffico di rete sulla dorsale principale del sistema. Il traffico interno al cluster viene inoltre ridotto memorizzando interamente il file sul disco locale della workstation su cui risiede il client che ne esegue l'accesso. Queste due tecniche garantiscono che il traffico di rete globale non aumenti al crescere della dimensione del sistema.

### **Name space condiviso**

In Andrew, i file di un singolo utente sono contenuti in un *volume*, e ogni disco può contenere molti volumi. Andrew considera un volume come una partizione del disco in Unix, sebbene il volume possa essere generalmente più piccolo di una partizione. Data la dimensione ridotta, l'operazione di mounting di un volume consente di ottenere una granularità più fine rispetto a quanto avviene in Unix. L'identificatore dei file usato da *Vice* contiene il numero del volume su cui risiede il file, e un indice per il vettore di inode contenuti nel volume.

Le informazioni relative ai volumi contenuti nel sistema sono memorizzate nel *volume location database* (VLDB), che viene replicato su ogni server. I volumi vengono spostati da un disco all'altro per bilanciare l'utilizzo dei dischi del sistema. In seguito a una migrazione, il server che originariamente conteneva il volume, conserva alcune informazioni necessarie a inoltrare eventuali richieste pendenti verso la nuova locazione del volume, finché tutti i server non hanno aggiornato il loro VLDB. In tal modo, si semplificano le operazioni di migrazione eliminando la necessità di aggiornare tutti i VLDB nello stesso istante. La migrazione di un volume viene effettuata cercando di disturbare il meno possibile le attività di accesso ai file in corso di svolgimento. Per garantire ciò, viene effettuata una copia del volume da trasferire sul server destinazione, mentre il server originario continua a gestire le richieste relative al volume. Al termine della copia, il volume originario viene messo off-line, eventuali aggiornamenti effettuati nel server originario durante l'operazione di copia vengono riprodotti nel nuovo server, e il volume viene rimesso in funzione sul nuovo server.

### **Operazioni su file e semantica della condivisione**

Quando un client apre un file, Andrew memorizza una copia di quest'ultimo sul disco locale della workstation del client, usando chunk da 64 KB. Tuttavia, la dimensione dei chunk viene adattata in base al client per conformarsi alla tipologia di accesso al file eseguita da quest'ultimo. Come riportato nel Paragrafo 19.5.2, gli studi condotti nella metà degli anni '90 hanno mostrato come sia sufficiente l'uso di chunk da 8 KB, e che il tasso di successo della cache in tali situazioni supera generalmente il 98%. Le chiamate alle funzioni di apertura e chiusura di un file sono dirette a un processo utente detto *Venus*. Quando un client apre un file, Venus ne memorizza una copia in locale. Successivamente, alla chiusura del file, Venus aggiorna la copia del server. Le operazioni di lettura e scrittura su file vengono quindi eseguite sulla copia locale senza coinvolgere Venus. Di conseguenza, le modifiche apportate a un file non sono immediatamente riportate sul server e quindi non sono visibili agli altri client che accedono al file. In tal modo, Andrew implementa una semantica della condivisione dei file simile alla semantica di sessione, tuttavia, non mantiene copie multiple di uno stesso file.

La copia del file memorizzata in locale da Venus rimane valida finché quest'ultimo non viene avvisato del contrario. In tal modo, una copia del file può rimanere valida nella cache tra un'operazione di chiusura e una successiva operazione di apertura nella stessa workstation. La validazione della cache viene avviata dal server usando un meccanismo chiamato *callback*. Quando si esegue una copia locale di un file *F* in un nodo *N*<sub>1</sub> a seguito di un'operazione di apertura, il server annota l'azione su un'opportuna tabella. Per tutto il tempo in cui tale annotazione rimane nella tabella, si dice che il nodo *N*<sub>1</sub> possiede una callback su *F*. Quando la copia di *F* sul server viene aggiornata, in seguito alla chiusura del file da parte di qualche cliente, il server rimuove l'annotazione relativa al nodo *N*<sub>1</sub> e notifica al processo Venus, in esecuzione su quest'ultimo, che la sua callback su *F* è stata eliminata. Se, in seguito, qualche client su *N*<sub>1</sub> cerca di aprire *F*, sapendo che *N*<sub>1</sub> non ha

più una callback valida su F, Venus eseguirà una nuova operazione di caching di F. Venus mantiene due cache - una per i dati contenuti nei file e una per le informazioni di stato dei file stessi. Entrambe le cache sono gestite usando la strategia LRU.

La risoluzione dei pathname viene effettuata componente per componente. Venus mantiene una cache per memorizzare informazioni sui volumi cui è stato fatto accesso recentemente. Tuttavia, dal momento che i volumi possono essere spostati nel sistema, Venus considera queste informazioni come un semplice suggerimento e le scarta quando si accorge che sono sbagliate. Durante la risoluzione di un pathname, Venus copia nella cache ogni directory coinvolta in esso. Ciò permette di velocizzare future operazioni di risoluzione. Inoltre, per evitare che i file server diventino colli di bottiglia per le prestazioni, si utilizzano più thread, ciascuno dei quali è in grado di servire le richieste dei client. La comunicazione client-server è gestita usando il meccanismo RPC.

### **Caratteristiche di Coda**

Nel file system Coda, successore della versione 2 del file system Andrew, sono state aggiunte due caratteristiche complementari, ovvero, *replicazione* e *disconnessione*, per garantire una più elevata disponibilità di servizio. Coda supporta, pertanto, la replicazione dei volumi. Il gruppo di server che possiede una copia di uno stesso volume è detto *volume storage group* (VSG). Coda controlla l'uso dei file replicati attraverso la politica *read one, write all*, secondo la quale per leggere un file è sufficiente che sia disponibile una delle sue copie, ma in caso di scrittura, tutte le copie devono essere aggiornate nello stesso istante. A tale scopo viene utilizzata una strategia basata su RPC multicasting chiamata multiRPC.

Quando un nodo non riesce ad accedere a nessuno dei server contenuti nel VSG di interesse, entra nella modalità *disconnesso*. Anche il file system Andrew supporta questa modalità. Quest'ultimo, infatti, supporta il caching di interi file sui nodi client, e quindi un client che non riesce a connettersi al server può comunque operare sulla copia locale del file. Le eventuali modifiche eseguite sul file si riflettono sulla sua copia originale quando il client riesce a connettersi nuovamente al server, momento in cui vengono anche risolti eventuali conflitti dovuti alla presenza di altre versioni dello stesso file create nel frattempo da altri client. Sebbene nel caso di applicazioni specifiche, questa operazione possa essere automatizzata, generalmente, è richiesto l'intervento dell'utente.

La memorizzazione di un unico file nella cache non è, tuttavia, sufficiente per operare in modo efficiente nella modalità disconnesso. Pertanto, Coda permette di effettuare il caching di un maggior numero di file. Ogni utente può fornire a Coda un database contenente i pathname di un certo numero di file ritenuti *importanti*. Quindi, Coda adotta una gestione della cache basata su priorità, grazie alla quale mantiene nella cache di un client i file utilizzati più di recente e quelli contenuti nel database fornito dall'utente. L'insieme dei file memorizzati nella cache viene aggiornato periodicamente ricalcolando le loro priorità. In tal modo, la cache può contenere un insieme sufficientemente grande di file da poter operare efficacemente anche nella modalità disconnesso.

### **19.6.3 GPFS**

Il *general parallel file system* (GPFS) è un file system ad alte prestazioni basato sulla condivisione dei dischi che opera su grandi cluster Linux. GPFS utilizza il data striping (Paragrafo 14.3.5) su tutti i dischi disponibili nel cluster. Pertanto, i dati dei file vengono scritti su parecchi dischi, che possono essere letti o scritti in parallelo. Per minimizzare il seek time delle operazioni di lettura/scrittura di un file si utilizzano blocchi dati di grandi dimensioni, ovvero strip. Tuttavia, l'uso di blocchi troppo grandi riduce il tasso di trasferimento dei dati per file di piccole dimensioni che occupano solo pochi strip.

Pertanto, per file piccoli, vengono utilizzati sottoblocchi, la cui dimensione può essere  $\frac{1}{32}$  di quella di un blocco tradizionale.

Per mantenere la consistenza dei dati quando un file viene utilizzato contemporaneamente da più processi residenti su nodi diversi del cluster, si adotta la tecnica del locking. In tale contesto, se da un lato è necessario usare il locking a grana fine per garantire un elevato parallelismo nelle operazioni di accesso ai file, dall'altro, bisogna usare tecniche meno raffinate per mantenere basso l'overhead dovuto al locking. Pertanto, GPFS utilizza un approccio combinato che opera nel seguente modo. Il primo processo P1 che effettua un'operazione di scrittura su un file F riceve un lock che copre tutti i byte di F. Se nessun altro processo desidera accedere allo stesso file, P accede a F

senza dover usare il lock. Al contrario, se un altro processo P2 desidera accedere a F in scrittura, riceve un lock che copre solo i byte di F da modificare, mentre l'insieme di byte coperti dal lock di P1 viene ridotto per escludere i byte modificati da P2. In tal modo, la granularità del lock viene mantenuta il più possibile grande, ma quando serve, viene ridotta, tenendo comunque presente che l'intervallo di byte coperto da un lock non può essere più piccolo della dimensione di un blocco dati su disco. Ogniqualvolta l'intervallo di byte di un lock viene ridotto, gli aggiornamenti effettuati su byte che non sono coperti dal nuovo intervallo vengono riportati sulla copia originale del file. In tal modo, un eventuale processo che acquisisce un lock per tali byte è in grado di vedere il loro ultimo valore.

La strategia di lock implementata nel GPFS si basa su un gestore di lock centrale e alcuni gestori distribuiti. Per ridurre la latenza e l'overhead richiesto per la gestione dei lock, GPFS utilizza il concetto di *lock token*. Quando un processo accede per la prima volta a un file, il gestore centrale fornisce un lock token al nodo N su cui il processo è in esecuzione. Il possesso di tale token autorizza il nodo a concedere eventuali ulteriori lock per lo stesso file ad altri processi locali. In questo modo si evita la generazione di traffico tra N e il gestore centrale per l'acquisizione di ulteriori lock per lo stesso file. Quando un processo di un altro nodo M desidera accedere allo stesso file, il gestore centrale preleva il lock token dal nodo N e lo assegna a M. I byte coperti dai lock forniti da un nodo possono essere memorizzati nella cache di quest'ultimo. Ciò non genera traffico per garantire la coerenza della cache quando i byte in essa memorizzati vengono letti o aggiornati. Infatti, nessun processo di altri nodi può accedere agli stessi byte a causa del lock.

I metadati di un file, come per esempio i blocchi indice della FMT, possono essere soggetti a corse critiche, se molti nodi tentano di aggiornarli contemporaneamente. Per esempio, quando due nodi aggiungono un puntatore ciascuno allo stesso blocco indice della FMT, l'aggiornamento effettuato da un client sul blocco potrebbe andar perso nel caso in cui un altro client eseguisse il proprio aggiornamento. Per prevenire inconsistenze dovute a corse critiche, ad ogni file viene assegnato un metanodo tra tutti i nodi del sistema, in modo tale che solo il metanodo possa eseguire accessi e aggiornamenti ai metadati del corrispondente file. Gli altri nodi che eseguono aggiornamenti sul file inviano i loro metadati al metanodo che si occupa di eseguirne il commit su disco.

Nel caso un cui le operazioni di accesso ai file eseguite dai vari nodi del sistema, richiedano ulteriore spazio su disco, la lista dello spazio libero può diventare un collo di bottiglia per le prestazioni. Il gestore centrale, responsabile dell'allocazione dello spazio, evita i colli di bottiglia partizionando la mappa dello spazio libero e fornendone una parte a ogni nodo. In tal modo, ogni nodo provvede ad allocare lo spazio sul disco usando solo la propria partizione. Quando tutto lo spazio di una partizione è stato assegnato, il nodo ne richiede una nuova al gestore centrale.

Ogni nodo è responsabile della scrittura di un proprio *journal*, ovvero un file di log, per eventuali operazioni di ripristino. Il journal è posizionato nel file system cui appartiene il file modificato. Tuttavia, se un nodo smette di funzionare, il suo journal viene aggiornato da altri nodi. In tal modo, la consistenza dei dati aggiornati è implicitamente garantita, in quanto un nodo che smette di funzionare avrà comunque precedentemente acquisito un lock per i dati cui i suoi processi stavano accedendo prima del guasto. Questi lock vengono rilasciati solo dopo che qualche altro nodo abbia correttamente elaborato il journal del nodo che ha smesso di funzionare.

Eventuali guasti sui canali di comunicazione possono partizionare il sistema. Tuttavia, le operazioni di accesso ai file in atto nei singoli nodi non subiscono conseguenze in seguito al partizionamento del sistema, qualora tali nodi siano in grado di accedere ad alcuni dischi del sistema. Tuttavia, questo modo di usare il file system può portare a inconsistenze nei metadati, per prevenire le quali, GPFS impone che solo i nodi di una singola partizione possano continuare a usare i file, mentre gli altri sono obbligati a cessare le loro operazioni. In particolare, solo i nodi appartenenti alla partizione maggioritaria, cioè quella che contiene il maggior numero di nodi, possono accedere ai file in ogni istante. GPFS utilizza un servizio che invia messaggi speciali, detti *heartbeat* (ovvero battito cardiaco) per rilevare l'eventuale malfunzionamento dei nodi. Tale servizio invia notifiche ai vari nodi per segnalare la loro eventuale esclusione dalla partizione maggioritaria, a causa di un guasto nel sistema, o la loro riammissione. Tuttavia, tali notifiche possono arrivare in ritardo o andare perse proprio a causa di un guasto nei canali di comunicazione, e quindi GPFS utilizza anche le funzionalità del sottosistema di I/O per evitare che i nodi non appartenenti alla partizione maggioritaria

accedano ai vari dischi del sistema. Inoltre, GPFS utilizza la replicazione per proteggersi contro guasti che possono verificarsi sui dischi.

#### 19.6.4 Windows

Il file system del sistema Windows Server 2003 mette a disposizione due funzionalità per la replicazione e la distribuzione dei dati.

- *Compressione differenziale remota* (RDC): un protocollo per la replicazione dei file che riduce il traffico tra i server dovuto alla replicazione dei dati e alla gestione della loro consistenza.
- *Namespace DFS*: un meccanismo per creare alberi di directory virtuali posizionati su server differenti, in modo che vi si possa accedere dai client di qualunque nodo.

La replicazione viene implementata usando la nozione di *gruppo di replicazione*, ovvero un gruppo di server che replica un gruppo di directory. Se un client desidera accedere a più directory contenute nello stesso gruppo, gli viene permesso di eseguire l'accesso in modalità off-line rispetto al server che le contiene. In tale contesto, il protocollo RDC viene usato per sincronizzare periodicamente le copie di una directory replicata all'interno del proprio gruppo di replicazione. Questo protocollo trasmette a tutti i membri di un gruppo di replicazione solo i cambiamenti effettuati sui file, o solo le differenze esistenti tra copie diverse dello stesso file. Inoltre, quando viene creato un nuovo file, RDC identifica i file già esistenti che sono simili a quello appena creato e trasmette ai membri del gruppo di replicazione solo le differenze rispetto a questi ultimi. In tal modo viene ridotta la banda utilizzata per la replicazione.

Il namespace DFS viene creato dall'amministratore del sistema. Per ogni directory contenuta nel namespace, l'amministratore specifica una lista di server che ne contengono una copia. Quando un client accede a una directory condivisa che appartiene al namespace, viene contattato il namespace server per risolvere il pathname all'interno dell'albero virtuale. Quest'ultimo invia al client un riferimento contenente la lista dei server che possiedono una copia della directory desiderata. Il client contatta quindi il primo server della lista per accedere alla cartella. Se tale server non risponde e nel client è stata abilitata l'opzione *fallback*, quest'ultimo notifica il fallimento e contatta il successivo server nella lista.

### Riepilogo

Un file system distribuito (FSD) memorizza i file degli utenti in vari nodi del sistema, e quindi un processo può risiedere in un nodo diverso rispetto a quello in cui si trova il file cui desidera accedere. Una simile situazione richiede che un file system distribuito utilizzi tecniche speciali tali che l'utente (1) non abbia bisogno di sapere dove si trova il file, (2) possa accedere al file anche in caso di guasti sui collegamenti e sui nodi del sistema, e (3) possa elaborare i file in modo efficiente. In questo capitolo è stato mostrato come i file system distribuiti riescano a soddisfare i precedenti requisiti.

La nozione di *trasparenza* si riferisce al legame esistente tra il pathname di un file e la sua locazione. Essa determina se un utente deve conoscere la locazione del file per potervi accedere e se il sistema può cambiare la posizione di un file senza ripercussioni sul nome di quest'ultimo. Un alto livello di trasparenza semplifica le operazioni degli utenti e permette a un FSD di ridurre il traffico di rete spostando i file verso i nodi in cui questi ultimi vengono usati più frequentemente.

Un altro aspetto importante per agevolare l'uso di un file system è rappresentato dalla *semantica della condivisione dei file*. Quest'ultima specifica se gli aggiornamenti effettuati da un processo su un file sono visibili ad altri processi che accedono allo stesso file nello stesso istante di tempo. Vi sono tre tipi principali di semantica. Nella *semantica Unix*, gli aggiornamenti eseguiti su un file da parte di un processo sono visibili immediatamente anche a tutti gli altri processi che usano lo stesso file. Al contrario, nella *semantica di sessione*, gli aggiornamenti sono visibili solo ad alcuni processi dello stesso nodo. Infine, nella *semantica delle transazioni*, ogni singola attività di accesso ai file viene trattata come un'unica *transazione atomica*, in modo che ognuno degli aggiornamenti eseguiti, oppure nessuno di essi, si rifletta sul file e sia visibile agli altri processi al termine dell'operazione di aggiornamento.

Per garantire un'elevata disponibilità di servizio da parte di un file system è necessario che le operazioni di accesso ai file effettuate da un processo non siano influenzate da guasti transienti che possono verificarsi nel *nodo server*, ovvero nel nodo che detiene il file. Per garantire ciò, alcune tipologie di FSD si basano su *server stateless* che non mantengono nessuna informazione di stato relativa alle operazioni di accesso ai file in corso. Di conseguenza, un malfunzionamento del nodo server non influenza un'eventuale operazione di accesso al file in corso - essa, infatti, può essere rieseguita non appena la funzionalità del server viene ripristinata. Tuttavia, per qualunque operazione su file, i server stateless necessitano di accedere alla directory che contiene il file per identificare la sua locazione. Per incrementare le prestazioni dei file system che utilizzano server stateless, si utilizzano gli *hint*, ovvero parti delle informazioni di stato dell'FSD. Quando, durante l'accesso a un file è disponibile un *hint*, l'FSD lo usa per velocizzare la relativa operazione. In caso contrario, l'FSD si comporta come server stateless.

Le prestazioni di un FSD sono influenzate dalla latenza della rete ogniqualvolta un processo e il file a cui esso desidera accedere si trovano in nodi diversi. Per migliorare le prestazioni, gli FSD utilizzano tecniche di *file caching* grazie alle quali una copia dei dati del file viene preservata nel nodo in cui risiede il processo che desidera accedervi. In tal modo, gli accessi al file sono implementati in locale e non necessitano dell'uso della rete. Tuttavia, se alcuni processi, posizionati su nodi differenti del sistema, aggiornano contemporaneamente un file, le relative cache si troveranno ad avere versioni diverse dello stesso file. Pertanto, ciascun processo potrebbe non essere in grado di vedere l'ultimo valore dei dati aggiornati da un altro processo. Tale problema può essere superato usando tecniche per garantire la *coerenza della cache*, che evitano il pericolo di accedere a file vecchi. Tuttavia, l'uso di tali tecniche genera traffico di rete per aggiornare opportunamente le copie dei file memorizzate nella varie cache, riducendo così, in parte, i benefici derivanti dallo stesso file caching. In tale contesto, la semantica di sessione elimina il traffico necessario per mantenere la coerenza della cache, in quanto gli aggiornamenti effettuati da un processo non sono visibili al di fuori del nodo in cui quest'ultimo si trova.

## Domande

- 19.1. Indicare se le seguenti affermazioni sono vere o false.
  - a. L'indipendenza dalla locazione in un file system distribuito porta benefici agli utenti.
  - b. La semantica di sessione usa molte copie modificabili di un file.
  - c. La robustezza di un file può essere garantita dal mirroring dei dischi.
  - d. Il file caching ha esattamente lo stesso effetto della migrazione dei file, ovvero causa lo spostamento dei file tra i nodi del sistema.
  - e. Il caching delle directory migliora le prestazioni degli accessi ai file effettuati in un file system distribuito.
  - f. I guasti che si verificano in un file server durante un'operazione di accesso ai file possono essere tollerati usando file server stateless.
- 19.2. Selezionare l'alternativa appropriata per ciascuna delle seguenti domande.
  - a. I file system distribuiti utilizzano il file caching per garantire prestazioni elevate durante le operazioni di accesso ai file. Quale semantica della condivisione dei file richiede il minor sovraccarico per la validazione della cache?
    - i. Semantica di sessione
    - ii. Semantica Unix
    - iii. Semantica delle transazioni.
  - b. La replicazione dei file migliora:
    - i. la robustezza del file system;
    - ii. la possibilità di effettuare operazioni di ripristino sul file system;
    - iii. la disponibilità di servizio del file system;
    - iv. nessuna delle alternative (i)-(iii).

## Problemi

- 19.1. Descrivere come sia possibile implementare la semantica di sessione.
- 19.2. Un FSD deve mantenere i file buffer sul nodo del server o del client? Qual è l'influenza di tale decisione sulla semantica Unix (Paragrafo 13.10) e sulla semantica di sessione?
- 19.3. Giustificare la seguente affermazione: "Il file caching si adatta bene alla semantica di sessione, ma non alla semantica Unix".
- 19.4. Descrivere le tecniche presentate in questo capitolo e nel [Capitolo 13](#) che possono essere usate per garantire la robustezza di un file.
- 19.5. Descrivere come un client possa proteggersi da eventuali fallimenti che possono verificarsi in un sistema distribuito usando (a) file server stateful, (b) file server stateless.
- 19.6. Quali sono i pregi e i difetti che si possono avere usando un file server multithread per gestire le richieste di accesso a file effettuate da client differenti?
- 19.7. Descrivere le principali problematiche che devono essere gestite durante il ripristino di un nodo in seguito a un malfunzionamento, in un sistema che usa la replicazione dei file per garantire la disponibilità del servizio.
- 19.8. Descrivere come le tecniche di locking possono essere usate per ridurre il sovraccarico richiesto dalle tecniche di validazione della cache e per migliorare la scalabilità di un sistema distribuito.

## Note bibliografiche

Sia Svobodova (1986), che Levy e Silberschatz (1990) hanno riassunto lo stato dell'arte relativo ai file system distribuiti. Comer e Peterson (1986) si sono invece focalizzati sui concetti relativi al naming e ai meccanismi di risoluzione dei pathname.

Lampson (1983) e Terry (1987) hanno presentato l'uso degli hint per migliorare le prestazioni di un file system distribuito. Makaroff ed Eager (1990) hanno invece analizzato l'effetto della dimensione della cache sulle prestazioni di un file system.

Brownbridge e altri autori (1982) hanno proposto il sistema Unix United, uno dei primi file system di rete. Sandberg (1987) e Callaghan (2000) hanno invece presentato il Network File System (NFS) di Sun. Satyanarayanan (1990) ha descritto il file system distribuito Andrew, mentre Kistler e Satyanarayanan (1992) hanno presentato il file system Coda. Braam e Nelson (1999) hanno concentrato la loro attenzione sull'analisi dei colli di bottiglia che possono presentarsi in Coda e in Intermezzo, un'versione successiva di Coda che implementa il journaling. Russinovich e Solomon (2005) hanno descritto le caratteristiche relative alla replicazione e alla distribuzione dei dati nel file system di Windows.

Thekkath e altri autori (1997) hanno presentato un file system distribuito scalabile per cluster di calcolatori. Preslan e altri autori (2000) hanno descritto tecniche di tolleranza ai guasti basate su journaling per file system su cluster. Carns e altri autori (2000) hanno proposto un file system parallelo che garantisce un'elevata quantità di banda per accessi concorrenti ai dati di file condivisi. Infine, Schmuck e Haskin (2002) hanno descritto l'uso di dischi condivisi in un file system parallelo, nonché tecniche distribuite per la sincronizzazione e la tolleranza ai guasti.

1. Braam, P.J., and P.A. Nelson (1999): "Removing bottlenecks in distributed file systems: Coda and InterMezzo as examples", *Proceedings of Linux Expo, 1999*.
2. Brownbridge, D.R., L.F. Marshall, and B. Randell (1982): "The Newcastle Connection or UNIXes of the World Unite!", *Software - Practice and Experience*, **12** (12), 1147-1162.
3. Callaghan, B. (2000): *NFS Illustrated*, Addison-Wesley, Reading, Mass.
4. Carns, P.H., W.B. Ligon III, R.B. Ross, and R. Thakur (2000): "PVFS: A parallel file system for Linux Clusters", *2000 Extreme Linux Workshop*.
5. Comer, D., and L.L. Peterson (1986): "A model of name resolution in distributed mechanisms", *Proceedings of the 6th International Conference on Distributed Computing Systems*, 509-514.
6. Ghemawat, S., H. Gobioff, and S.T. Leung (2003): "The Google file system", *Proceedings of the 19th ACM Symposium on Operating System Principles*, 29-43.

7. Gray, C.G., and D.R. Cheriton (1989): "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 202-210.
8. Kistler, J.J., and M. Satyanarayanan (1992): "Disconnected operation in the Coda file system", *ACM Transactions on Computer Systems*, **10**, 1, 3-25.
9. Lampson, B.W. (1983): "Hints for computer system designers", *Proceedings of the 9th Symposium of Operating Systems Principles*, 33-48.
10. Levy, E., and A. Silberschatz (1990): "Distributed File Systems: Concepts and Examples", *Computing Surveys*, **22** (4), 321-374.
11. Melamed, A.S. (1987): "Performance analysis of Unix-based network file systems", *IEEE Micro*, 25-38.
12. Makaroff, D.J., and D.L. Eager (1990): "Disk cache performance for distributed systems", *Proceedings of the 10th International Conference on Distributed Computing Systems*, 212-219.
13. Preslan, K.W., A.P. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S.V. Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal (2000): "Implementing journaling in a Linux shared disk file system", *Proceedings of the 7th IEEE Symposium on Mass Storage Systems*, 351-378.
14. Russinovich, M.E., and D.A. Solomon (2005): *Microsoft Windows Internals*, 4th ed., Microsoft Press, Redmond, Wash.
15. Sandberg, R. (1987): *The Sun Network File System: Design, Implementation, and experience*, Sun Microsystems, Mountain View, Calif.
16. Satyanarayanan, M. (1990): "Scalable, secure, and highly available distributed file access", *Computer*, **23** (5), 9-21.
17. Schmuck, F., and R. Haskin (2002): "GPFS: A shared-disk file system for large computing clusters", *Proceedings of the First USENIX Conference on File and Storage Technologies*, 231-244.
18. Svobodova, L. (1986): "File servers for network-based distributed systems", *Computing Surveys*, **16** (4), 353-398.
19. Terry, D.B. (1987): "Caching hints in distributed systems", *IEEE Transactions on Software Engineering*, **13** (1), 48-54.
20. Thekkath, C.A., T. Mann, and E.K. Lee (1997): "Frangipani: A scalable DFS", *Proceedings of the 16th ACM symposium on Operating System Principles*, 224-237.

---

# CAPITOLO 20

## Sicurezza nei sistemi distribuiti

---

### Obiettivi di apprendimento

- Introduzione alla sicurezza di un sistema distribuito
- Attacchi alla sicurezza di un sistema distribuito
- Politiche di sicurezza dei messaggi
- Introduzione alle tecniche di crittografia
- Politiche di autenticazione
- Firma digitale
- Kerberos e SSL

I processi di un sistema operativo distribuito utilizzano la rete per accedere alle risorse remote e per comunicare gli uni gli altri. L'infrastruttura di rete può coinvolgere canali di comunicazione pubblici e sistemi di calcolo, detti *communication processor*, che non sono sotto il controllo diretto del sistema operativo distribuito. Pertanto, è possibile che un *intruso* posizionato all'interno di un *communication processor* sia in grado di eseguire operazioni dannose, quali: corrompere i messaggi che transitano nella rete, danneggiare le operazioni eseguite dai processi, creare messaggi falsi facendosi passare per un altro utente (furto di identità) in modo da poterne usare le risorse.

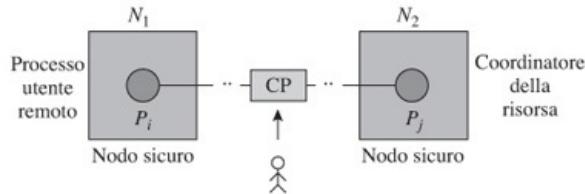
Per prevenire che eventuali intrusi manomettano i messaggi scambiati dai processi, i sistemi operativi distribuiti utilizzano speciali tecniche volte a garantire la sicurezza dei messaggi. La componente principale di tali tecniche è costituita dalla *crittografia*.

Tuttavia, l'utilizzo di quest'ultima richiede che il sistema sia in grado di difendersi da eventuali attacchi volti a violare proprio il meccanismo di crittografia, e che i processi siano a conoscenza delle chiavi usate per cifrare i messaggi che vengono inviati sulla rete. Queste problematiche sono affrontate tramite l'uso di approcci basati su *chiave pubblica* oppure su *chiavi di sessione* che vengono distribuite in modo sicuro ai vari processi del sistema. Inoltre, per prevenire furti di identità, i sistemi operativi distribuiti forniscono mezzi di autenticazione basati su terze parti la cui identità è assolutamente certa.

In questo capitolo, verranno descritte le tecniche messe a disposizione dai sistemi operativi distribuiti per garantire l'autenticazione dei processi e degli utenti, e la sicurezza dei messaggi. Inoltre, verrà presentato come sia possibile garantire l'integrità e l'autenticità dei dati usando, rispettivamente, *codici di autenticazione* e *firme digitali*.

### 20.1 Problematiche di sicurezza in un sistema operativo distribuito

Nel seguito indicheremo come *sicuri* i nodi che sono sotto il controllo diretto del sistema operativo distribuito. Essi contengono risorse e offrono servizi agli utenti e ai processi del sistema. Come mostrato nella [Figura 20.1](#), un processo utente accede a una risorsa remota inviando un messaggio al processo che la coordina. Tale messaggio può attraversare reti pubbliche e *communication processor* che agiscono sotto il controllo di sistemi operativi locali. In particolare, i *communication processor* utilizzano il modello *store-and-forward* (ovvero memorizza e inoltra) per indirizzare un messaggio verso la sua destinazione. Pertanto, i messaggi scambiati dai processi del sistema sono esposti al rischio che entità esterne possano leggerne il contenuto e possano interferire con quest'ultimo. Tale situazione può creare rischi per la sicurezza che generalmente non esistono nel caso di sistemi convenzionali.



**Figura 20.1** Attacco alla sicurezza di una rete.

Esistono quattro tipologie principali di attacchi alla sicurezza di un sistema operativo distribuito.

1. *Fuga di notizie (leakage)*: rilascio di informazioni contenute nei messaggi a utenti non autorizzati.
2. *Manomissione (tampering)*: modifiche al contenuto dei messaggi.
3. *Furto (stealing)*: utilizzo di risorse del sistema senza l'opportuna autorizzazione.
4. *Negazione del servizio (denial of service)*: distruzione intenzionale delle risorse di un sistema od operazioni che ne impediscono l'accesso da parte di utenti autorizzati.

*Leakage* e *tampering* sono attacchi alla *sicurezza dei messaggi*. Le tecniche di *tampering* possono essere usate sia per modificare il testo di un messaggio, in tal caso si parla di attacco all'*integrità* del messaggio, sia per modificare l'*identità* del mittente, caso in cui si parla invece di attacco all'*autenticità* del mittente. Un intruso può essere in grado di accedere alle risorse di un sistema senza la necessaria autorizzazione (*stealing*), utilizzando tecniche di *tampering* per mascherare la propria identità. Inoltre, è possibile effettuare attacchi di tipo *denial of service* usando tecniche di *tampering* per manomettere il testo di un messaggio o gli identificatori dei relativi processi mittente e destinazione, oppure mascherando la propria identità.

Per contrastare le tipologie di attacco precedentemente descritte esistono due strategie alternative basate sull'uso di:

- *tecniche per garantire la sicurezza dei messaggi*: si tratta di approcci mirati a contrastare gli attacchi nei confronti dell'*integrità* dei messaggi;
- *tecniche per l'autenticazione degli utenti remoti*: si tratta di approcci mirati a garantire con certezza l'*autenticità* degli utenti remoti che accedono alle risorse del sistema.

Gli attacchi volti sia all'*integrità* dei messaggi che all'*autenticità* degli utenti vengono contrastati usando strategie che combinano entrambi gli approcci.

### 20.1.1 Politiche e meccanismi per garantire la sicurezza

La [Figura 20.2](#) mostra le politiche e i meccanismi che vengono utilizzati per garantire la sicurezza in un sistema distribuito. Le tecniche di *autenticazione* impiegate in un sistema convenzionale sono state descritte nel [Capitolo 15](#). A differenza di queste ultime, le tecniche di autenticazione impiegate nei sistemi distribuiti devono tener conto anche del fatto che il servizio di autenticazione deve essere affidabile e disponibile per tutti i nodi del sistema. Per garantire la segretezza e l'*integrità* dei database usati per le operazioni di autenticazione e autorizzazione viene utilizzata la *crittografia*. Quest'ultima viene impiegata anche per codificare i messaggi in modo da garantire la loro sicurezza. Di conseguenza, i processi, per poter comunicare tra loro, devono conoscere le chiavi usate dall'algoritmo di crittografia impiegato per codificare i messaggi. Nel [Paragrafo 20.2.1](#) verrà presentato il meccanismo utilizzato per generare e distribuire tali chiavi.



**Figura 20.2** Meccanismi e politiche usate per garantire la sicurezza in un sistema distribuito.

### 20.1.2 Attacchi alla sicurezza di un sistema distribuito

Gli attacchi alla sicurezza di un sistema distribuito, effettuati generalmente tramite scambio di messaggi, possono essere classificati nelle quattro classi riassunte nella **Tabella 20.1**. Gli attacchi basati su intercettazione (*eavesdropping*) possono avere diversi obiettivi, come, per esempio, quello di catturare il contenuto di un messaggio, oppure di raccogliere informazioni sulle attività di scambio di messaggi che intercorrono tra i processi del sistema o che utilizzano determinati canali di comunicazione. In particolare, quest'ultimo tipo di intercettazioni può essere usato in un sistema informativo militare per scoprire l'identità delle parti comunicanti. Gli attacchi basati su *manomissione dei messaggi* (*message tampering*) possono invece essere utilizzati per ingannare il destinatario di un messaggio. Questa tipologia di attacco è particolarmente efficace nelle reti di tipo store-and-forward.

| Attacco                  | Descrizione                                                                                                                                                                                                                                                |
|--------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Intercettazione          | L'intruso si mette in ascolto sulla rete per catturare il contenuto dei messaggi scambiati dai vari processi ed eventualmente derivare informazioni statistiche sulle loro caratteristiche.                                                                |
| Manomissione di messaggi | L'intruso intercetta i messaggi, ne altera il contenuto, e infine li reinserirà nel flusso di comunicazione                                                                                                                                                |
| Replicazione di messaggi | L'intruso crea copie dei messaggi scambiati tra i vari processi del sistema. Quindi, in istanti successivi, inserisce le copie nel flusso di comunicazione facendo credere ai processi riceventi che si tratta di nuovi messaggi originali appena spediti. |
| Scambio di identità      | L'intruso si finge un utente autorizzato in modo da poter accedere alle risorse del sistema, inviare e ricevere messaggi dagli altri processi.                                                                                                             |

**Tabella 20.1.** Tipologie di attacco alla sicurezza in un sistema distribuito.

Gli attacchi basati su replicazione dei messaggi (*message replay*) possono essere usati per svariati scopi. Un messaggio replicato può essere scambiato per un nuovo messaggio appena spedito. Se il ricevente è un processo utente, quest'ultimo potrebbe essere indotto a effettuare azioni non necessarie, illogiche o particolarmente dispendiose per le risorse coinvolte. Inoltre, è possibile che un messaggio replicato induca a rivelare informazioni confidenziali. Se, per esempio, il ricevente è un processo server, un messaggio replicato può causare errori nelle operazioni di autenticazione, favorendo, così, attività volte a mascherare l'identità di un intruso, o l'accesso da parte di quest'ultimo a risorse del sistema senza avere la necessaria autorizzazione.

Negli attacchi basati su scambio di identità (*masquerading*), l'intruso finge di essere un utente del sistema. Esso può pertanto corrompere o distruggere le informazioni relative al vero utente oppure comunicare con altri processi sotto mentite spoglie.

#### Attacchi passivi e attivi

Gli attacchi alla sicurezza possono essere di natura *passiva* o *attiva*. I primi non interferiscono con la funzionalità del sistema attaccato, e quindi non creano messaggi aggiuntivi, né distruggono quelli inviati lecitamente dagli utenti autorizzati. Le tecniche

basate su intercettazione sono un esempio di attacco passivo. Al contrario, un attacco attivo interferisce con il funzionamento del sistema attaccato replicando, modificando o distruggendo i messaggi scambiati dai suoi utenti o creandone di nuovi. Gli attacchi passivi sono più difficili da identificare e contrastare rispetto a quelli attivi.

## 20.2 Sicurezza dei messaggi

Gli approcci volti a garantire la sicurezza dei messaggi scambiati in un sistema distribuito possono essere classificati in *tecniche link-oriented* e *tecniche end-to-end*. Nelle tecniche link-oriented vengono prese precauzioni, atte a garantire la sicurezza, su tutti i canali (*link*) che costituiscono il cammino di comunicazione. Questa tipologia di approccio tende a essere particolarmente costosa, poiché il suo costo dipende dal numero di canali di comunicazione attraversati dai messaggi di cui si vuole garantire la sicurezza. Per esempio, se un messaggio inviato dal processo  $P_i$  nel nodo  $N_1$  al processo  $P_j$  nel nodo  $N_3$  deve attraversare il cammino  $N_1-N_2-N_3$ , il sistema subisce un sovraccarico per gestire sia la sicurezza del link  $N_1-N_2$  che quella del link  $N_2-N_3$ . Negli approcci end-to-end, invece, i nodi e i processi del sistema possono usare accorgimenti volti a garantire la sicurezza di quest'ultimo in modo selettivo. Di conseguenza, gli utenti possono scegliere tra un'ampia gamma di strategie che si differenziano per costo e sofisticatezza delle metodologie impiegate. Nel prosieguo del paragrafo, si assumerà l'uso di soli approcci end-to-end.

Nel seguito, verranno descritte tre strategie volte a garantire la sicurezza dei messaggi, che fanno uso di tecniche di crittografia basate, rispettivamente, su chiavi pubbliche, chiavi private e chiavi di sessione. La [Tabella 20.2](#) riassume le loro caratteristiche.

### **Crittografia a chiave privata**

La crittografia a chiave privata (chiamata anche *crittografia a chiave segreta*) rappresenta il classico approccio basato su chiavi simmetriche. Ogni processo  $P_i$  possiede una *chiave privata*  $V_i$  che è nota, oltre a  $P_i$  stesso, a tutti i processi del sistema che desiderano comunicare con esso. Infatti, un processo che desidera inviare un messaggio a  $P_i$  deve cifrarlo usando  $V_i$ . La stessa chiave  $V_i$  è quindi usata da  $P_i$  per decifrare i messaggi ricevuti. Il principale vantaggio di tale strategia consiste nel fatto che in un sistema con  $n$  entità comunicanti basta avere  $n$  chiavi private. Dal momento che tutti i messaggi inviati a un processo  $P_i$  sono cifrati con la stessa chiave,  $P_i$  non ha bisogno di conoscere l'identità dei processi mittenti per poter leggere i messaggi ricevuti.

D'altra parte, la crittografia a chiave privata presenta alcuni svantaggi. I processi mittenti devono conoscere la chiave privata dei rispettivi processi destinazione. Pertanto, la chiave privata di ogni processo può essere nota a molti altri processi e quindi un eventuale intruso ha maggior possibilità di scoprirla. La chiave privata è inoltre costantemente esposta agli attacchi dell'intruso, e la possibilità che quest'ultimo riesca nel suo intento aumenta con il passare del tempo. D'altra parte non è possibile cambiare la chiave privata di un processo, perché ciò comporterebbe il dover comunicare il nuovo valore della chiave a un numero elevato di processi.

I processi utente non conoscono le rispettive chiavi private, pertanto la crittografia a chiave privata non è generalmente utilizzabile per garantire la sicurezza dei messaggi inviati tra processi utente. Al contrario, i processi del sistema operativo conoscono le chiavi private dei processi utente. Quindi, la crittografia a chiave pubblica viene usata per la comunicazione tra i processi del sistema operativo e i processi utente. Come descritto nel Paragrafo 20.2.1, tale caratteristica è usata anche per implementare centri di distribuzione delle chiavi. I processi utente usano invece altri schemi di crittografia per comunicare tra loro.

| Tecnica                           | Descrizione                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Crittografia a chiave privata     | Utilizza la crittografia simmetrica. Ogni processo $P_i$ possiede un'unica chiave di crittografia $V_i$ , chiamata <i>chiave privata</i> . Tutti i messaggi inviati a $P_i$ devono essere cifrati usando $V_i$ in modo tale che $P_i$ possa decifrarli usando la stessa chiave. Va osservato che una chiave può essere esposta ad attacchi da parte di eventuali intrusi pertutta la vita del processo.                                                                                                                                |
| Crittografia a chiave pubblica    | Utilizza la crittografia asimmetrica. Ogni processo $P_i$ possiede una coppia di chiavi $(U_i, V_i)$ . $U_i$ è la <i>chiave pubblica</i> , e pertanto deve essere nota a tutti i processi del sistema. $V_i$ , invece, è la <i>chiave privata</i> , nota solo a $P_i$ . I messaggi inviati a $P_i$ devono essere cifrati usando $U_i$ , mentre $P_i$ usa $V_i$ per decifrarli. Per creare la coppia di chiavi $(U_i, V_i)$ , si usa generalmente l'algoritmo Rivest-Shamir-Adelman (RSA). La chiave privata non è esposta ad attacchi. |
| Crittografia a chiave di sessione | Ogni coppia di processi $(P_i, P_j)$ che inizia una sessione di comunicazione riceve una chiave di sessione $SK_{ij}$ . La chiave di sessione viene usata per cifrare e decifrare tutti i messaggi scambiati durante la sessione. La chiave di sessione ha un tempo di vita più breve rispetto alla chiave pubblica e alla chiave privata, pertanto essa è meno soggetta ad attacchi da parte di eventuali intrusi.                                                                                                                    |

**Tabella 20.2** Tecniche di crittografia usate per garantire la sicurezza dei messaggi.

### Crittografia a chiave pubblica

Ogni processo  $P_i$  possiede una coppia di chiavi  $(U_i, V_i)$ .  $U_i$  è la *chiave pubblica* e deve essere nota a tutti i processi che vogliono comunicare con  $P_i$ .  $V_i$ , invece è la *chiave privata* nota solo a  $P_i$ .  $U_i$ , e  $V_i$  vengono scelte in modo che:

- $V_i$  non possa essere derivata a partire da  $U_i$ , e
- per ogni messaggio  $m$

$$D_{Vi}(E_{Ui}(P_m)) = P_m \quad \forall i \quad (20.1)$$

dove  $P_m$  rappresenta il testo non cifrato del messaggio  $m$  mentre  $E$  e  $D$  sono, rispettivamente, le funzioni di cifratura e decifratura (si veda il Paragrafo 15.4).

Quando un processo  $P_j$  desidera inviare un messaggio a  $P_i$ , il sistema operativo fornisce a  $P_j$  la chiave pubblica di  $P_i$ . Quindi, la trasmissione del messaggio avviene come segue:

1. il processo  $P_j$  cifra il messaggio con la chiave pubblica  $U_i$  di  $P_i$ ;
2. il messaggio cifrato, cioè  $E_{Ui}(P_m)$ , viene trasmesso attraverso la rete e viene ricevuto da  $P_i$
3. il processo  $P_i$  decifra il messaggio ricevuto con la sua chiave privata, cioè  $V_i$ . Per far ciò  $P_i$  calcola  $D_{Vi}(E_{Ui}(P_m))$ , il cui risultato è  $P_m$ .

Per generare coppie di chiavi  $(U_i, V_i)$  che soddisfino l'Equazione 20.1, viene usato l'algoritmo Rivest-Shamir-Adelman (RSA). Supponiamo che  $(u, v)$  sia una coppia di chiavi. Dati due numeri  $x$  e  $y$ , entrambi più piccoli di un numero  $n$ , la cifratura e la decifratura usando  $u$  e  $v$  si ottiene come segue:

$$E_u(x) = x^u \bmod n$$

$$D_v(y) = y^v \bmod n$$

Per cifrare e decifrare un messaggio  $m$ , l'algoritmo RSA viene usato come un cifrario con blocchi di dimensione  $s$ , tale che  $2^s < n$ .  $x$  è quindi il numero formato dalla stringa di bit

costituita da un blocco di  $P_m$  (dove  $P_m$  rappresenta il testo non cifrato del messaggio  $m$ ), e  $y$  è il numero formato dalla stringa di bit costituita dal corrispondente blocco di  $C_m$  (dove  $C_m$  rappresenta il testo cifrato del messaggio  $m$ ). In tal modo,  $x < 2^s$  e  $y < 2^s$ , e quindi sia  $x$  che  $y$  sono più piccoli di  $n$ , come richiesto.

L'algoritmo RSA sceglie come numero  $n$  il prodotto di due numeri primi molto grandi  $p$  e  $q$ . Tipicamente,  $p$  e  $q$  sono composti da 100 cifre ciascuno, e quindi  $n$  è un numero con 200 cifre. Assumendo che  $u$  e  $v$  siano le corrispondenti chiavi pubblica e privata, per soddisfare l'equazione (20.1)  $v$  dovrebbe essere un numero primo rispetto a  $(p - 1) \times (q - 1)$ , cioè  $v$  e  $(p - 1) \times (q - 1)$  non dovrebbero avere fattori comuni eccetto il numero 1, e  $u$  dovrebbe soddisfare la relazione:

$$u \times v \bmod [(p - 1) \times (q - 1)] = 1$$

La scelta di  $u$  e  $v$  come chiavi pubblica e privata implica che nel sistema venga usato un valore standard per  $n$  per tutti i processi. In alternativa, è possibile usare la coppia  $(u, n)$  come chiave pubblica e la coppia  $(v, n)$  come chiave privata. In tal modo è possibile usare valori di  $n$  diversi per ogni processo.

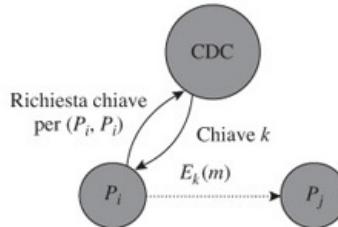
Un attacco a un messaggio cifrato con l'algoritmo RSA può aver successo se  $p$  e  $q$  sono fattori di  $n$ . Tuttavia, si stima che per calcolare la fattorizzazione di un numero di 200 cifre, operazione necessaria se si desidera violare l'algoritmo, siano richiesti 4 miliardi di anni usando un computer che sia in grado di effettuare 1 milione di operazioni al secondo. La crittografia con chiave pubblica presenta alcuni svantaggi rispetto agli approcci basati su chiave privata. Le chiavi usate nella crittografia a chiave pubblica hanno una dimensione maggiore rispetto a quelle usate nella crittografia a chiave privata di circa un ordine di grandezza. Ciò è inevitabile, in quanto le chiavi pubbliche devono essere sufficientemente grandi da rendere proibitivo il costo per una loro fattorizzazione. Inoltre, le operazioni di codifica e decodifica effettuate con chiave pubblica sono molto costose se comparate con le corrispondenti operazioni effettuate nella crittografia simmetrica; in molte situazioni, le prime possono essere 1000 volte più lente rispetto alle seconde. Perciò, la crittografia a chiave pubblica non è particolarmente conveniente da usare durante lo scambio di messaggi tra processi. Al contrario, essa può essere usata per comunicare in modo sicuro una chiave di sessione a una coppia di processi che intendono iniziare una sessione di comunicazione. Questo aspetto è discusso nel prossimo paragrafo.

### **Chiavi di sessione**

Nella crittografia simmetrica basata su chiave di sessione, ogni coppia di processi  $P_i$  e  $P_j$ , che desidera iniziare una comunicazione, riceve una *chiave di sessione*, chiamata anche *chiave di conversazione*. Essa viene usata per tutta la durata della sessione e viene eliminata al termine di quest'ultima. Se, successivamente, i processi desiderano iniziare una nuova sessione di comunicazione, gli verrà assegnata una nuova chiave. Questo approccio limita l'esposizione della chiave usata per codificare e decodificare i messaggi a eventuali intrusi, riducendo così il conseguente rischio di violazione del sistema.

### **20.2.1 Distribuzione delle chiavi**

I processi hanno bisogno di conoscere le chiavi da usare per comunicare tra loro. A tale scopo, il sistema operativo fornisce un servizio interattivo chiamato *centro per la distribuzione delle chiavi* (CDC). La Figura 20.3 mostra lo schema di un CDC. Il processo  $P_i$  dopo aver chiesto una chiave al CDC riceve da quest'ultimo  $k$ . Quindi,  $P_i$  usa  $k$  per cifrare il messaggio  $m$  prima di inviarlo al processo  $P_j$ . Se i processi usano chiavi pubbliche per comunicare tra loro, il CDC mantiene una lista con le chiavi pubbliche di tutte le entità del sistema. Se invece i processi usano chiavi private, il CDC non mantiene alcuna lista, ma genera una nuova chiave di sessione a ogni richiesta.



**Figura 20.3** Centro per la distribuzione delle chiavi (CDC) di un sistema operativo distribuito.

Un aspetto importante relativo alla funzionalità di un CDC riguarda il protocollo usato per condividere le chiavi in modo sicuro. Quando viene richiesta una chiave pubblica, quest'ultima deve essere fornita solo al richiedente. Quando invece un processo  $P_i$  richiede una chiave di sessione per comunicare con un processo  $P_j$ , la chiave deve essere comunicata sia a  $P_i$  che a  $P_j$ . Tuttavia, visto che  $P_j$  non sa che  $P_i$  desidera iniziare una comunicazione, il CDC non invia la chiave di sessione direttamente a  $P_j$ . Sarà invece  $P_i$  a inviare la chiave a  $P_j$  assieme al primo dei suoi messaggi. I protocolli per la trasmissione delle chiavi sono presentati nei paragrafi seguenti.

### Distribuzione di chiavi pubbliche

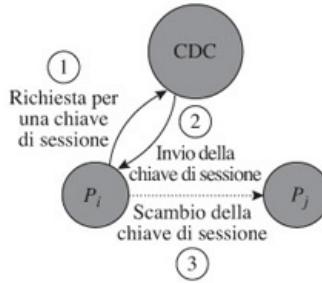
Per la distribuzione della chiave pubblica, il processo richiedente  $P_i$  e il CDC si scambiano i seguenti messaggi:

1.  $P_i \rightarrow \text{CDC} : E_{U_{cdc}}(P_i, P_j)$
  2.  $\text{CDC} \rightarrow P_i : E_{U_i}(P_j, U_j)$
- (20.2)

$P_i$  invia al CDC un messaggio contenente il suo identificatore e quello del processo destinazione, cioè  $P_j$ . Tale messaggio viene cifrato con  $U_{cdc}$ , la chiave pubblica del CDC. Quest'ultimo risponde inviando la chiave pubblica di  $P_j$ , ovvero  $U_j$ , cifrata con la chiave pubblica di  $P_i$ . In questo caso, la crittografia non è usata per proteggere la chiave di  $P_j$ , poiché un eventuale intruso può legittimamente ottenerla chiedendola direttamente al CDC. Al contrario, la cifratura viene usata per prevenire che un intruso manometta i messaggi scambiati tra  $P_i$  e CDC con lo scopo di effettuare un attacco di tipo déniel of service. In assenza di cifratura, un eventuale intruso potrebbe manomettere i messaggi inviati da  $P_i$  al CDC scambiando l'identificatore di  $P_i$  con quello di qualche altro processo  $P_k$ , in modo che  $P_j$  non ottenga la chiave di  $P_j$ . Oppure, l'intruso potrebbe manomettere i messaggi inviati dal CDC a  $P_i$  cambiando la chiave fornita dal CDC per  $P_j$ . Usando la cifratura, invece, sia il CDC che il processo  $P_i$  sono in grado di riconoscere e scartare eventuali messaggi manomessi.

### Distribuzione delle chiavi di sessione

Quando un processo  $P_i$  richiede una chiave di sessione per comunicare con un altro processo  $P_j$ , è necessario che la chiave venga resa nota sia a  $P_i$  che a  $P_j$ . Per ottenere ciò è possibile eseguire i tre passi mostrati in [Figura 20.4](#). Nel primo passo,  $P_i$  invia al CDC un messaggio contenente il suo identificatore e quello di  $P_j$ . Nel secondo passo, il CDC alloca una chiave di sessione  $SK_{i,j}$  per i processi  $P_i$  e  $P_j$  e la invia a  $P_i$  assieme a un dato cifrato, contenente  $SK_{i,j}$ , che può essere decifrato solo da  $P_j$ . Infine, nel terzo passo,  $P_i$  invia a  $P_j$  il dato contenente la chiave cifrata all'interno del primo messaggio della sessione, o in un messaggio specificatamente creato per tale scopo. In tal modo,  $P_j$  riceve  $SK_{i,j}$  e può comunicare in modo sicuro con  $P_i$  per tutta la durata della sessione.



**Figura 20.4** Distribuzione delle chiavi di sessione.

In un sistema basato su chiave privata, i tre passi del precedente meccanismo possono essere implementati come segue:

1.  $P_i \rightarrow \text{CDC} : P_i, P_j$
2.  $\text{CDC} \rightarrow P_i : E_{V_i}(P_j, SK_{i,j}), E_{V_j}(P_i, SK_{i,j})$
3.  $P_i \rightarrow P_j : E_{V_j}(P_i, SK_{i,j}), E_{SK_{i,j}}(<\text{messaggio}>)$  (20.3)

In particolare, nel secondo passo del Protocollo 20.3, il CDC invia a  $P_i$  una risposta cifrata con la chiave privata di  $P_i$ . Tale risposta contiene la chiave di sessione  $SK_{i,j}$  e il dato  $E_{V_j}(P_i, SK_{i,j})$ , contenente la stessa chiave di sessione cifrata con la chiave privata di  $P_j$ .  $P_i$  decifra il messaggio del CDC con la sua chiave privata e recupera la chiave di sessione  $SK_{i,j}$  e il dato  $E_{V_j}(P_i, SK_{i,j})$ . Quindi,  $P_i$  invia il dato  $E_{V_j}(P_i, SK_{i,j})$  all'interno del primo messaggio indirizzato a  $P_j$ . Di conseguenza,  $P_j$  decifrando tale dato può venire a conoscenza della chiave  $SK_{i,j}$  da usare per decifrare tutti i successivi messaggi di  $P_i$ .

Al contrario, in un sistema a chiave pubblica, le chiavi di sessione non devono essere distribuite dal CDC, infatti, ogni processo mittente può scegliere da solo una propria chiave di sessione. Basta solo che la chiave scelta, sia successivamente comunicata in modo sicuro al processo destinazione. Per fare ciò, è necessario inviare al processo destinazione un messaggio, contenente la chiave di sessione, cifrato con la chiave pubblica di quest'ultimo. Pertanto, il processo mittente  $P_i$  può usare il seguente protocollo per comunicare la chiave di sessione al processo destinazione  $P_j$ .

1.  $P_i \rightarrow \text{CDC} : E_{U_{cdc}}(P_i, P_j)$
2.  $\text{CDC} \rightarrow P_i : E_{U_i}(P_j, U_j)$
3.  $P_i \rightarrow P_j : E_{U_j}(P_i, SK_{i,j}), E_{SK_{i,j}}(<\text{messaggio}>)$  (20.4)

I primi due passi del Protocollo 20.4 sono identici ai primi due passi del Protocollo 20.2; essi infatti forniscono a  $P_i$  la chiave pubblica di  $P_j$ , in modo che  $P_i$  stesso possa generare una chiave di sessione  $SK_{i,j}$  e inviarla a  $P_j$ , durante il passo 3, all'interno del primo messaggio della sessione.

## 20.2.2 Prevenzione degli attacchi basati su replicazione dei messaggi

Durante un *attacco basato su replicazione dei messaggi*, l'intruso copia i messaggi che passano sulla rete per reintrodurli in essa in un secondo momento. Un messaggio replicato può confondere il suo destinatario inducendolo a effettuare operazioni sbagliate o a ripetere azioni che possono influenzare la consistenza dei dati o rivelare informazioni confidenziali. Per esempio, in un sistema che usa chiavi di sessione, l'intruso potrebbe replicare il messaggio inviato nel passo 3 del Protocollo 20.3 o del Protocollo 20.2. Alla ricezione del messaggio replicato,  $P_j$  è indotto a pensare che  $P_i$  stia cercando di comunicare con esso usando la chiave di sessione  $SK_{i,j}$ . Quando  $P_j$  risponde a tale messaggio, l'intruso può replicare il successivo messaggio precedentemente copiato. In tal modo, l'intruso può replicare l'intera sessione.

Il destinatario di un messaggio può utilizzare il *protocollo challenge-response*

(letteralmente *sfida-risposta*) per verificare se il messaggio ricevuto è stato realmente inviato dal mittente o se si tratta di un messaggio replicato. I passi di tale protocollo sono i seguenti:

- *sfida*: quando un processo  $P_j$  riceve un messaggio creato da un processo  $P_i$ ,  $P_j$  invia a  $P_i$  una sfida (*challenge*) per chiedergli di dimostrare che sta effettivamente scambiando messaggi con  $P_j$  in tempo reale. La sfida è costituita da un messaggio contenente una stringa, detta *challenge string*, cifrata in modo tale che solo  $P_i$  sia in grado di decifrarla;
- *risposta*: alla ricezione del messaggio di sfida,  $P_i$  dovrebbe decifrare la stringa e modificarla come atteso da  $P_j$ . Quindi,  $P_i$  dovrebbe inviare a  $P_j$  la stringa modificata, opportunamente cifrata in modo che solo  $P_j$  sia in grado di decifarla;
- *rilevazione*: alla ricezione della risposta inviata da  $P_i$ ,  $P_j$  decifra il messaggio ricevuto e verifica se il contenuto è conforme alle sue aspettative. Un'eventuale discordanza indica che è in atto un attacco basato su replicazione dei messaggi.

Come challenge string è possibile usare un numero  $n$ , in tal caso la risposta attesa da parte del processo che viene messo alla prova sarà il risultato di una semplice trasformazione effettuata su  $n$ , quale, per esempio,  $n + 1$ . Tuttavia, il processo che lancia la sfida dovrebbe usare numeri sempre diversi in modo che l'eventuale replicazione di una vecchia risposta, da parte dell'intruso, non fornisca proprio il valore atteso. Due possibili valori da inserire nella challenge string sono: un numero casuale, oppure il valore corrispondente all'istante di tempo corrente. Tuttavia, va osservato che il valore inserito nella challenge string non è assolutamente importante. Per questo motivo si è soliti riferirsi a tale valore con il termine *nonce*, che può essere tradotto in italiano come "occasione".

Il protocollo sfida-risposta dovrebbe essere usato in tutte le situazioni in cui un attacco basato su replicazione dei messaggi potrebbe essere pericoloso per il sistema. Per esempio, si consideri il Protocollo 20.3 utilizzato per la distribuzione delle chiavi di sessione. Un eventuale intruso potrebbe salvare il messaggio inviato al passo 3, e replicarlo in un istante successivo per indurre il processo  $P_j$  a iniziare una comunicazione con esso stesso, usando  $SK_{i,j}$ . Pertanto, prima di usare la chiave di sessione ottenuta al passo 3, è consigliabile che  $P_j$  utilizzi il protocollo sfida-risposta per assicurarsi che il messaggio ricevuto sia stato effettivamente inviato dal mittente originale in tempo reale:

4.  $P_j \rightarrow P_i : E_{SK_{i,j}}(n)$
5.  $P_i \rightarrow P_j : E_{SK_{i,j}}(n + 1)$

Nei passi precedenti  $n$  rappresenta il valore nonce inviato da  $P_j$  a  $P_i$  come sfida. Quest'ultimo, una volta decifrato il messaggio contenente il valore nonce usando la chiave di sessione  $SK_{i,j}$ , somma 1 a  $n$ , cifra il risultato con la stessa chiave  $SK_{i,j}$  e spedisce la risposta a  $P_j$ . Un eventuale intruso non sarebbe in grado di eseguire queste stesse azioni in modo corretto, in quanto non conosce  $SK_{i,j}$ . L'abilità dimostrata da  $P_i$  nell'estrare il valore  $n$  dal messaggio inviato da  $P_j$  prova implicitamente la sua identità. Questa proprietà è utile per il meccanismo di autenticazione reciproca presentato nel prossimo paragrafo.

### 20.2.3 Autenticazione reciproca

Per contrastare attacchi basati su scambio di identità, i processi coinvolti in una sessione di comunicazione dovrebbero validare reciprocamente la propria identità all'inizio della sessione. Come visto nella paragrafo precedente, il protocollo sfida-risposta permette di verificare implicitamente l'identità del processo che risponde alla sfida, e quindi può essere usato anche per implementare un meccanismo di autenticazione reciproca. Si consideri il Protocollo 20.4, utilizzato per selezionare le chiavi di sessione in un sistema basato su chiave pubblica. Nel passo 3,  $P_i$  invia la chiave di sessione a  $P_j$  all'interno di un messaggio cifrato usando la chiave pubblica di  $P_j$ . In linea di principio, qualunque processo potrebbe creare lo stesso messaggio per ingannare  $P_j$ , facendogli credere di iniziare una comunicazione con  $P_i$ . Pertanto, è essenziale che  $P_j$  sia in grado di autenticare l'identità di  $P_i$  prima di iniziare la sessione.  $P_j$  può effettuare tale verifica tramite il seguente protocollo, i cui primi tre passi sono identici a quelli del Protocollo

20.4:

1.  $P_i \rightarrow \text{CDC} : E_{Ucdc}(P_i, P_j)$
2.  $\text{CDC} \rightarrow P_i : E_{Ui}(P_j, U_j)$
3.  $P_i \rightarrow P_j : E_{Uj}(P_i, SK_{i,j})$
4.  $P_j \rightarrow P_i : E_{Ui}(P_j, n)$
5.  $P_i \rightarrow P_j : E_{Uj}(n + 1)$
6.  $P_i \rightarrow P_j : E_{SK_{i,j}}(<\text{messaggio}>)$

Nel passo 4,  $P_j$  invia il valore nonce  $n$  cifrato con la chiave pubblica di  $P_i$ . L'identità di  $P_i$  viene verificata accertando l'abilità di quest'ultimo nel decifrare tale messaggio, estrarre il valore nonce, e trasformare quest'ultimo nel modo atteso da  $P_j$ . Va osservato che, nel passo 4,  $P_j$  non deve cifrare il suo messaggio usando la chiave di sessione  $SK_{i,j}$ , poiché l'intruso potrebbe essere in grado di decifrare tale messaggio qualora sia stato proprio lui a creare il messaggio inviato nel passo 3.

## 20.3 Autenticazione di dati e messaggi

Per *autenticazione* dei dati si intende la capacità di un processo di verificare che tali dati siano stati effettivamente creati o inviati dal processo che dichiara di averlo fatto, senza che siano stati manomessi da un eventuale intruso. In particolare, quest'ultimo aspetto riguarda l'*integrità* dei dati stessi.

L'*integrità* dei dati viene garantita come segue. Quando un dato  $d$  viene creato o trasmesso, viene usata una particolare funzione di hash  $h$  per calcolare un valore  $v$ . Tale valore, chiamato anche *digest* (letteralmente, riassunto) del messaggio, ha una lunghezza predefinita indipendentemente dalla dimensione del dato. Oltre a possedere le proprietà di tipo one-way descritte precedentemente nel Paragrafo 15.4.1, la funzione di hash  $h$  rende inefficace qualunque attacco basato sul cosiddetto attacco del compleanno (meglio noto con il termine inglese *birthday attack*), ovvero dato il valore di hash  $v$  corrispondente al dato  $d$ , risulta impossibile costruire un altro dato  $d'$  il cui valore di hash sia sempre  $v$ . Il dato e il valore di hash sono memorizzati e trasmessi come una coppia  $< d, v >$ . Per verificare l'autenticità di  $d$ , il suo valore di hash viene calcolato usando  $h$ , e confrontato con  $v$ . In base alla particolare proprietà di  $h$ , precedentemente enunciata, se i due valori coincidono,  $d$  viene considerato integro, altrimenti se ne deriva che è stato manomesso. Affinché un simile approccio funzioni correttamente, anche il valore  $v$  dovrebbe essere protetto da eventuali manomissioni o sostituzioni da parte di un intruso, altrimenti, quest'ultimo potrebbe sostituire la coppia  $< d, v >$  con la coppia  $< d', v' >$  inducendo, quindi, gli altri processi a pensare che il dato originale sia  $d'$ . Di conseguenza, la persona o il processo che originariamente ha creato o trasmesso  $d$ , cifra  $v$  o la coppia  $< d, v >$ , usando la sua chiave, in modo che eventuali manomissioni o sostituzioni possano essere facilmente rilevate. Si osservi che è meno costoso cifrare  $v$  piuttosto che  $< d, v >$ .

Il processo di *autenticazione* richiede, infine, un ulteriore controllo - verificare se  $v$  e  $< d, v >$  sono stati effettivamente cifrati dalla persona o dal processo che dichiara di averlo fatto. Tale verifica, descritta dettagliatamente nel paragrafo seguente, viene effettuata per mezzo di *un'autorità di certificazione*, che fornisce in modo sicuro le informazioni relative alle chiavi di cifratura usate.

### 20.3.1 Autorità di certificazione e certificati digitali

Un'autorità di certificazione (AC) assegna chiavi pubbliche e private alle entità che ne fanno richiesta, siano esse persone o processi, dopo aver accertato la loro identità usando un opportuno meccanismo di verifica. Le chiavi assegnate rimangono valide per un periodo di tempo limitato. L'autorità di certificazione agisce anche come centro per la distribuzione delle chiavi (Paragrafo 20.2.1). Infatti, ogni autorità di certificazione preserva un database con tutte le chiavi che essa ha precedentemente assegnato. In tal modo, quando un processo richiede una chiave pubblica per comunicare con qualche entità, l'autorità di certificazione genera un *certificato di chiave pubblica* contenente le seguenti informazioni:

- numero di serie del certificato;
- distinguished name (DN) del proprietario del certificato, che contiene nome, unità,

- ubicazione, stato e nazione del proprietario, e nome del suo DNS in forma testuale;
- informazioni per l'identificazione del proprietario del certificato, come per esempio il suo indirizzo;
  - chiave pubblica del proprietario del certificato;
  - data di emissione e di scadenza e nome di chi emette il certificato;
  - firma digitale dell'autorità di certificazione relativa alle informazioni precedentemente elencate.

È consentito che operino contemporaneamente più autorità di certificazione. Pertanto, un server può chiedere che gli venga generato un certificato di chiave pubblica da una qualunque delle varie AC. Se un client conosce su quale AC è stato registrato un server, esso può chiedere a quest'ultima il certificato di chiave pubblica del server. In alternativa, se il client conosce l'indirizzo IP del server di interesse, esso può chiedere direttamente al server di inviargli il suo certificato.

Il certificato di un'entità viene richiesto per ottenere la corrispondente chiave pubblica, per mezzo della quale è possibile comunicare con l'entità stessa in modo sicuro. Tuttavia, prima di usare la chiave ricevuta, è necessario verificare che il certificato in cui essa era contenuta sia originale e appartenga effettivamente all'entità con cui si desidera comunicare; ovvero è necessario controllare che la chiave non abbia subito un attacco del tipo *man-in-the-middle* (letteralmente, uomo nel mezzo). In questa tipologia di attacco, un eventuale intruso finge di essere il server a cui il client si è rivolto per ottenere il relativo certificato. In particolare, quando il client richiede il certificato al server, l'intruso intercetta il messaggio e risponde al client inviando un certificato falso contenente la propria chiave pubblica. In seguito a ciò, il client invia al server messaggi cifrati usando la chiave pubblica dell'intruso. Quindi, quest'ultimo, dopo aver intercettato i messaggi inviati dal client al server, sarà in grado di decifrarne il contenuto usando la propria chiave privata. Contemporaneamente, l'intruso ha la possibilità di iniziare una comunicazione con il server, fingendo di essere il client, per inoltrargli i messaggi del client dopo averli letti. In tal modo né il client né il server si accorgono di essere vittime dell'attacco.

Il certificato contenente la chiave pubblica include, tuttavia, vari elementi per prevenire attacchi del tipo *man-in-the-middle*. Innanzitutto, esso riporta la firma digitale dell'autorità di certificazione, tramite la quale il client può verificare che il certificato non sia stato manomesso o falsificato (ulteriori dettagli sul concetto di certificato digitale saranno presentati nel Paragrafo 20.3.2). A tale scopo, il client necessita della chiave pubblica dell'autorità di certificazione che ha emesso il certificato. Se tale chiave non è già nota al client, quest'ultimo può richiederla a un'autorità di certificazione di livello più alto. Una volta che l'originalità del certificato è stata verificata, il client può controllare se il periodo di validità di quest'ultimo è compatibile con la data corrente. Infine, se il client è a conoscenza dell'indirizzo IP del server, esso può verificare che tale indirizzo sia conforme con quello indicato nel certificato. Di conseguenza, il client inizia lo scambio di messaggi con il server solo se tutte le verifiche precedenti vanno a buon fine.

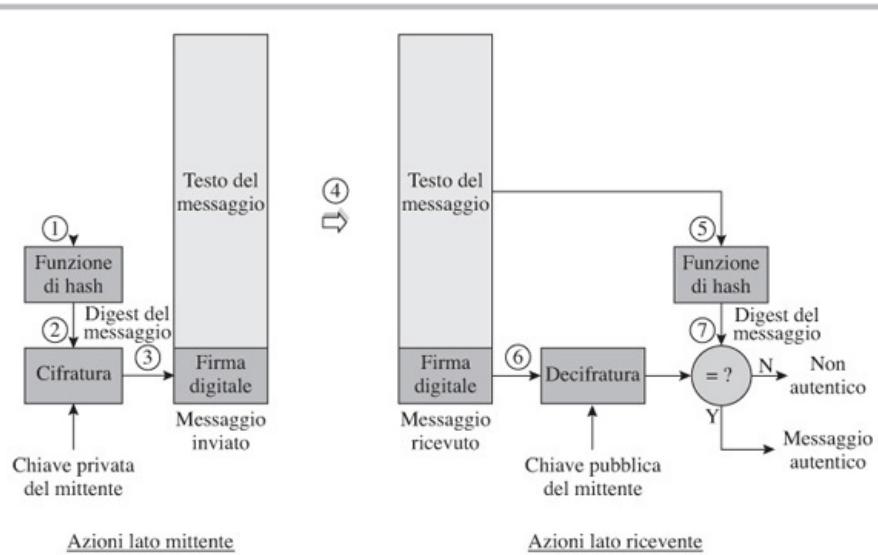
### 20.3.2 Codici per l'autenticazione dei messaggi e firme digitali

I codici per l'autenticazione dei messaggi (*message authentication code*, abbreviato in MAC) vengono usati per verificare l'integrità dei dati. Ogni processo, nel momento in cui crea o trasmette un dato  $d$ , genera un codice di autenticazione per  $d$ ,  $MAC_d$ . Tale codice viene creato dal processo mittente cifrando, con una chiave nota solo a se stesso e al destinatario del dato  $d$ , un digest  $v$ , appositamente generato per  $d$  usando una funzione di hash di tipo one-way. Successivamente, lo stesso processo memorizza o trasmette la coppia  $\langle d, MAC_d \rangle$ . Solo il destinatario di  $d$  potrà così verificare l'integrità del dato.

La firma digitale viene invece usata per verificare l'autenticità di un dato. Un'entità che crea o trasmette un dato  $d$  genera, a partire da quest'ultimo, il valore  $v$  come precedentemente descritto. Successivamente, la stessa entità può generare  $DS_d$ , ovvero la firma digitale di  $d$ , cifrando  $v$  e, opzionalmente, un timestamp, tramite la sua chiave privata. Quindi, essa memorizza o trasmette la coppia  $\langle d, DS_d \rangle$ . Qualunque processo che desidera verificare l'autenticità di  $d$  decifra  $DS_d$  usando la chiave pubblica di chi ha creato  $d$ . Se tale operazione va a buon fine, l'integrità di  $d$  viene validata e viene identificato anche il corrispondente processo che ha generato o inviato  $d$ . Questo tipo di identificazione non è ripudiabile, ovvero l'entità che ha effettivamente creato o inviato il

dato non può successivamente negare di averlo fatto, poiché il dato è stato cifrato usando la sua chiave privata, di cui nessun altro è a conoscenza. La firma digitale può essere usata anche per identificare eventuali modifiche apportate a un dato dopo che quest'ultimo è stato creato o inviato da un processo.

La [Figura 20.5](#) illustra i passi necessari per usare la firma digitale. Il mittente applica una funzione hash di tipo one-way al testo di un messaggio per creare il corrispondente digest. Quindi, esso firma il digest cifrandolo con la sua chiave privata. La firma digitale così ottenuta viene, successivamente, aggiunta alla fine del testo del messaggio prima che quest'ultimo venga inviato. Alla ricezione del messaggio, il ricevente applica, a sua volta, la stessa funzione al testo del messaggio per calcolarne il proprio digest. Quindi, recupera il certificato della chiave pubblica del mittente e usa la chiave per decifrare la firma digitale contenuta nel messaggio ricevuto. In tal modo il ricevente può ottenere il digest calcolato dal mittente. Infine, il ricevente confronta tale digest con il proprio: l'autenticità del messaggio è verificata solo se i due digest corrispondono e il timestamp contenuto nella firma digitale è compatibile con il periodo di validità del certificato della chiave pubblica.



| Passo | Azione                                  | Descrizione                                                                                                                                                                        |
|-------|-----------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.    | Digest del testo del messaggio          | Una funzione di hash di tipo one-way viene applicata al testo del messaggio per produrre il <i>digest</i> del messaggio, che consiste in una stringa di bit di lunghezza standard. |
| 2.    | Creazione della firma digitale          | Il digest del messaggio e un timestamp vengono cifrati usando la chiave privata del mittente. Il risultato della cifratura è la firma digitale.                                    |
| 3.    | Concatenazione della firma              | La firma digitale viene aggiunta alla fine del testo del messaggio.                                                                                                                |
| 4.    | Trasmissione                            | Il messaggio composto dal testo e dalla firma digitale viene trasmesso verso il destinatario.                                                                                      |
| 5.    | Digest del testo del messaggio ricevuto | La stessa funzione di hash usata nel passo 1 viene applicata al testo del messaggio ricevuto per creare il corrispondente digest.                                                  |
| 6.    | Decifratura della firma digitale        | La firma digitale viene estratta dal messaggio ricevuto e decifrata usando la chiave pubblica del mittente.                                                                        |
| 7.    | Verifica dell'autenticità               | Il digest prodotto al passo 5 viene confrontato con il risultato della decifratura effettuata al passo 6. Il messaggio è autentico se il confronto ha esito positivo.              |

**Figura 20.5** Autenticazione di un messaggio tramite firma digitale.

## 20.4 Autenticazione basata su entità esterne

Un sistema aperto utilizza interfacce standard, ben specificate, per comunicare con altri sistemi. In tal modo, qualunque processo in esecuzione su un nodo con un'interfaccia adeguata può richiedere accesso a risorse e servizi forniti da un sistema aperto. Questo aspetto, però, dà origine a un problema di autenticazione – come fa un server a sapere se un processo, che desidera agire come client nei suoi confronti, è stato creato da un utente autorizzato? Una possibile soluzione consiste nel fare in modo che ogni server autentichi tutti gli utenti tramite password. Questo approccio, tuttavia, risulta essere scomodo, poiché ogni server dovrebbe possedere un database di autenticazione a livello di sistema, e ogni utente sarebbe costretto ad autenticarsi parecchie volte durante l'uso del sistema stesso. Un approccio alternativo consiste nell'usare un'entità di autenticazione esterna, la quale, grazie all'uso di un meccanismo sicuro, introduce gli utenti autorizzati nei vari server. In tal modo, non è necessario che i server si preoccupino di autenticare gli utenti.

Nel seguito, verranno presentati due protocolli di autenticazione basati su entità esterne. In particolare, il protocollo Kerberos utilizza un database di autenticazione centralizzato, mentre il protocollo *secure sockets layer* (SSL) effettua l'autenticazione in modo decentralizzato.

### 20.4.1 Kerberos

Kerberos è un'entità di autenticazione esterna sviluppata al MIT, all'interno del progetto Athena, per essere usata in un ambiente basato su sistemi aperti. Esso permette all'utente di provare la sua identità ai server di un sistema aperto senza dover continuamente ripetere l'autenticazione. L'utente viene autenticato tramite password solo nel momento in cui accede al sistema. Quindi, il servizio di autenticazione fornisce a ogni utente autenticato una serie di *ticket*. Ogni ticket rappresenta il privilegio di poter accedere a un server. Pertanto, quando un utente desidera accedere a un servizio, è sufficiente che presenti un ticket al corrispondente server. Di conseguenza, il server fornirà il servizio solo se il ticket è valido. Sia agli utenti che ai server vengono inoltre fornite delle chiavi private. La chiave dell'utente serve per cifrare messaggi inviati da Kerberos ai processi utente, mentre la chiave del server viene usata per cifrare i ticket per il server. Inoltre, vengono usate chiavi di sessione per garantire la sicurezza dei messaggi. Queste ultime vengono generate usando uno schema simile a quello mostrato in [Figura 20.4](#). Per limitare l'esposizione delle chiavi di sessione a eventuali intrusi, esse rimangono valide per un periodo di tempo limitato. A tale scopo vengono usati timestamp. Questi ultimi vengono impiegati anche per sventare eventuali attacchi basati su replicazione dei messaggi. L'uso dei timestamp impone quindi che i nodi del sistema gestiscano qualche meccanismo per la sincronizzazione dei clock.

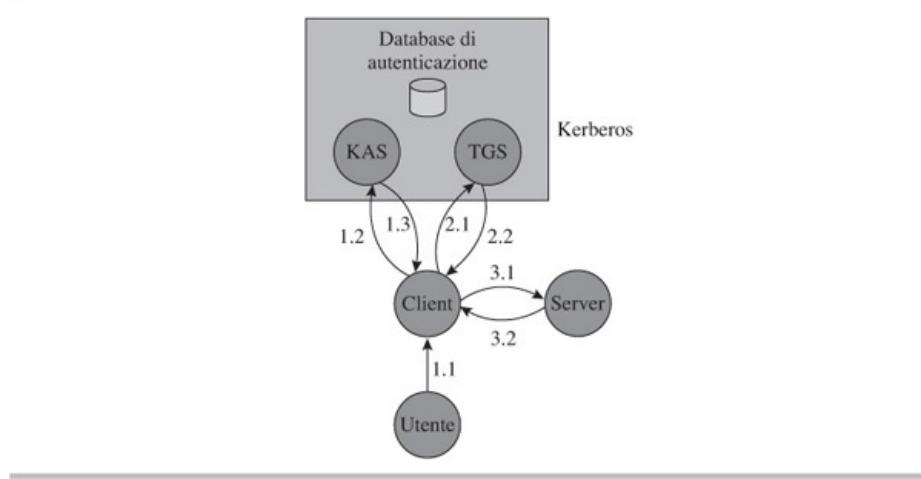
Un *client* è un processo in esecuzione sulla macchina di un utente, che richiede servizi remoti a nome di quest'ultimo. Quando un client  $C$  desidera usare i servizi di un server  $S_j$ ,  $C$  crea una nuova entità di autenticazione, detta *authenticator*, e la presenta, assieme a un ticket, a  $S_j$ . Il ticket viene usato per comunicare in modo sicuro la chiave di sessione al server, mentre l'authenticator viene usato per prevenire attacchi basati su replicazione dei messaggi. Il ticket contiene gli identificatori del client e del server, ovvero  $C$  e  $S_j$ , la chiave assegnata alla sessione di comunicazione tra  $C$  e  $S_j$ , un timestamp per indicare il momento in cui è stato creato il ticket, e il *lifetime* del ticket, cioè il suo periodo di validità. Il ticket è valido quindi solo durante l'intervallo di tempo compreso tra il suo timestamp e il suo lifetime. Tipicamente, tale intervallo è di 8 ore. L'authenticator fornito da  $C$  contiene, invece, l'identificatore di  $C$ , il suo indirizzo, e un timestamp cifrato usando la chiave di sessione.

Il server decifra il ticket usando la propria chiave e ne verifica la validità controllando i corrispondenti timestamp e lifetime. Quindi, il server estrae la chiave di sessione e la usa per decifrare l'authenticator. Dopodiché, verifica il timestamp dell'authenticator per essere certo che la richiesta sia stata effettuata in tempo reale e all'interno del periodo di validità del ticket. Il server soddisfa la richiesta effettuata dal client solo se tutte le verifiche precedenti vanno a buon fine. In tal modo, un eventuale intruso non può replicare né i ticket né l'authenticator per accedere a un servizio senza permesso.

#### Funzionamento di Kerberos

Il sistema Kerberos è composto da due componenti principali: il *Kerberos authentication*

server (KAS) e il *ticket granting server* (TGS). Il KAS autentica gli utenti al momento dell'accesso al sistema, usando un opportuno database, e fornisce a questi ultimi un ticket per il TGS. A sua volta, il TGS permette ai client di ottenere ticket per gli altri servizi del sistema. Gli utenti hanno accesso ai server tramite un protocollo a tre fasi. La [Figura 20.6](#) mostra i vari passi del protocollo.  $n_1$ ,  $n_2$  e  $n_3$  rappresentano valori nonce.



**Figura 20.6** Kerberos.

**1. Autenticazione iniziale:** l'utente viene autenticato al momento dell'accesso al sistema come segue:

- 1.1 Utente → C :  $U, \text{password}$
- 1.2 C → KAS :  $U, \text{TGS}, n_1$
- 1.3 KAS → C :  $E_{V_U}(n_1, \text{SK}_{U,TGS}, T_{TGS})$

Nel passo 1.1, l'utente sottmette il suo identificatore e la sua password al client. Nel passo 1.2, il client inoltra l'identificatore dell'utente al KAS e inserisce il valore  $n_1$  per autenticare il KAS. Tale messaggio funge da richiesta per ottenere un ticket per il TGS. Si noti che la password dell'utente non viene inviata al KAS. In tal modo si evita l'esposizione della password nella rete. L'autenticazione non viene pertanto effettuata dal KAS, bensì dal client C, come verrà descritto in seguito. Nel passo 1.3, il KAS utilizza l'identificatore dell'utente,  $U$ , per recuperare la chiave privata di  $U$ ,  $V_U$ , dal database di autenticazione, e la usa per cifrare la sua risposta per C.  $\text{SK}_{U,TGS}$  è la chiave per la sessione tra l'utente e il TGS, e  $T_{TGS}$  è un ticket per il TGS cifrato con la chiave del TGS.  $T_{TGS}$  viene anche detto *ticket granting ticket* (TGT).

Il client C deve decifrare la risposta ricevuta dal KAS usando la chiave  $V_U$  per recuperare  $\text{SK}_{U,TGS}$  e  $T_{TGS}$ . Questa fase permette di autenticare l'utente nel seguente modo: la chiave privata dell'utente  $V_U$  soddisfa la relazione  $V_U = f(\text{password})$ , dove  $f$  è una funzione di hash di tipo one-way nota a C. Quest'ultimo ottiene  $V_U$  applicando  $f$  alla *password*. A seguito di ciò, il client decifra la risposta ricevuta dal KAS usando proprio  $V_U$ . La decifratura fallisce se la password fornita dall'utente non è valida. In questo caso, il client non riesce a estrarre  $T_{TGS}$  dalla risposta ricevuta dal KAS, e quindi l'utente non può usare né i servizi né le risorse del sistema.

**2. Acquisizione di un ticket per il server:** quando un utente desidera usare un server, il client C può ottenere il corrispondente ticket usando il seguente protocollo:

- 2.1 C → TGS <Server\_id>,  $T_{TGS}$ , AU,  $n_2$
- 2.2 TGS → C  $E_{\text{SK}_{U,Server\_id}}(n_2, T_{Server\_id}, \text{SK}_{U,Server\_id}, <Server\_id>)$

dove  $<Server\_id>$  è il nome del server che C desidera usare, AU è un authenticator,  $\text{SK}_{U,Server\_id}$  è la chiave per la sessione di comunicazione iniziata tra client e server, e  $T_{Server\_id}$  è il ticket per il server, cifrato usando la chiave del server. Prima di rispondere al client, il TGS verifica che il ticket presentato da quest'ultimo sia valido e che la richiesta sia stata effettuata in tempo reale all'interno del periodo di validità del ticket.

3. *Acquisizione del servizio*: quando l'utente  $U$  richiede un servizio,  $C$  genera un authenticator e un valore nonce, e scambia i seguenti messaggi con il server:

3.1  $C \rightarrow \text{Server} : T_{\langle \text{Server\_id} \rangle}, AU, E_{SKU, \langle \text{Server\_id} \rangle} (\langle \text{richiesta di servizio} \rangle, n_3)$

3.2  $\text{Server} \rightarrow C : E_{SKU, \langle \text{Server\_id} \rangle} (n_3)$

Il server fornisce il servizio richiesto solo se il ticket è valido e la richiesta è stata effettuata in tempo reale all'interno del periodo di validità del ticket. Inoltre, il server ritorna al client il valore nonce  $n_3$  in modo che quest'ultimo possa autenticare il server, se lo desidera.

## 20.4.2 Secure sockets layer (SSL)

Il *secure sockets layer* è un protocollo per garantire la sicurezza dei messaggi che fornisce servizi di autenticazione e preserva la privacy durante la comunicazione. Per funzionare correttamente, il secure socket layer si basa su un protocollo di livello trasporto, come per esempio il TCP/IP. Il suo successore, noto come protocollo *transport layer security* (TLS), si basa su SSL 3.0. In questa paragrafo presenteremo caratteristiche comuni sia al protocollo SSL che al protocollo TLS.

Quando un client desidera comunicare con un server, SSL usa un opportuno *protocollo di handshaking* prima che lo scambio dei messaggi abbia inizio. Quest'ultimo impiega RSA per autenticare il server, e facoltativamente anche il client, e per generare una coppia di chiavi di sessione simmetriche da usare durante lo scambio di messaggi tra client e server. Lo scambio di messaggi vero e proprio viene invece effettuato attraverso *SSL record protocol*, che si occupa di cifrare i messaggi e di trasmetterli alla rete. Pertanto, la comunicazione tra client e server risulta essere *affidabile*, grazie all'uso del protocollo di trasporto, *sicura*, grazie all'autenticazione effettuata dal protocollo di handshaking, e *privata*, grazie alla fase di crittografia eseguita dal record protocol. L'autenticazione dei dati viene garantita applicando una *firma digitale* ai messaggi. Se si desidera solo verificare l'integrità dei dati, viene usato un opportuno MAC. I protocolli del livello applicazione, quali HTTP e FTP, possono essere implementati sfruttando le funzionalità fornite da SSL.

Il protocollo di handshaking di SSL effettua le seguenti operazioni:

1. autentica il server;
2. permette a client e server di selezionare l'algoritmo di crittografia da usare durante la paragrafo tra RC2, RC4, DES, triplo DES, e pochi altri, nonché l'algoritmo e la funzione di hash per la firma digitale tra DSA, MD5 e SHA-1;
3. esegue facoltativamente l'autenticazione del client;
4. abilita il client e il server a generare un valore segreto comune da usare per generare le chiavi di sessione.

Semplificando, il protocollo di handshaking di SSL opera come segue. Il client invia al server un messaggio di tipo *client-hello*. Tale messaggio contiene la specifica delle opzioni di crittografia e di compressione, e un numero casuale a 28 byte che chiameremo  $n_{client}$ . Il server risponde inviando un messaggio di tipo *server-hello*, che contiene un numero casuale  $n_{server}$ , e il proprio certificato. Di conseguenza, il client verifica che il certificato del server sia stato generato da un'autorità di certificazione presente in un'apposita lista e controlla l'autenticità del server usando la crittografia a chiave pubblica. Il server, se lo desidera, può chiedere il certificato del client per verificarne l'identità. In seguito a ciò, il client invia un messaggio che contiene una stringa composta da 48 byte (*premaster secret string*) cifrata con la chiave pubblica del server.

Quindi, sia il client che il server generano una stringa di 48 byte (*master secret*) a partire da  $n_{client}$ ,  $n_{server}$  e dalla *premaster secret string* usando una funzione di hash. L'utilizzo di  $n_{client}$  e  $n_{server}$ , che sono valori generati casualmente, garantisce che la stringa *master secret* sia diversa per ogni sessione di comunicazione iniziata dalla stessa coppia client-server. La stringa *master secret* viene usata per ottenere quattro chiavi di sessione simmetriche tramite un algoritmo standard. In particolare, le chiavi  $k_{c \rightarrow s}^{crypt}$  e  $k_{s \rightarrow c}^{crypt}$  sono usate per cifrare e decifrare i messaggi inviati, rispettivamente, dal client al server, e viceversa, mentre le chiavi  $k_{c \rightarrow s}^{mac}$  e  $k_{s \rightarrow c}^{mac}$  servono per generare i codici di autenticazione dei messaggi inviati, rispettivamente, dal client al server, e viceversa. Dopo la generazione delle chiavi, client e server si scambiano reciprocamente un

messaggio di tipo *finished* per indicare la chiusura del protocollo di handshaking.

Lo scambio dei messaggi viene effettuato secondo il record protocol di SSL usando le chiavi di sessione generate durante la fase di handshaking. In particolare, l'invio di un messaggio  $m$  da parte di un client verso un server è costituito dai seguenti passi:

1. il client genera  $MAC_m$ , ovvero il codice di autenticazione del messaggio  $m$ , usando la chiave key  $k_{c \rightarrow s}^{mac}$ ;
2. la coppia  $\langle m, MAC_m \rangle$  viene crittata usando la chiave  $k_{c \rightarrow s}^{crypt}$ , e la stringa risultante viene inviata al server;
3. il server decifra la stringa usando la chiave  $k_{c \rightarrow s}^{crypt}$  per ottenere la coppia  $\langle m, MAC_m \rangle$ .  
Quindi, il server accetta  $m$  se il corrispondente MAC, calcolato usando la chiave  $k_{c \rightarrow s}^{mac}$ , coincide con  $MAC_m$ .

Il protocollo SSL può essere sovvertito da un attacco di tipo *man-in-the-middle*, con il quale un eventuale intruso, dopo aver intercettato i messaggi inviati dal client al server durante la fase di handshaking, finge di essere il server nella successiva fase di scambio dei messaggi. Analogamente, l'intruso può fingere il client per instaurare una connessione SSL con il server. Per contrastare attacchi di questo tipo, client e server devono usare adeguate precauzioni durante la fase di handshaking. In particolare, nel momento in cui un server fornisce il proprio certificato al client, quest'ultimo deve verificare che il distinguished name e l'indirizzo IP citati nel certificato corrispondano a quelli con cui il server stesso sta tentando di instaurare la connessione SSL. Una mancata corrispondenza indica che è in corso un attacco di tipo man-in-the-middle, e quindi il protocollo di handshaking va interrotto.

Al contrario, il server non conosce l'indirizzo IP su cui risiede il client. Pertanto, il server deve usare un approccio diverso per autenticare il client. Quando il server richiede l'autenticazione del client durante la fase di handshaking, quest'ultimo deve inviare il proprio certificato e un valore casuale firmato, noto al server, usando la propria chiave privata. Quindi, il server, ottenuta la chiave pubblica del client dal certificato di quest'ultimo, può validare la corrispondente firma digitale. L'eventuale fallimento di questa fase di validazione indica che è in corso un attacco di tipo man-in-the-middle, e quindi il server interrompe il protocollo di handshaking. Questa fase è analoga a quanto avviene nel protocollo sfida-risposta descritto nel Paragrafo 20.2.2.

## Riepilogo

I messaggi scambiati in un sistema distribuito attraversano collegamenti e nodi che non sono sotto controllo diretto del sistema operativo distribuito. Ciò fornisce l'opportunità a un eventuale intruso di lanciare vari tipi di attacchi al sistema, che possono permettergli di leggere messaggi senza avere la necessaria autorizzazione, manomettere messaggi, fingere di essere un utente registrato, o interferire con l'uso delle risorse e dei servizi offerti dal sistema. In questo capitolo, sono state presentate le tecniche fondamentali che possono essere usate da un sistema operativo distribuito per fronteggiare tali attacchi.

Gli attacchi basati su fuga di notizie o scambio di identità mettono a rischio la *sicurezza dei messaggi*. Essi vengono contrastati tramite la *crittografia*. Nella *crittografia a chiave privata*, i messaggi inviati a un processo devono essere cifrati usando una chiave precedentemente assegnata al processo stesso. Questo meccanismo è comodo per garantire la sicurezza in comunicazioni instaurate tra entità del sistema e processi utente, poiché le prime possono recuperare facilmente la chiave dell'utente. Tuttavia, lo stesso meccanismo non è adatto per essere usato in comunicazioni instaurate tra processi utente. Nella *crittografia a chiave pubblica*, ogni processo  $P_i$  possiede una coppia di chiavi  $(u_i, v_i)$ , dove  $u_i$  rappresenta la chiave pubblica, nota a tutti i processi del sistema, mentre  $v_i$  rappresenta la chiave privata, nota solo a  $P_j$ . Questa coppia di chiavi possiede una proprietà importante: un messaggio cifrato con  $U_j$  può essere decifrato usando  $v_i$ , e viceversa. Per generare tali chiavi viene usato l'algoritmo RSA. La crittografia a chiave pubblica presenta però uno svantaggio: le chiavi usate hanno una dimensione di un ordine di grandezza superiore rispetto alle chiavi usate nella crittografia a chiave privata, quindi le corrispondenti operazioni di crittografia sono costose. A causa di tale svantaggio, ai processi vengono assegnate particolari *chiavi di*

*sessione* da usare durante ogni sessione di comunicazione. Sia le chiavi pubbliche che le chiavi di sessione vengono fornite da un apposito *centro per la distribuzione delle chiavi* (CDC).

Un eventuale intruso può lanciare attacchi basati su *replicazione dei messaggi* per fingere di essere un altro utente. In tale tipo di attacco, l'intruso registra i messaggi inviati o ricevuti da un processo, e li reintroduce nella rete in un momento successivo per ingannare il sistema operativo o il CDC. Per contrastare tale tipologia di attacco si usa il protocollo *sfidarisposta* che permette di autenticare il mittente di un messaggio. Questo protocollo è incluso nel protocollo usato per la comunicazione con il CDC. Lo stesso protocollo può anche essere usato per implementare un meccanismo di mutua autenticazione, che risulta però essere scomodo e costoso da usare. Pertanto, in pratica, si utilizzano tecniche di autenticazione basate su entità esterne, come Kerberos o SSL.

Durante lo scambio di messaggi che avviene tra due processi, è importante verificare che i dati siano autentici, ovvero che siano stati creati o inviati dal processo che dice di averlo fatto e che non siano stati manomessi da altri. Per verificare l'autenticità dei dati viene usata la *firma digitale*. Quest'ultima è costituita da un codice hash generato a partire dai dati creati o inviati dal processo, che viene cifrato usando la chiave privata di quest'ultimo. L'autenticità dei dati viene quindi verificata recuperando, da un'opportuna autorità di certificazione, un *certificato di chiave pubblica* del processo che ha creato o inviato i dati. La firma digitale viene decifrata usando la chiave pubblica contenuta in tale certificato. Se la decifratura va a buon fine, il processo è effettivamente colui che ha creato o inviato i dati. Inoltre, i dati sono autentici se il codice hash da loro generato corrisponde al valore ottenuto decifrando la firma digitale.

## Domande

- 20.1. Indicare se le seguenti affermazioni sono vere o false.
  - a. La replicazione dei messaggi è un attacco attivo alla sicurezza.
  - b. La crittografia previene le intercettazioni, ma non può prevenire la manomissione dei messaggi.
  - c. In un sistema distribuito che usa la crittografia a chiave pubblica, un messaggio inviato da un processo  $P_i$  a un processo  $P_j$  dovrebbe essere cifrato usando la chiave privata di  $P_i$ .
  - d. La crittografia a chiave pubblica genera un sovraccarico maggiore rispetto alla crittografia a chiave privata.
  - e. Le chiavi di sessione vengono usate per limitare l'esposizione delle chiavi utilizzate durante la crittografia a eventuali attacchi da parte di un intruso.
  - f. Il protocollo sfida-risposta può essere usato per prevenire attacchi basati su fuga di identità.
  - g. Il centro per la distribuzione delle chiavi viene usato per distribuire le chiavi private dei processi.
- 20.2. Selezionare l'alternativa appropriata per ciascuna delle seguenti domande.
  - a. Un messaggio contiene l'identificatore del suo processo mittente, l'identificatore del processo ricevente, e la forma cifrata del testo del messaggio. Un eventuale intruso può intercettare il messaggio:
    - i. se il testo del messaggio è stato cifrato usando una chiave di sessione;
    - ii. se il testo del messaggio è stato cifrato usando la chiave pubblica del processo ricevente;
    - iii. se il testo del messaggio è stato cifrato usando la chiave privata del processo mittente;
    - iv. in nessuno dei casi elencati nelle alternative (i)-(iii).
  - b. In un sistema basato su chiave pubblica, il centro per la distribuzione delle chiavi viene usato:
    - i. per garantire la confidenzialità della chiave privata di un processo;
    - ii. per distribuire informazioni sulle chiavi private dei processi;
    - iii. per garantire la confidenzialità della chiave pubblica di un processo;
    - iv. per distribuire informazioni sulle chiavi pubbliche dei processi.
  - c. La firma digitale:

- i. è una stringa che identifica in modo univoco la persona che invia un messaggio;
- ii. è costituita dal testo di un messaggio e dal nome o dall'identificatore del suo mittente;
- iii. è costituita dalla valore cifrato del messaggio e dal nome o dall'identificatore del suo mittente;
- iv. nessuna delle alternative (i)-(iii).

## Problemi

- 20.1. Progettare un cifrario a chiave pubblica per  $n = 77$  usando l'approccio RSA.
- 20.2. Commentare la seguente affermazione: "Non c'è pericolo di scambio di identità se il messaggio inviato da un client a un centro per la distribuzione delle chiavi per richiedere una chiave di sessione viene intercettato da un intruso.-
- 20.3. Si consideri che, in un sistema basato su chiave pubblica, le chiavi di sessione vengano distribuite come segue:
1.  $P_i \rightarrow \text{CDC: } E_{Ukd}(P_i, P_j)$
  2.  $\text{CDC} \rightarrow P_i: E_{Ui}(P_j, SK_{i,j}, E_{Uj}(P_i, SK_{i,j}))$
  3.  $P_i \rightarrow P_j: E_{Uj}(P_i, SK_{i,j})$
  4.  $P_j \rightarrow P_i: E_{SK_{i,j}}(n)$
  5.  $P_i \rightarrow P_j: E_{SK_{i,j}}(n + 1)$
  6.  $P_i \rightarrow P_j: E_{SK_{i,j}}(<\text{messaggio}>)$
- Il precedente protocollo presenta punti critici? In caso di risposta affermativa, suggerire un modo per superare i punti critici identificati.
- 20.4. La manomissione dei messaggi può portare a negazione del servizio?  
(*Suggerimento:* si pensi al concetto di centro per la distribuzione delle chiavi).
- 20.5. Spiegare come sia possibile evitare la trasmissione della password sulla rete quando un utente effettua l'accesso a un sistema basato su Kerberos. Quali azioni dovrebbero essere eseguite quando un utente desidera cambiare la propria password?
- 20.6. Descrivere l'uso del campo lifetime contenuto nei ticket di Kerberos. Quindi, commentare vantaggi e svantaggi derivanti dall'utilizzo di tale campo.
- 20.7. Commentare la validità della seguente affermazione: "In Kerberos, è possibile effettuare un attacco basato su replicazione dei messaggi fino a quando il ticket non scade".
- 20.8. Il protocollo sfida-risposta descritto nel Paragrafo 20.2.2 è costituito dai seguenti passi:

$$\begin{aligned} P_j \rightarrow P_i & \quad E_{SK_{i,j}}(n) \\ P_i \rightarrow P_j & \quad E_{SK_{i,j}}(n + 1) \end{aligned}$$

- Tuttavia, nel passo 3.2 del protocollo di Kerberos, il server restituisce il valore nonce  $n_3$  al client senza effettuare nessuna operazione su di esso. Spiegare perché questo modo di operare è corretto.
- 20.9. Il protocollo sfida-risposta citato nel Problema 20.8 assume che un eventuale intruso non sia in grado di indovinare o di recuperare la chiave  $SK_{i,j}$ . Al contrario, se tale assunzione non è valida, il protocollo fallisce. Un intruso può usare questo fatto per sovvertire la sicurezza della distribuzione delle chiavi di sessione, in un sistema basato su chiavi private, come segue. L'intruso ottiene in qualche modo  $SK_{i,j}$ . Quindi, esso replica il messaggio nel passo 3 del protocollo 20.3, ovvero:

$$\begin{aligned} 3. P_i \rightarrow P_j & : E_{Vj}(P_i, SK_{i,j}), \\ & E_{SK_{i,j}}(<\text{messaggio}>) \end{aligned}$$

In tal modo l'intruso è in grado di instaurare un numero infinito di comunicazioni con  $P_j$ , mentre quest'ultimo crede di interagire con  $P_i$ . Definire un protocollo per prevenire questo rischio. (*Suggerimento:* sarebbe utile se, invece di usare il protocollo sfida-risposta,  $P_j$  provasse a iniziare una nuova sessione con  $P_i$  ottenendo una chiave di sessione dal CDC?)

## Note bibliografiche

Rivest (1978) e Pfeleger e Pfeleger (2003) hanno presentato la teoria su cui si basa la crittografia RSA. Woo e Lam (1992) hanno descritto vari protocolli di autenticazione usati nei sistemi distribuiti. Steiner e altri autori (1988) hanno invece presentato il protocollo Kerberos.

Denning e Denning (1998) hanno proposto un insieme di articoli incentrati sugli attacchi cui è soggetto il ciberspazio e sulla sicurezza di Internet. Khare (1997) ha invece riassunto un insieme di lavori focalizzati sulle problematiche relative all'affidabilità del Web, sulla firma digitale e sulle infrastrutture basate su chiave pubblica. Infine, Cheswick e altri autori (2003) hanno discusso problematiche relative a firewall e autenticazione.

1. Cheswick, W.R., S.M. Bellovin, and A.D. Rubin (2003): *Firewalls and Internet Security*, 2nd ed., Addison-Wesley Professional, Reading, Mass.
2. Denning, D.E., and P.J. Denning (eds.) (1998): *Internet Besieged: Countering Cyberspace Scofflaws*, Addison-Wesley, Reading, Mass.
3. Khare, R. (ed.) (1997): *Web Security: A Matter of Trust*, O'Reilly, Sebastopol, Calif.
4. Lampson, B., M. Abadi, M. Burrows, and E. Wobblor (1992): "Authentication in distributed systems: theory and practice," *ACM Transactions on Computers*, **10**, 4, 265-310.
5. Pfeleger, C.P., and S. Pfeleger (2003): *Security in computing*, Prentice Hall, Englewood Cliffs, N.J.
6. Rivest, R.L., A. Shamir, and L. Adelman (1978): 'On digital signatures and public key cryptosystems,' *Communications of the ACM*, **21**, 2, 120-126.
7. Steiner, J.G., C. Newman and J.l. Schiller (1988): "Kerberos: an authentication service for open network system", *Proceedings of the Winter USENIX conference*.
8. Woo, T.Y. C. and S.S. Lam (1992): "Authentication for distributed systems," *IEEE Computer*.

---

## Indice analitico

---

**Nota: I link di questo indice si riferiscono alla versione stampata. Negli e-reader, per visualizzare il contenuto a cui l'indice si riferisce, potresti aver bisogno di scorrere in avanti di una o più pagine.**

### A

affidabilità, 75-76  
allocazione pool-based, 13  
ambiente di elaborazione, 15, 55, 58  
applicazione real time, 73, 75  
- deadline, 73  
- definizione, 73  
- esempio, 73  
- requisito di risposta, 73  
assenza di interreferenze, 10  
astrazione, 92

### B

batch, 63  
- processing system, 64  
- - command processor, 64  
booting, 85  
bootstrapping, 85  
bus, 35

### C

cache, memoria, 32, 35  
- block, 32  
- line, 32  
- livello, 34  
CC, codice di condizione, 29-31  
cluster, 75  
codice di condizione, CC, 30-31  
command processor, 64  
computer, 28, 41  
comunicazione, 75  
condivisione delle risorse, 75  
consolidazione del workload, 96  
controller DMA, 37  
controllo distribuito, 77  
convenienza per l'utente, 8-10  
CPU, 29, 31  
- codice di condizione (CC), 29  
- flags, 29, 31  
- general purpose register (GPR), 29-30, 43, 46  
- istruzione privilegiata, 30  
- modalità, 30  
- - kernel, 30, 65  
- - privilegiata, 30  
- - utente, 30  
- modello, 30  
- program  
- - counter, PC, 29

- - status word, PSW, 30
- registro
- - base, 35, 65
- - size, 35, 65
- stato, 30-31, 87
- switch, 87
- switching, 27
- crescita incrementale, 75

## **D**

- deadline, 73
- deadlock, 19
- dispatching, 87
- divisione del tempo, 71
- DLL, 108
- DMA, direct memory access, 37, 64, 77
- controller, 37
- driver di dispositivo, 88
- livello utente, 100

## **E**

- elaborazione, velocizzare, 76
- espandibilità, 88
- estendibilità, 17
- evento, 16, 38
- gestione, 38
- exokernel, 102

## **F**

- fault tolerance, 75
- file system, 21
- meccanismo del, 99

## **G**

- gap semantico, definizione, 90
- general purpose register (GPR), 29-30, 43, 46
- gerarchia
  - della memoria, 32, 35
- gestione
  - delle comunicazioni (network), 86
  - degli eventi, 43, 86, 87
  - dei file, 86
  - dei processi, 86
  - del I/O, 86
  - della memoria, 86
- GPR, general purpose register, 29, 30, 43, 46
- graceful degradation, 75
- grado di multiprogrammazione, 66

## **H**

- HAL, hardware abstraction layer, 107
- hypervisor, 96

## **I**

- I/O, 37-38
  - interrupt di, 38
  - meccanismo del, 99
  - modalità

- - ad interrupt, [37](#)
- - DMA, [37](#)
- - programmata, [37](#)
- programmato, [37](#)
- idea astratta di sistema operativo, [6](#)
- idle loop, [96](#)
- indirizzo
  - fisico, [32](#)
  - logico, [32](#)
  - traduzione del, [32](#)
- informazione
  - di protezione della memoria, MPI (memory protection information), [30](#)
  - salvata del PSW, [41](#), [43](#), [44](#)
- insieme delle risorse, [13](#)
- interfaccia
  - a linea di comando, [7](#)
  - GUI, [7](#)
  - utente, [7](#), [86](#)
  - - grafica, [9](#)
- internet, [10](#)
- interprete dei comandi, [7](#)
- interrupt, [16](#), [28](#), [38](#), [41](#)
  - azione, [41](#)
  - classi, [39](#)-[40](#)
  - codice, [40](#), [43](#)
  - I/O, [39](#), [44](#)
  - maschera, [40](#)-[41](#)
  - program, [39](#), [44](#)
  - routine di servizio del, [38](#)
  - servizio, [42](#)-[43](#)
  - software, [39](#), [44](#), [47](#), [86](#)
  - timer, [39](#), [44](#)
  - vettore, [41](#), [94](#)
- istruzione
  - di controllo, [63](#)
  - privilegiata, [30](#)

## **J**

- java
  - bytecode, [97](#)
  - virtual machine, [97](#)

job, [56](#)-[57](#)

## **K**

- kernel, [7](#), [28](#), [97](#), [100](#)
  - exokernel, [102](#)
  - Linux, [104](#), [105](#)
  - microkernel, [100](#), [102](#)
  - non-interruptible, [46](#)
  - preemptible, [46](#)
  - Solaris, [106](#)
  - Windows, [107](#)

## **L**

LAN, local area network, [77](#)

Linux

- kernel, [104](#)-[105](#)
- - [2.6](#), [105](#)

lista di schedulazione, 71-72  
local area network, LAN, 77  
località, 34  
- spaziale, 34  
- temporale, 34  
logical view, 16

## M

macchina  
- di base, 89  
- estesa, 91  
- virtuale, 14, 93  
- - SO, 95  
Mach, 101  
meccanismo, 87  
- del file system, 99  
- dell'I/O, 99  
- della rete, 99  
- di protezione, 99  
- di scheduling, 99  
- per la gestione della memoria, 99  
memoria  
- cache, 32, 35  
- - block, 32  
- - line, 32  
- gestione del meccanismo per la, 99  
- hit ratio, 34  
- virtuale, 14, 20-21, 32, 35  
- - pagina, 35  
memorizzazione del contesto, 86  
memory  
- management unit, MMU, 32  
- map, 63  
microkernel, 100, 102  
mix di programmi, 67  
MMU, memory management unit, 32  
MPI, memory protection information, 30  
modalità privilegiata, 41, 44

## O

OS ospite, 93  
overhead, 8, 17, 27, 74

## P

pagina, 35  
partizione, 13  
PC, program counter, 30, 40, 41  
plug-and-play, 88  
politica, 87  
portabilità, 17, 88, 100  
prelazione, 12, 67  
priorità, definizione, 66  
processo, 17, 57  
- commutazione, 17  
- sincronizzazione, 17  
processore dei comandi, 63  
progettazione di SO a livelli, 90, 93  
program

- contesto, 43
- counter, 30, 40-41
- mix, 66
- status word, PSW, 29, 36
- programma, 56-57
  - CPU-bound, 67
  - I/O-bound, 67
  - kernel, 7
  - mix, 66
  - non kernel, 7
  - prelazione, 59
  - priorità, 59, 66, 69
- protezione, 10-11, 14-15, 22, 86
  - della memoria, 35, 37, 65
  - meccanismo della, 99
- PSW, program status word, 29, 36
  - informazione salvata del, 41, 43, 44

## **Q**

- quasi virtualizzazione, 95

## **R**

- real time
  - applicazione, 73, 75
  - scheduling, 59
  - sistema operativo, 59-60, 73, 75
    - fault tolerance, 75
    - graceful degradation, 75
    - hard, 74
    - soft, 74
- registro
  - base, 35, 65
  - limite
    - inferiore, 35, 37
    - superiore, 35, 37
  - size, 35, 65
  - timer, 70
- remote procedure call, RPC, 77
- rete, meccanismo della, 99
- risorse
  - allocazione, 12, 14
  - pool-based, 13
  - partizionamento, 13
  - condivisione, 75
  - partizione, 13
  - virtuali, 13-14
- round-robin scheduling, 59, 70
- routine di servizio dell'interrupt, 38
- RPC, remote procedure call, 77

## **S**

- salvataggio del contesto, 43
- scambio di messaggi, 18
- scheduler, 12, 87
  - scheduling, 12, 18, 44, 66, 70-71
    - a priorità, 67
    - coda, 70
    - in time sharing, 70-71

- meccanismo di, 99
- nella multiprogrammazione, 66, 69
- real time, 59
- round-robin, 59, 70
- server di stampa, 14
- servizio
  - dell'utente, 63
  - per l'utente, 62-63, 70
  - in multiprogrammazione, 66
  - in time sharing, 70
  - nell'elaborazione batch, 63
  - nella multiprogrammazione, 66
- sicurezza, 11, 14-15, 22, 86
- sincronizzazione di processi, 17
- sistema operativo
  - Amoeba, 101
  - basati su kernel, 97, 100
  - basati su microkernel, 100, 102
  - batch, di elaborazione, 59, 60, 63
  - processore dei comandi, 63
  - tempo di turnaround, 63
  - boot, 11-12
  - classi, 58, 60
  - elaborazione batch, 60
  - multiprogrammazione, 60
  - convenienza per l'utente
  - buon servizio, 60
  - condivisione delle risorse, 60
  - necessità, 60
  - di elaborazione batch, 59, 60, 63
  - processore dei comandi, 63
  - tempo di turnaround, 63
  - distribuito, 59, 75, 77
  - controllo, 60
  - distribuito, 77
  - definizione, 76
  - trasparenza, 60
  - vantaggi, 75
  - e utilizzo effettivo, 8, 10
  - funzionamento, 10
  - funzioni, 86
  - gestione
    - dei programmi, 10, 12
    - delle risorse, 10, 12, 14
  - kernel, basati su, 97, 100
  - microkernel
    - basati su, 100, 102
    - Mach, 101
  - meccanismo, 87
  - multiprogrammato, 59, 60, 64, 69
  - classificazione dei programmi, 66
  - concorrenza CPU-I/O, 66, 69
  - funzioni del kernel, 64, 69
  - grado di multiprogrammazione, 66-67
  - mix di programmi, 66
  - priorità del programma, 59, 66, 69
  - programma
    - CPU-bound, 66
    - I/O-bound, 66

- - - priorità del, 59, 66, 69
- - - scheduling, 66, 69
- - - supporto dell'architettura, 65
- obiettivo, 8, 10
- operazioni, 15
- ospite, 93
- overhead, 8
- politica, 87
- real time, 59-60, 73, 75
- - - fault tolerance, 75
- - - graceful degradation, 75
- - - hard, 74
- - - soft, 74
- struttura, 7
- - a livelli, 90, 93
- - - macchina virtuale SO, 93, 95
- - - monolitica, 89
- - - SO basati su kernel, 97, 100
- - - SO basati su microkernel, 100, 102
- THE, 92-93
- time sharing, 59, 69, 72
- - - gestione della memoria, 71-72
- - - scheduling, 60, 70-71
- - - suddivisione del tempo, 70
- - - time slicing, 60
- - - visione astratta del, 8
- software interrupt, 28, 47
- Solaris, kernel, 106
- sottorichiesta, 57
- spooling, 59
- swapping, 72
  - definizione, 72
  - schema, 73
- system call, 47, 49, 86
  - relativa ai file, 49
  - relativa al programma, 49
  - relativa alle comunicazioni, 49
  - relativa alle informazioni, 49
  - relativa alle risorse, 49

## T

- tempo
  - di risposta, 60, 62
  - - definizione, 62
  - di turnaround, 60, 62
  - - definizione, 62
- thread, 17
- throughput, 60, 62, 66
  - definizione, 62
- time sharing
  - scheduling, 70, 71
  - sistema operativo, 59, 69, 72
  - - gestione della memoria, 71-72
  - - scheduling, 60, 70-71
  - - suddivisione del tempo, 70
  - - - time-slicing, 60
- time-slice, 59, 70
  - definizione, 70
- time-slicing, scheduling round-robin con, 70

traduzione dell'indirizzo, 32

trap, 39

trasparenza, 59, 77

## U

Unix

- architettura di, 103, 104

- shell, 103

user mode, 28

## V

virtual machine, 93

- monitor, 96

virtualizzazione, 95

VM/370, 94

## W

WAN, wide area network, 77

Windows

- architettura, 107-108

- DLL, 108

- executive, 107

- hardware abstraction layer, HAL, 107

- kernel, 107

- sottosistemi dell'ambiente, 107

workload, consolidazione del, 96