



A. Silberschatz - P. B. Galvin - G. Gagne

# SISTEMI OPERATIVI

## concetti ed esempi

ottava edizione

PEARSON

Abraham Silberschatz  
Peter Baer Galvin  
Greg Gagne

# SISTEMI OPERATIVI

## Concetti ed esempi

Ottava edizione

Edizione italiana a cura di Vincenzo Marra  
Università degli Studi di Milano



SISTEMA BIBLIOTECARIO  
DI ATENEO  
*Mantova*

FUM  
469

PEARSON

**village.org**

*il P2P non è un crimine*

SCAMBIO ETICO

# Indice

Prefazione all'ottava edizione italiana  
Prefazione

XIII  
XV

## Parte prima Generalità

### Capitolo 1 Introduzione

1.1	Che cos'è un sistema operativo	3
1.2	Organizzazione di un sistema di calcolo	6
1.3	Architettura degli elaboratori	12
1.4	Struttura del sistema operativo	17
1.5	Attività del sistema operativo	20
1.6	Gestione dei processi	23
1.7	Gestione della memoria	24
1.8	Gestione della memoria di massa	25
1.9	Protezione e sicurezza	29
1.10	Sistemi distribuiti	30
1.11	Sistemi a orientamento specifico	31
1.12	Ambienti d'elaborazione	34
1.13	Sistemi operativi open-source	37
1.14	Sommario	40
	Esercizi pratici	42
	Esercizi	43
1.15	Note bibliografiche	45

### Capitolo 2 Strutture dei sistemi operativi

2.1	Servizi di un sistema operativo	47
2.2	Interfaccia con l'utente del sistema operativo	50
2.3	Chiamate di sistema	53
2.4	Categorie di chiamate di sistema	57
2.5	Programmi di sistema	64
2.6	Progettazione e realizzazione di un sistema operativo	65
2.7	Struttura del sistema operativo	67
2.8	Macchine virtuali	72
2.9	Debugging dei sistemi operativi	79
2.10	Generazione di sistemi operativi	85
2.11	Avvio del sistema	86
2.12	Sommario	87
	Esercizi pratici	88
	Esercizi	89
	Problemi di programmazione	90
	Progetti di programmazione	90
2.13	Note bibliografiche	94

## **Seconda parte Gestione dei processi**

---

### **Capitolo 3 Processi**

3.1	Concetto di processo	97
3.2	Scheduling dei processi	101
3.3	Operazioni sui processi	106
3.4	Comunicazione tra processi	111
3.5	Esempi di sistemi per la IPC	118
3.6	Comunicazione nei sistemi client-server	122
3.7	Sommario	134
	Esercizi pratici	135
	Problemi di programmazione	137
	Progetti di programmazione	141
3.8	Note bibliografiche	144

### **Capitolo 4 Thread**

4.1	Introduzione	145
4.2	Modelli di programmazione multithread	149
4.3	Librerie dei thread	151
4.4	Questioni di programmazione multithread	158
4.5	Esempi di sistemi operativi	163
4.6	Sommario	165
	Esercizi pratici	166
	Esercizi	166
	Progetti di programmazione	169
4.7	Note bibliografiche	173

### **Capitolo 5 Scheduling della CPU**

5.1	Concetti fondamentali	175
5.2	Criteri di scheduling	179
5.3	Algoritmi di scheduling	180
5.4	Scheduling dei thread	191
5.5	Scheduling per sistemi multiprocessore	192
5.6	Esempi di sistemi operativi	198
5.7	Valutazione degli algoritmi	204
5.8	Sommario	209
	Esercizi pratici	210
	Esercizi	211
5.9	Note bibliografiche	214

### **Capitolo 6 Sincronizzazione dei processi**

6.1	Introduzione	215
6.2	Problema della sezione critica	217
6.3	Soluzione di Peterson	219
6.4	Hardware per la sincronizzazione	220
6.5	Semafori	224
6.6	Problemi tipici di sincronizzazione	229
6.7	Monitor	234
6.8	Esempi di sincronizzazione	241
6.9	Transazioni atomiche	246

6.10	Sommario	255
	Esercizi pratici	256
	Esercizi	256
	Problemi di programmazione	261
	Progetti di programmazione	261
6.11	Note bibliografiche	267

## Capitolo 7 Stallo dei processi

7.1	Modello del sistema	269
7.2	Caratterizzazione delle situazioni di stallo	271
7.3	Metodi per la gestione delle situazioni di stallo	275
7.4	Prevenire le situazioni di stallo	276
7.5	Evitare le situazioni di stallo	279
7.6	Rilevamento delle situazioni di stallo	285
7.7	Ripristino da situazioni di stallo	289
7.8	Sommario	290
	Esercizi pratici	291
	Esercizi	292
	Problemi di programmazione	295
7.9	Note bibliografiche	295

## Parte terza Gestione della memoria

### Capitolo 8 Memoria centrale

8.1	Introduzione	299
8.2	Avvicendamento dei processi (swapping)	306
8.3	Allocazione contigua della memoria	308
8.4	Paginazione	312
8.5	Struttura della tabella delle pagine	322
8.6	Segmentazione	327
8.7	Un esempio: Pentium Intel	330
8.8	Sommario	334
	Esercizi pratici	335
	Esercizi	336
	Problemi di programmazione	339
8.9	Note bibliografiche	339

### Capitolo 9 Memoria virtuale

9.1	Introduzione	341
9.2	Paginazione su richiesta	344
9.3	Copiatura su scrittura	351
9.4	Sostituzione delle pagine	353
9.5	Allocazione dei frame	365
9.6	Paginazione degenere (thrashing)	369
9.7	File mappati in memoria	375
9.8	Allocazione di memoria del kernel	380
9.9	Altre considerazioni	383
9.10	Esempi tra i sistemi operativi	389

---

9.11	Sommario	391
	Esercizi pratici	392
	Esercizi	395
	Problemi di programmazione	399
9.12	Note bibliografiche	400

## Parte quarta Gestione della memoria secondaria

---

### Capitolo 10 Interfaccia del file system

10.1	Concetto di file	403
10.2	Metodi d'accesso	411
10.3	Struttura della directory e del disco	414
10.4	Montaggio di un file system	425
10.5	Condivisione di file	427
10.6	Protezione	432
10.7	Sommario	437
	Esercizi pratici	438
	Esercizi	439
10.8	Note bibliografiche	440

### Capitolo 11 Realizzazione del file system

11.1	Struttura del file system	441
11.2	Realizzazione del file system	443
11.3	Realizzazione delle directory	449
11.4	Metodi di allocazione	451
11.5	Gestione dello spazio libero	459
11.6	Efficienza e prestazioni	462
11.7	Ripristino	466
11.8	NFS	470
11.9	Esempio: il file system WAFL	476
11.10	Sommario	479
	Esercizi pratici	480
	Esercizi	480
11.11	Note bibliografiche	482

### Capitolo 12 Memoria secondaria e terziaria

12.1	Struttura dei dispositivi di memorizzazione	483
12.2	Struttura dei dischi	486
12.3	Connessione dei dischi	487
12.4	Scheduling del disco	489
12.5	Gestione dell'unità a disco	494
12.6	Gestione dell'area d'avvicendamento	498
12.7	Strutture RAID	500
12.8	Realizzazione della memoria stabile	510
12.9	Strutture per la memorizzazione terziaria	512
12.10	Sommario	522
	Esercizi pratici	523
	Esercizi	526
12.11	Note bibliografiche	530

**Capitolo 13 Sistemi di I/O**

13.1	Introduzione	531
13.2	Architetture e dispositivi di I/O	532
13.3	Interfaccia di I/O per le applicazioni	541
13.4	Sottosistema per l'I/O del kernel	547
13.5	Trasformazione delle richieste di I/O in operazioni dei dispositivi	554
13.6	STREAMS	557
13.7	Prestazioni	559
13.8	Sommario	562
	Esercizi pratici	562
	Esercizi	563
13.9	Note bibliografiche	564

**Parte quinta Protezione e sicurezza****Capitolo 14 Protezione**

14.1	Scopi della protezione	567
14.2	Principi di protezione	568
14.3	Domini di protezione	569
14.4	Matrice d'accesso	574
14.5	Realizzazione della matrice d'accesso	578
14.6	Controllo dell'accesso	580
14.7	Revoca dei diritti d'accesso	581
14.8	Sistemi basati su abilitazioni	583
14.9	Protezione basata sul linguaggio	585
14.10	Sommario	591
	Esercizi pratici	591
	Esercizi	592
14.11	Note bibliografiche	593

**Capitolo 15 Sicurezza**

15.1	Problema della sicurezza	595
15.2	Minacce per i programmi	599
15.3	Minacce ai sistemi e alle reti	608
15.4	Crittografia come strumento per la sicurezza	613
15.5	Autenticazione degli utenti	624
15.6	Realizzazione di misure di sicurezza	629
15.7	Barriere di sicurezza a protezione di sistemi e reti	636
15.8	Classificazione della sicurezza dei sistemi di calcolo	638
15.9	Un esempio: Windows XP	640
15.10	Sommario	641
	Esercizi	642
15.11	Note bibliografiche	643

## Parte sesta Sistemi distribuiti

### Capitolo 16 Strutture dei sistemi distribuiti

16.1	Introduzione	647
16.2	Tipi di sistemi operativi distribuiti	649
16.3	Tipi di reti	653
16.4	Topologie di rete	656
16.5	Struttura della comunicazione	658
16.6	Protocolli di comunicazione	664
16.7	Robustezza	668
16.8	Problemi di progettazione	670
16.9	Un esempio di comunicazione in rete	672
16.10	Sommario	674
	Esercizi pratici	674
	Esercizi	675
16.11	Note bibliografiche	676

### Capitolo 17 File system distribuiti

17.1	Introduzione	677
17.2	Nominazione e trasparenza	679
17.3	Accesso ai file remoti	682
17.4	Servizio con informazioni di stato e senza informazioni di stato	687
17.5	Replicazione dei file	688
17.6	Un esempio: AFS	690
17.7	Sommario	695
	Esercizi	695
17.8	Note bibliografiche	696

### Capitolo 18 Coordinazione distribuita

18.1	Ordinamento degli eventi	697
18.2	Mutua esclusione	700
18.3	Atomicità	702
18.4	Controllo della concorrenza	706
18.5	Gestione delle situazioni di stallo	710
18.6	Algoritmi di elezione	717
18.7	Raggiungimento di un accordo	719
18.8	Sommario	721
	Esercizi	722
18.9	Note bibliografiche	723

## Parte settima Sistemi a orientamento specifico

### Capitolo 19 Sistemi real-time

19.1	Generalità	727
19.2	Caratteristiche del sistema	728
19.3	Caratteristiche dei kernel real-time	730
19.4	Realizzazione dei sistemi operativi real-time	731
19.5	Scheduling real-time della CPU	735

19.6	VxWorks 5.x	740
19.7	Sommario	743
	Esercizi	744
19.8	Note bibliografiche	744

**Capitolo 20 Sistemi multimediali**

20.1	Che cosa significa multimedia?	745
20.2	Compressione	748
20.3	Requisiti dei kernel multimediali	750
20.4	Scheduling della CPU	752
20.5	Scheduling del disco	753
20.6	Organizzazione della rete	755
20.7	Un esempio: CineBlitz	758
20.8	Sommario	760
	Esercizi	761
20.9	Note bibliografiche	762

**Parte ottava Casi di studio****Capitolo 21 Linux**

21.1	Storia	765
21.2	Principi di progettazione	769
21.3	Moduli del kernel	772
21.4	Gestione dei processi	775
21.5	Scheduling	779
21.6	Gestione della memoria	783
21.7	File system	792
21.8	Input e Output	798
21.9	Comunicazione fra processi	800
21.10	Strutture di rete	801
21.11	Sicurezza	803
21.12	Sommario	806
	Esercizi pratici	806
	Esercizi	807
21.13	Note bibliografiche	808

**Capitolo 22 Windows XP**

22.1	Storia	809
22.2	Principi di progettazione	811
22.3	Componenti del sistema	814
22.4	Sottosistemi d'ambiente	837
22.5	File system	840
22.6	Servizi di rete	848
22.7	Interfaccia per il programmatore	855
22.8	Sommario	862
	Esercizi pratici	863
	Esercizi	863
22.9	Note bibliografiche	864

## **Capitolo 23 Prospettiva storica**

23.1	Migrazione delle caratteristiche	865
23.2	Primi sistemi	866
23.3	Atlas	872
23.4	XDS-940	873
23.5	THE	874
23.6	RC 4000	875
23.7	CTSS	876
23.8	MULTICS	876
23.9	OS/360 di IBM	877
23.10	TOPS-20	878
23.11	CP/M e MS/DOS	879
23.12	Macintosh OS e Windows	880
23.13	Mach	880
23.14	Altri sistemi	882
	Esercizi	882

<b>Bibliografia</b>	883
---------------------	-----

<b>Credits</b>	903
----------------	-----

<b>Indice analitico</b>	905
-------------------------	-----

## Prefazione all'ottava edizione italiana

---

Il testo di Silberschatz, Galvin e Gagne sui moderni sistemi operativi è uno dei manuali universitari di riferimento nell'insegnamento della materia. L'autorevolezza di Silberschatz si coniuga con le esperienze accademiche e professionali di Galvin e Gagne per dare vita a una trattazione aggiornata e approfondita dell'argomento.

L'ottava edizione mantiene inalterato l'impianto originale del testo, rilevatosi nel corso degli anni di grande efficacia didattica. D'altro canto, quasi tutti i capitoli sono stati rivisti e aggiornati per includere le principali novità emerse nel mondo dei sistemi operativi da quattro anni a questa parte. Il materiale sui sistemi operativi open-source, per esempio, è un'importante caratteristica della nuova edizione, come pure il maggiore spazio dedicato alle macchine virtuali. Gli Autori, inoltre, dimostrano ancora una volta particolare attenzione per la platea dei professionisti dell'informatica: i progetti pratici che corredano il testo offrono una concreta possibilità d'applicazione dei concetti teorici trattati.

Insomma, anche questa nuova fatica di Silberschatz e dei suoi coautori riesce nella difficile impresa di illuminare gli aspetti concettuali della materia senza metterne in ombra quelli tecnologici.

*Vincenzo Marra  
Dipartimento di Informatica e Comunicazione  
Università degli Studi di Milano*

## Prefazione

Ci sono i sistemi operativi sono una parte essenziale dei sistemi di calcolo, un corso sui sistemi operativi è una parte essenziale di un percorso di studio d'informatica. Con i calcolatori presenti praticamente in ogni campo, dai giochi ai più complessi e raffinati strumenti di pianificazione impiegati dagli enti governativi e dalle grandi multinazionali, la loro evoluzione ha ormai assunto un ritmo vertiginoso. D'altra parte, i concetti fondamentali sono abbastanza chiari; su di loro si fonda la trattazione svolta in questo libro.

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

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

I concetti fondamentali e gli algoritmi trattati in questo testo spesso si basano su quelli impiegati nei sistemi operativi disponibili sul mercato. Si è in ogni modo cercato di presentare questi concetti e algoritmi in una forma generale, non legata a un particolare sistema operativo. Nel libro sono presenti molti esempi che riguardano i sistemi operativi più diffusi, oltre che i più innovativi, tra cui Solaris di Sun Microsystems, Linux, Microsoft Windows Vista, Windows 2000 e Windows XP, Apple Mac OS X.

Quando nel corso del libro si usa Windows XP come esempio di sistema operativo, si intendono Windows Vista, Windows XP e Windows 2000. Se ci si riferisce a una caratteristica propria di una specifica versione del sistema, la distinzione viene resa esplicita.

## Organizzazione del libro

La struttura di questo testo rispecchia la lunga esperienza degli Autori, in qualità di docenti, nei rispettivi corsi di sistemi operativi; nel redigerlo si è tenuto conto delle valutazioni dei revisori del testo, nonché dei commenti inviati dai lettori delle precedenti edizioni. Inoltre, il testo recepisce nei contenuti quanto suggerito sull'insegnamento dei sistemi operativi da *Computing Curricula 2005*, edito dall'Unità Operativa Comune della IEEE Computing Society e dall'Association for Computing Machinery (ACM).

Come consiglio generale per il lettore, raccomandiamo di seguire i capitoli nell'ordine in cui sono presentati, poiché questo metodo garantisce la massima accuratezza nello studio dei sistemi operativi. È tuttavia possibile selezionare i capitoli (o i paragrafi) secondo una progressione diversa.

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

- **Sistemi distribuiti.** I Capitoli dal 16 al 18 trattano i sistemi distribuiti. Con tale locuzione di solito ci si riferisce a un insieme di unità d'elaborazione che operano senza condivisione di memoria o di clock. Un sistema di questo tipo è in grado di mettere a disposizione dei propri utenti le varie risorse da esso controllate. La possibilità d'accesso a risorse condivise consente un incremento delle prestazioni del sistema, oltre che una maggiore affidabilità e disponibilità di dati e programmi. Attraverso l'uso di file system distribuiti, utenti, server e unità di memorizzazione si possono dislocare tra i vari siti del sistema distribuito. Tale tipo di sistema deve quindi provvedere alla realizzazione di diversi meccanismi per la sincronizzazione e la comunicazione capaci di gestire particolari questioni legate alle situazioni di stallo e alle situazioni critiche che non si incontrano nei sistemi centralizzati.
- **Sistemi a orientamento specifico.** I Capitoli 19 e 20 si occupano dei sistemi mirati a obiettivi specifici, compresi i sistemi real-time e multimediali. Tali sistemi, per la specificità dei propri requisiti, si differenziano da quelli a carattere generale, ovvero dall'argomento che permea il resto dell'opera. Per i sistemi real-time è importante non solo la "correttezza" dei risultati dell'elaborazione, ma anche il rispetto di scadenze temporali determinate a priori. I sistemi multimediali postulano le cosiddette garanzie di qualità-del-servizio, affinché i dati multimediali siano recapitati ai client entro tempi prestabiliti.
- **Casi di studio.** Nei Capitoli dal 21 al 23 del libro e nelle Appendici dalla A alla C (disponibili sul sito) si integrano i concetti esposti descrivendo alcuni sistemi operativi reali, tra i quali i sistemi Linux, Windows XP, FreeBSD, Mach e Windows 2000. I sistemi Linux e FreeBSD sono stati scelti poiché UNIX è un sistema operativo sufficientemente ridotto da poter essere studiato e compreso nei dettagli, pur non essendo un sistema operativo giocattolo. La maggior parte dei suoi algoritmi interni è stata scelta sulla base della semplicità e non per le prestazioni o la raffinatezza. Sia Linux sia FreeBSD sono tra l'altro presenti in tutti i dipartimenti d'informatica delle principali università, perciò un gran numero di studenti può accedervi liberamente. I sistemi Windows XP e Windows 2000 sono stati scelti perché offrono l'opportunità di studiare un sistema operativo moderno progettato e realizzato in modo radicalmente diverso da UNIX. Nel Capitolo 23 sono descritti brevemente alcuni tra i sistemi operativi che hanno maggiormente influenzato l'evoluzione di questo settore.

## Ambienti di programmazione

Per delineare i concetti fondamentali della materia, in questo libro sono esaminati molti sistemi operativi realmente esistenti. Tuttavia si è dedicata particolare attenzione alla famiglia di sistemi operativi di Microsoft (Windows Vista, Windows 2000 e Windows XP) e alle varie versioni di UNIX (tra cui Solaris, BSD e Mac OS X). Si è scelto, altresì, di riservare ampio spazio al sistema operativo Linux, in particolare all'ultima versione del kernel (la 2.6 al momento della stesura del testo).

## Contenuti del libro

Il testo è suddiviso in otto parti principali.

- ◆ **Generalità.** Nei Capitoli 1 e 2 si spiega che cosa sono i sistemi operativi, che cosa fanno e come sono *progettati* e *realizzati*. Questa spiegazione è offerta attraverso un'analisi dell'evoluzione dei concetti dei sistemi operativi, delle comuni caratteristiche e di quei servizi che un sistema operativo deve fornire agli utenti e agli amministratori del sistema. Essendo privi di riferimenti al funzionamento interno, questi capitoli sono consigliabili a chiunque voglia sapere che cos'è un sistema operativo, senza soffermarsi sui dettagli degli algoritmi che ne controllano il funzionamento.
- ◆ **Gestione dei processi.** Nei Capitoli dal 3 al 7 si descrivono i concetti di processo e concorrenza che costituiscono il fondamento dei moderni sistemi operativi. Per processo s'intende l'unità di lavoro di un sistema, che sarà caratterizzato da un insieme di processi eseguiti in modo concorrente, alcuni dei quali sono parte del sistema operativo (quelli incaricati di eseguire il codice di sistema), mentre i rimanenti sono i processi utenti (il cui scopo è, appunto, l'esecuzione del codice utente). In questi capitoli si affrontano i differenti metodi impiegati per lo scheduling e la sincronizzazione dei processi, la comunicazione tra processi e la gestione delle situazioni di stallo. Tra questi argomenti è compresa un'analisi dei thread e un esame dei temi relativi ai sistemi multicore.
- ◆ **Gestione della memoria.** Nei Capitoli 8 e 9 è trattata la gestione della memoria centrale durante l'esecuzione di un processo. Al fine di migliorare sia l'utilizzo della CPU sia la velocità di risposta ai propri utenti, un calcolatore deve essere in grado di mantenere contemporaneamente più processi in memoria. Esistono molti schemi per la gestione della memoria centrale; questi schemi riflettono diverse strategie di gestione della memoria e l'efficacia dei diversi algoritmi dipende dal particolare contesto in cui si applicano.
- ◆ **Gestione della memoria secondaria.** Nei Capitoli dal 10 al 13 si spiega come gli elaboratori moderni gestiscano il file system, la memoria di massa e l'I/O. Il file system fornisce il meccanismo per l'accesso ai dati e ai programmi che risiedono sui dischi, nonché per la loro memorizzazione in linea. Descriviamo gli algoritmi interni fondamentali e le strutture di gestione della memoria e forniamo una solida conoscenza pratica degli algoritmi utilizzati, analizzandone le proprietà, i vantaggi e gli svantaggi. La nostra trattazione comprenderà temi relativi alla memoria secondaria e terziaria. Dal momento che i dispositivi di I/O collegabili a un calcolatore sono del più vario genere, è necessario che il sistema operativo possa contare su funzionalità ampie e diversificate per le applicazioni, cosicché queste possano tenere sotto controllo i dispositivi in ogni loro aspetto. La trattazione dell'I/O mira ad approfondirne progettazione, interfacce, strutture e funzioni interne del sistema. Per molti aspetti, i dispositivi di I/O si guadagnano il primato di componenti più "macchinose" del calcolatore (tra quelle più importanti): analizzeremo, dunque, i problemi che derivano da questo collo di bottiglia nelle prestazioni.
- ◆ **Protezione e sicurezza.** I Capitoli 14 e 15 illustrano gli aspetti principali che riguardano protezione e sicurezza dei sistemi d'elaborazione. Tutti i processi di un sistema operativo devono essere reciprocamente protetti; a tale scopo esistono meccanismi capaci di garantire che solo i processi autorizzati dal sistema operativo possano impiegare le risorse del sistema, come file, segmenti di memoria, cpu e altre risorse. La protezione è il meccanismo attraverso il quale il sistema controlla l'accesso alle risorse da parte di programmi,

A scopo di esempio sono stati inoltre inseriti alcuni programmi in C e in Java, concepiti per i seguenti ambienti di programmazione.

- ◆ **Sistemi Windows.** Il più rilevante ambiente di programmazione per i sistemi Windows è l'API Win32 (*interfaccia per la programmazione di applicazioni*), che dispone di un insieme completo di funzioni per la gestione di processi, thread, memoria e periferiche. Per illustrare l'uso di API Win32 ci si è avvalsi di alcuni programmi in C. I programmi dimostrativi sono stati testati su piattaforme Windows Vista, Windows 2000 e Windows XP.
- ◆ **POSIX.** La sigla POSIX (ossia *interfaccia portabile del sistema operativo*) rappresenta una serie di standard creati essenzialmente per sistemi operativi della famiglia UNIX. Sebbene Windows Vista, Windows XP e Windows 2000 possano eseguire alcuni programmi POSIX, la nostra trattazione è prettamente incentrata sui sistemi UNIX e Linux. I sistemi compatibili con POSIX devono possedere lo standard di base (POSIX.1): è questo il caso di Linux, Solaris e Mac OS X. Esistono poi numerose estensioni degli standard di base; tra queste, l'estensione real-time (POSIX1.b) e quella per i thread (POSIX1.c, meglio nota come Pthreads). Vari programmi, scritti in C, fungono da esempio per chiarire non solo il funzionamento dell'interfaccia API di base, ma anche quello di Pthreads e dello standard per la programmazione real-time. Questi programmi dimostrativi sono stati testati sulle versioni 2.4 e 2.6 di Linux Debian, sul Mac OS X 10.5 e su Solaris 10 con l'ausilio del compilatore `gcc` 3.3 e 4.0.
- ◆ **Java.** Java è un linguaggio di programmazione largamente utilizzato, dotato di una ricca API, oltre che di funzionalità integrate per la creazione e la gestione dei thread. I programmi Java sono eseguibili da qualsiasi sistema operativo, purché vi sia installata una macchina virtuale Java (o JVM). Si illustrano vari concetti in relazione ai sistemi operativi e alle architetture di rete, grazie a programmi testati con la JVM 1.5.

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

In alcuni casi – come per la creazione dei thread – ci serviamo di tutti e tre gli ambienti di programmazione per illustrare un dato concetto, invitando il lettore a confrontare le diverse soluzioni delle tre librerie, in riferimento al medesimo problema. In altri frangenti, soltanto una delle API è chiamata in causa per esemplificare un concetto. Per descrivere la memoria condivisa, per esempio, ricorriamo esclusivamente alla API di POSIX; la programmazione con le socket nell'ambito del protocollo TCP/IP è illustrata tramite la API di Java.

## Ottava edizione

L'ottava edizione di questo testo è stata scritta considerando i numerosi suggerimenti ricevuti riguardo alle precedenti edizioni del testo, insieme con le osservazioni proprie degli Autori, sul mutevole campo dei sistemi operativi e delle reti di calcolatori. È stato riscritto molto nella maggior parte dei capitoli, aggiornando il vecchio materiale ed eliminando quello non più interessante.

È stata fatta una revisione sostanziale e sono state apportate modifiche nell'organizzazione di molti capitoli. In particolare, nel Capitolo 1 è stato dato ampio spazio ai sistemi operativi open-source. Sono stati aggiunti esercizi pratici per gli studenti, le cui soluzioni so-

sono presenti su WileyPLUS, dove si trovano anche nuovi simulatori per esemplificare il funzionamento dei sistemi operativi.

Ecco un breve riepilogo delle principali modifiche apportate a ogni capitolo.

- Il Capitolo 1, **Introduzione**, è stato arricchito includendo CPU multicore, cluster e sistemi operativi open-source.
- Il Capitolo 2, **Strutture dei sistemi operativi**, contiene significativi aggiornamenti sulle macchine virtuali, oltre che sulle CPU multicore e sul debugging dei sistemi operativi.
- Il Capitolo 3, **Processi**, dà ampio spazio alle pipe come forma di comunicazione tra processi.
- Il Capitolo 4, **Thread**, amplia la trattazione sulla programmazione dei sistemi multicore.
- Il Capitolo 5, **Scheduling della CPU**, approfondisce le tematiche riguardanti lo scheduling delle macchine virtuali e le architetture multithread e multicore.
- Il Capitolo 6, **Sincronizzazione dei processi**, comprende nuovi punti riguardanti i lock per la mutua esclusione, l'inversione delle priorità e la memoria transazionale.
- Il Capitolo 8, **Memoria centrale**, include le architetture ad accesso non uniforme (NUMA).
- Il Capitolo 9, **Memoria virtuale**, arricchisce la trattazione di Solaris, includendo la gestione della memoria in Solaris 10.
- Il Capitolo 10, **Interfaccia del file system**, è stato aggiornato sulla scia delle tecnologie più recenti.
- Il Capitolo 11, **Realizzazione del file system**, presenta una descrizione esaustiva del file system ZFS di Sun e sviluppa le tematiche relative ai volumi e alle directory.
- Il Capitolo 12, **Memoria secondaria e terziaria**, contiene novità riguardanti l'ISCSI, i volumi e i pool ZFS.
- Il Capitolo 13, **Sistemi di I/O**, è stato integrato con tematiche riguardanti PCI-X PCI Express e l'HyperTransport.
- Il Capitolo 16, **Strutture dei sistemi distribuiti**, comprende ora anche la trattazione delle reti wireless 802.11.
- Il Capitolo 21, **Linux**, è stato aggiornato all'ultima versione del kernel Linux.
- Il Capitolo 23, **Prospettiva storica**, presenta informazioni sui primi calcolatori e sistemi operativi, come il TOPS-20, il CP/M, MS-DOS, Windows e il Mac OS originale.

## Problemi di programmazione e progetti

Il testo è stato arricchito con diversi progetti ed esercizi di programmazione, che prevedono l'utilizzo delle API POSIX, Win32 e Java. Sono stati aggiunti più di 15 nuovi problemi di programmazione, che pongono in rilievo processi, thread, memoria condivisa, sincronizzazione di processi e questioni attinenti alle reti. Abbiamo inoltre inserito vari progetti di programmazione che presentano maggiore complessità rispetto agli ordinari esercizi di programmazione. Tra questi progetti vi è l'aggiunta di una chiamata di sistema al kernel di Linux, della creazione di pipe sia in Windows sia in Linux e di code di messaggi UNIX, la creazione di

## Ringraziamenti

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

Parti del Capitolo 12 sono state tratte da una relazione di [Hillyer e Silberschatz 1996]; parti del Capitolo 17 da una relazione di [Levy e Silberschatz 1990]; il Capitolo 21 da un manoscritto inedito di Stephen Tweedie; il Capitolo 22 da un manoscritto inedito di Dave Probert, Cliff Martin e Avi Silberschatz; l'Appendice C da un manoscritto inedito dello stesso Cliff Martin, che è stato anche d'aiuto nell'aggiornamento al sistema FreeBSD. Quest'edizione è stata arricchita di molti nuovi esercizi e relative soluzioni a cura di Arvind Krishnamurthy.

Mike Shapiro, Bryan Cantrill e Jim Mauro hanno risposto a diverse domande riguardanti Solaris. Bryan Cantrill della Sun Microsystems ci ha aiutato per la parte su ZFS. Steve Robbins dell'Università del Texas a San Antonio ha disegnato i simulatori presenti su WileyPLUS. Rees Newman del Westminster College ha analizzato i simulatori e valutato quanto fossero adatti a questo libro. Josh Deed e Rob Reynolds hanno contribuito alla parte su Microsoft .NET. John Trono, dell'Università di Saint Michael di Colchester, Vermont, ha contribuito al progetto sulle code di messaggi POSIX.

Marilyn Turnamian ha contribuito alla creazione di immagini e lucidi per le presentazioni. Mark Wogahn si è preoccupato del corretto funzionamento del software utilizzato per la stesura di questo volume (ad esempio Latex, macro, e font).

Il nostro publisher associato, Dan Sayre, ci ha guidato in maniera esperta nella preparazione di questo testo. La sua assistente, Carolyn Weisman, ha curato scrupolosamente molti dettagli. L'editor di produzione, Ken Santor, ci ha dato un valido supporto nella cura dei dettagli di produzione. Lauren Sapira e Cindy Johnson sono state di notevole aiuto nel rendere disponibile il materiale per WileyPLUS.

L'immagine di copertina è di Susan Cyr, e il disegno della copertina è di Howard Grossman. Beverly Peavler si è occupato del copy editing. La correzione delle bozze è a cura di Katrina Avery; l'indicizzazione è stata effettuata da WordCo, Inc.

*Abraham Silberschatz, New Haven, CT, 2008*

*Peter Baer Galvin, Burlington, MA, 2008*

*Greg Gagne, Salt Lake City, UT, 2008*

un'applicazione multithread e la condivisione della memoria quale soluzione al problema dei produttori e consumatori.

L'ottava edizione include anche una serie di simulatori di sistemi operativi progettati da Steven Robbins dell'Università del Texas di San Antonio. I simulatori sono finalizzati a modellare il comportamento di un sistema operativo nel momento in cui esegue diverse attività, quali lo scheduling della CPU e del disco, la creazione di processi e la comunicazione tra processi, la starvation e la traduzione di indirizzi. Tali simulatori sono scritti in Java e funzionano su tutti i computer dotati di Java 1.4.

## Supplementi didattici

I supplementi didattici in lingua inglese, disponibili su <http://pearson.it/silberschatz>, comprendono:

- ◆ le tre appendici dedicate a Unix BSD, Mach e Windows 2000;
- ◆ i simulatori di sistemi operativi;
- ◆ tutti i sorgenti, sia in C sia in Java.

I professori italiani che adottano il testo potranno richiedere, compilando l'apposito modulo on-line che troveranno nello spazio docenti, le slide in italiano e il *Solutions Manual* (in inglese) che comprende anche le soluzioni agli esercizi di programmazione.

## Mailing list

Come mezzo di comunicazione tra i lettori del volume abbiamo scelto il gestore di mailing list di mailman. Se desiderate fruire di questa possibilità, vi preghiamo di consultare il seguente URL, seguendo le istruzioni fornite per iscriversi:

<http://mailman.cs.yale.edu/mailman/listinfo/os-book>

Le mailing list di mailman offrono numerosi vantaggi, tra cui un archivio dei messaggi e la possibilità di scegliere fra varie opzioni di ricezione (per esempio, la cernita dei messaggi o la lettura esclusiva in rete). Per inviare un messaggio alla lista, mandate un messaggio e-mail a:

`os-book@cs.yale.edu`

A seconda del contenuto del messaggio, potremo rispondervi personalmente o inoltrare il messaggio a tutti coloro che fanno parte della lista; la lista è moderata: non riceverete messaggi impropri. Gli studenti che usano questo manuale come libro di testo si astengano dal chiedere le risposte agli esercizi attraverso la lista: non saranno fornite.

## Suggerimenti

In questa nuova edizione si è cercato di eliminare tutti gli errori ma, come accade con i sistemi operativi, qualche oscuro baco probabilmente rimane. È benvenuta qualsiasi segnalazione riguardante errori o omissioni riscontrate nel testo.

Sono altrettanto benvenuti il suggerimento di miglioramenti e i contributi per nuovi esercizi. La corrispondenza deve essere inviata a `os-book-authors@cs.yale.edu`.

**Abraham Silberschatz** è titolare della cattedra Sidney J. Weinberg e direttore del Dipartimento di Informatica dell'Università di Yale. Prima di arrivare a Yale è stato vice presidente del Centro Ricerche Informatiche presso i Laboratori Bell. Prima ancora è stato titolare di una cattedra presso il Dipartimento di Informatica dell'Università del Texas ad Austin.

Il Professor Silberschatz è socio ACM e IEEE. Nel 2002 è stato insignito dell'Education Award Taylor L. Booth della IEEE, nel 1998 ha ricevuto l'Outstanding Educator Award Karl V. Karlstrom dell'ACM e nel 1997 il Contribution Award SIGMOD ACM. In riconoscimento dell'alto livello di innovazione e dell'eccellenza tecnica, è stato premiato dal presidente dei Laboratori Bell per tre differenti progetti: il Progetto QTM (1998), il Progetto DataBlitz (1999) e il Progetto NetInventory (2004).

Gli articoli del Professor Silberschatz sono comparsi su numerose pubblicazioni dell'ACM e dell'IEEE, oltre che in conferenze e altre riviste del settore. Silberschatz è uno degli autori del volume *Database System Concepts*. Ha inoltre scritto articoli come opinionista per alcuni giornali, tra cui il New York Times, il Boston Globe e l'Hartford Courant.

**Peter Baer Galvin** è chief technologist presso la Corporate Technologies ([www.cptech.com](http://www.cptech.com)), società che si occupa di integrazione e rivendita di servizi per l'informatica. In precedenza, è stato amministratore di sistema per il Dipartimento di Informatica della Brown University. Galvin scrive per la rivista *:login:* la rubrica sul sistema Sun. Ha inoltre scritto articoli per Bytes e per altre testate, oltre a rubriche per *SunWorld* e *SysAdmin*. In qualità di consulente e trainer ha tenuto in tutto il mondo conferenze e seminari sulla sicurezza e la gestione dei sistemi.

**Greg Gagne** dirige il Dipartimento di Informatica del Westminster College a Salt Lake City, dove insegna dal 1990. Oltre a essere docente di sistemi operativi, insegna reti, sistemi distribuiti e ingegneria del software. Tiene inoltre corsi di aggiornamento per insegnanti di informatica e professionisti.

## Parte prima

# Generalità

Un *sistema operativo* è un programma che agisce come intermediario tra l'utente e gli elementi fisici di un calcolatore. Lo scopo di un sistema operativo è fornire un ambiente nel quale un utente possa eseguire programmi in modo *conveniente* ed *efficiente*.

Un sistema operativo è il programma che gestisce il *sostrato materiale* di un calcolatore; deve fornire meccanismi idonei che assicurino il corretto funzionamento dell'elaboratore e lo preservino da eventuali interferenze improprie da parte dei programmi utenti.

La struttura interna dei sistemi operativi è soggetta a notevole variabilità ed è adattabile a criteri di organizzazione estremamente differenti. La progettazione di un nuovo sistema operativo è un compito impegnativo che richiede, in via preliminare, una chiara definizione degli obiettivi del sistema. In base a tali obiettivi si selezionano le possibili strategie e si individuano i relativi algoritmi.

I sistemi operativi sono programmi complessi e di vaste dimensioni, e vanno pertanto realizzati un pezzo per volta, per moduli. Ciascuno di loro dovrebbe costituire una parte del sistema chiaramente identificata: è necessario definire scrupolosamente sia le funzionalità sia i dati in ingresso e in uscita.

## Capitolo 1

# Introduzione



### OBIETTIVI

- Panoramica dei più importanti componenti di un sistema operativo.
- Organizzazione degli elementi essenziali di un elaboratore.

Un **sistema operativo** è un insieme di programmi (*software*) che gestisce gli elementi fisici di un calcolatore (*hardware*); fornisce una piattaforma ai programmi applicativi e agisce da intermediario fra l'utente e la struttura fisica del calcolatore. Un aspetto sorprendente dei sistemi operativi è quanto siano diversi i modi in cui eseguono questi compiti: i sistemi operativi per i *mainframe* si progettano innanzitutto per ottimizzare l'utilizzo delle risorse; i sistemi operativi per PC consentono l'esecuzione di un'ampia varietà di programmi, dai giochi ai programmi gestionali; quelli per i sistemi palmari si progettano per fornire un ambiente in cui l'utente possa interagire facilmente col calcolatore per l'esecuzione dei programmi. Alcuni sistemi operativi si progettano per essere *d'uso agevole*, altri per essere *efficienti* e altri ancora per possedere una combinazione di tali qualità.

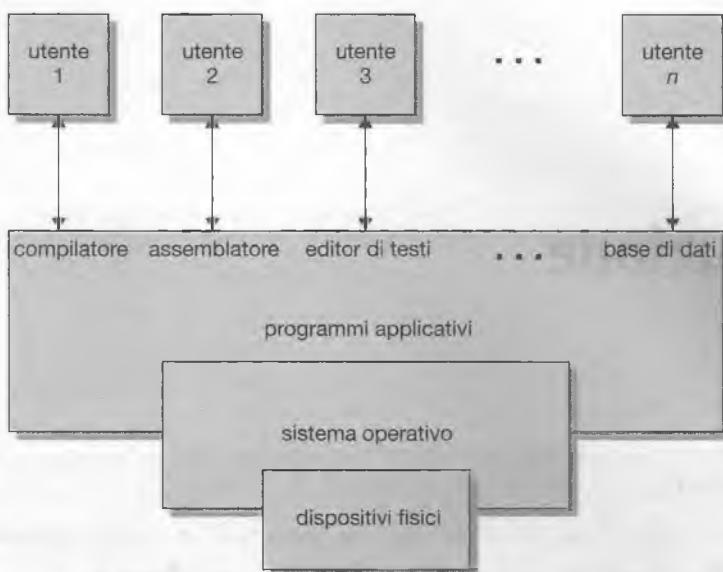
Prima di esplorare i particolari del funzionamento di un calcolatore, contempleremo brevemente la struttura del sistema, a partire dalle funzioni basilari connesse ad avvio, ingresso, uscita (I/O, per Input/Output) e memorizzazione dei dati. Inoltre, delineeremo l'architettura essenziale dell'elaboratore, grazie alla quale si può scrivere un sistema operativo funzionale.

In ragione della sua complessità e ampiezza, un sistema operativo deve essere costruito gradualmente, per parti. Ciascuna di loro dovrebbe rappresentare un'unità ben riconoscibile del sistema, dotata di funzioni, dati in entrata e in uscita accuratamente definiti. Questo capitolo presenta una panoramica generale dei principali componenti di un sistema operativo.

## 1.1 Che cos'è un sistema operativo

La nostra analisi parte dalla considerazione del ruolo del sistema operativo nell'insieme dei sistemi di calcolo. Un sistema di calcolo si può suddividere in quattro componenti: *dispositivi fisici*, *sistema operativo*, *programmi applicativi* e *utenti* (Figura 1.1).

I **dispositivi fisici** composti dall'unità centrale d'elaborazione, cioè la CPU (*central processing unit*), dalla **memoria** e dai dispositivi d'immissione ed emissione dei dati, cioè l'I/O (*input/output*), forniscono al sistema le risorse di calcolo fondamentali. I **programmi applicativi** sono programmi che eseguono compiti specifici per l'utente.



**Figura 1.1** Componenti di un sistema di calcolo.

cativi (editor di testo, fogli di calcolo, compilatori e programmi di consultazione del Web) definiscono il modo in cui si usano queste risorse per la risoluzione dei problemi computazionali degli utenti. Il sistema operativo controlla e coordina l'uso dei dispositivi da parte dei programmi applicativi per gli utenti.

Un sistema di calcolo si può anche considerare come l'insieme dei suoi elementi fisici, programmi e dati. Il sistema operativo offre gli strumenti per impiegare in modo corretto queste risorse; non compie operazioni di per sé utili, ma definisce semplicemente un *ambiente* nel quale altri programmi possono lavorare in modo utile.

Per un maggior approfondimento del ruolo del sistema operativo, lo esploriamo da due punti di vista: quello degli utenti e quello del sistema.

### 1.1.1 Punto di vista dell'utente

La percezione di un calcolatore da parte di un utente dipende principalmente dall'interfaccia impiegata. La maggior parte degli utenti usa PC, composti da schermo, tastiera, mouse e un'unità di sistema. I PC sono progettati per un singolo utente, che impiega le risorse in modo esclusivo, con lo scopo di massimizzare la quantità di lavoro che l'utente può svolgere. In questo caso il sistema operativo si progetta considerando principalmente la **facilità d'uso**, con qualche attenzione alle prestazioni, ma nessuna all'**utilizzo delle risorse**, a come cioè sono condivise le risorse hardware e software. Le prestazioni sono naturalmente importanti; questi sistemi, tuttavia, si focalizzano sull'esperienza del singolo utente più che sull'utilizzo delle risorse in generale.

Alcuni utenti usano terminali connessi a **mainframe** o **minicalcolatori** condividendo le risorse con altri utenti anche loro connessi tramite terminali. In questo caso il sistema operativo si progetta per massimizzare l'utilizzo delle risorse, garantendo che tutto il tempo disponibile di CPU, la memoria e le periferiche di I/O siano impiegati in modo equo ed efficiente.

In altri casi ci sono utenti che usano **stazioni di lavoro**, connesse a reti di altre stazioni di lavoro e a **server**; dispongono di risorse loro riservate: altre devono condividerle, come la rete e i server, per l'accesso ai file e per i servizi di stampa e di elaborazione. Tutto ciò richie-

de che il sistema operativo sia progettato ponderando l'adeguatezza all'uso individuale e l'utilizzo ottimale delle risorse.

Negli ultimi tempi si sono diffusi calcolatori palmari dei tipi più svariati; si tratta prevalentemente di unità a sé stanti, usate singolarmente da ciascun utente. Alcuni sono collegati a reti di calcolatori, direttamente tramite cavi o più spesso tramite dispositivi d'accesso senza fili. A causa della loro scarsa autonomia e interfaccia limitata, svolgono un numero piuttosto ridotto di operazioni remote. I loro sistemi operativi sono progettati principalmente per facilitare l'uso individuale, ma anche per ridurre il consumo delle batterie.

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

### 1.1.2 Punto di vista del sistema

Dal punto di vista del calcolatore, il sistema operativo è il programma più strettamente correlato ai suoi elementi fisici. In tale contesto è possibile considerare un sistema operativo come un **assegnatore di risorse**. Un sistema di calcolo dispone di risorse (fisiche e programmi) utili per la risoluzione di un problema: tempo di CPU, spazio di memoria, spazio per la registrazione di file, dispositivi di I/O e così via. Il sistema operativo agisce come gestore di tali risorse. Di fronte a numerose ed eventualmente conflittuali richieste di risorse, il sistema operativo deve decidere come assegnarle agli specifici programmi e utenti affinché il sistema di calcolo operi in modo equo ed efficiente. Come abbiamo visto, l'assegnazione delle risorse è importante soprattutto nel caso in cui molti utenti accedono agli stessi mainframe o minicalcolatori.

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

### 1.1.3 Definizione di sistema operativo

Abbiamo osservato il ruolo del sistema operativo sia dal punto di vista dell'utente sia da quello del sistema. Tuttavia, in generale, non si dispone di una definizione completa ed esauriente di sistema operativo. I sistemi operativi esistono poiché rappresentano una soluzione ragionevole al problema di realizzare un sistema di calcolo che si possa impiegare in modo utile, per eseguire i programmi utenti e facilitare la soluzione dei problemi degli utenti, ed è proprio per questo scopo che sono stati costruiti i calcolatori; ma, poiché un calcolatore di per sé non è molto facile da utilizzare, sono stati sviluppati i programmi applicativi. Questi programmi sono diversi tra loro, ma richiedono alcune funzioni comuni, per esempio il controllo dei dispositivi di I/O. Tali funzioni comuni, di controllo e assegnazione delle risorse, sono state racchiuse in un unico insieme coerente di programmi: il sistema operativo.

Non si dispone nemmeno di una definizione universalmente accettata di che cosa faccia parte di un sistema operativo. Un punto di vista semplice è considerare che esso comprenda tutto quello che il rivenditore fornisce quando gli si richiede "il sistema operativo". Tuttavia, i requisiti della memoria e della capacità di registrazione in dischi e nastri, e le funzioni richieste variano molto da sistema a sistema. Alcuni sistemi usano meno di un megabyte di memoria e

non possiedono neppure un *editor* a pieno schermo, mentre altri richiedono centinaia di me-gabyte di memoria e sono interamente basati su interfacce grafiche a finestre. La capacità di registrazione di un sistema si misura in gigabyte (un kilobyte o KB è pari a 1024 byte, un me-gabyte o MB è  $1024^2$  byte e un gigabyte o GB è  $1024^3$  byte; spesso i costruttori di calcolatori approssimano questi valori esprimendosi in termini di un milione di byte per un megabyte e di un miliardo di byte per un gigabyte). Una definizione più comune è quella secondo cui il sistema operativo è il solo programma che funziona sempre nel calcolatore, generalmente chiamato *kernel* (*nucleo*). (Oltre al kernel vi sono due tipi di programmi: i programmi di sistema, associati al sistema operativo, ma che non fanno parte del kernel, e i programmi applicativi, che includono tutti i programmi non correlati al funzionamento del sistema.)

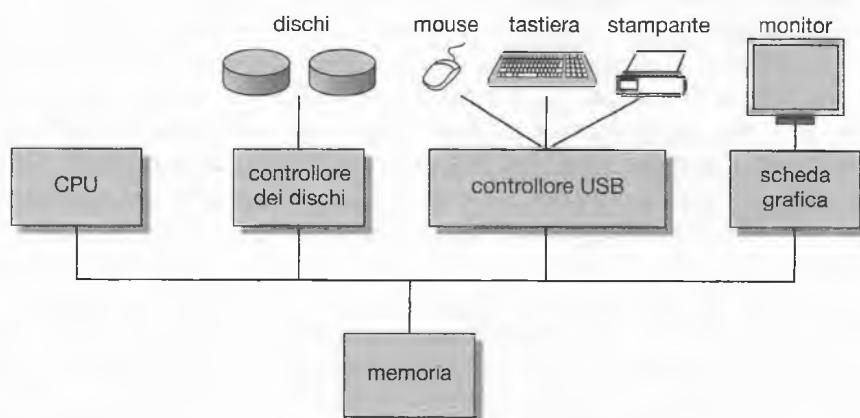
La questione riguardante i componenti di un sistema operativo ha assunto una certa rilevanza anche in seguito all'azione legale promossa nel 1998 dal Dipartimento della giustizia degli Stati Uniti contro la Microsoft, accusata di includere troppe funzioni nel sistema operativo (come nel caso di programmi di consultazione del Web) e quindi di concorrenza sleale nei confronti dei produttori e rivenditori di applicazioni. Di conseguenza fu dichiarata colpevole di sfruttamento di monopolio a danno della concorrenza.

## 1.2 Organizzazione di un sistema di calcolo

Prima di addentrarci nello studio dei dettagli di un sistema operativo è necessaria una conoscenza generale della struttura di un calcolatore. Lo scopo del presente paragrafo è proprio consolidare le basi di questa conoscenza, proponendo una trattazione introduttiva dei diversi elementi di questa struttura. La trattazione riguarda principalmente le architetture dei calcolatori, quindi chi già conosce questo argomento può saltare il paragrafo o limitarsi a una semplice occhiata.

### 1.2.1 Funzionamento di un sistema di calcolo

Un moderno calcolatore d'uso generale è composto da una CPU e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (*bus*) che permette l'accesso alla memoria condivisa dal sistema (Figura 1.2). Ciascuno di questi con-



**Figura 1.2** Moderno sistema di calcolo.

## LO STUDIO DEI SISTEMI OPERATIVI

Lo studio dei sistemi operativi non è mai stato così interessante come al giorno d'oggi, e non è mai stato così facilitato. Con la diffusione del movimento open-source molti sistemi operativi, tra cui Linux, UNIX BSD, Solaris e una parte di Mac OS X sono diventati disponibili sia in formato sorgente sia in formato binario (eseguibile). Disponendo del codice sorgente è possibile studiare i sistemi operativi partendo dal loro interno e rispondere a domande che richiedevano in precedenza lo studio della documentazione o l'osservazione del comportamento di un sistema operativo.

Inoltre, l'incremento della virtualizzazione quale funzione principale (e spesso gratuita) del computer permette di far funzionare più sistemi operativi su un unico sistema (*core system*). Ad esempio, VMware (<http://www.vmware.com>) fornisce un programma gratuito sul quale possono girare centinaia di applicazioni virtuali gratuite. Grazie a questo metodo, gli studenti possono sperimentare senza alcun costo centinaia di sistemi operativi diversi all'interno di un sistema operativo dato.

Anche quei sistemi operativi non più attuali dal punto di vista commerciale sono spesso disponibili in versione open-source, il che permette di studiarne il funzionamento su sistemi con CPU lenta e poche risorse di memoria. All'indirizzo [http://dmoz.org/Computers/Software/Operating\\_Systems/Open\\_Source/](http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/) è reperibile un'ampia lista, seppur non esaustiva, di progetti di sistemi operativi open-source. In alcuni casi sono disponibili anche simulatori di hardware specifico che consentono al sistema operativo di funzionare sull'hardware nativo, anche quando si stanno utilizzando un computer e un sistema operativo moderni. Ad esempio, un simulatore DECSYSTEM-20 attivo su Mac OS X può avviare TOPS-20, caricare nastri e modificare e compilare un nuovo kernel TOPS-20. Se interessato, lo studente può cercare in Internet i manuali e i documenti originali che descrivono il sistema operativo.

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

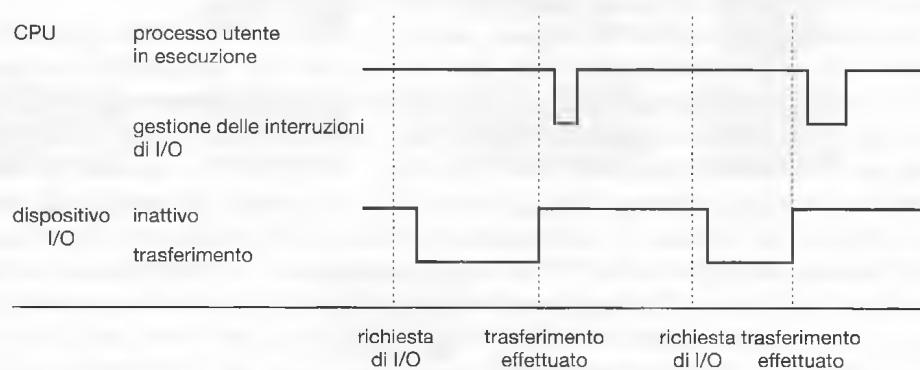
trollori si occupa di un particolare tipo di dispositivo fisico (per esempio, unità a disco, dispositivi audio e unità video). La CPU e questi controllori possono operare in modo concorrente, contendendosi i cicli d'accesso alla memoria. La sincronizzazione degli accessi alla memoria è garantita dalla presenza di un controllore di memoria.

L'avviamento del sistema, conseguente all'accensione fisica di un calcolatore, così come il riavvio di un calcolatore già acceso, richiede la presenza di uno specifico programma iniziale, di solito non troppo complesso, detto **programma d'avviamento** (*bootstrap program*), in genere contenuto in tipi di memoria noti con il termine generale di *firmware*, il cui supporto fisico è parte integrante della macchina. Esempi di firmware sono le memorie a sola lettura (*read only memory*, ROM), e le memorie programmabili cancellabili elettricamente (EEPROM). La funzione di tale programma consiste nell'inizializzare i diversi componenti del sistema, dai registri della CPU ai controllori dei diversi dispositivi, fino al contenuto della memoria centrale. Il programma d'avviamento deve caricare nella memoria il sistema operativo e avviare l'esecuzione, perciò individua e carica nella memoria il kernel del sistema operativo; il sistema operativo avvia quindi l'esecuzione del primo processo d'elaborazione, per esempio `init`, e attende che si verifichi qualche evento.

Un evento è di solito segnalato da un'interruzione dell'attuale sequenza d'esecuzione della CPU, che può essere causata da un dispositivo fisico o da un programma. Nel primo caso si parla di **segnalet d'interruzione** o, più brevemente, **interruzione** (*interrupt*); si tratta di segnali che i controllori dei dispositivi e altri elementi dell'architettura possono inviare alla CPU, di solito attraverso il bus di sistema. Nel secondo caso si parla di **segnalet di eccezione** o, più brevemente, **eccezione** (*exception* o *trap*), che può essere causata da un programma in esecuzione a seguito di un evento eccezionale, riconosciuto tramite l'architettura della CPU (per esempio un errore: una divisione per zero o un accesso alla memoria non valido); oppure a seguito di una richiesta specifica effettuata da un programma utente per ottenere l'esecuzione di un servizio del sistema operativo, attraverso una speciale istruzione detta **chiamata di sistema** (*system call*) o **chiamata supervisore** (*supervisor call*, SVC).

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

I segnali d'interruzione sono un elemento importante nell'architettura di un calcolatore, e ciascun tipo di calcolatore ha il proprio meccanismo delle interruzioni; ciò nonostante molte funzioni sono comuni. Un segnale d'interruzione deve causare il trasferimento del controllo all'appropriata procedura di servizio dell'evento a esso associato. Il modo più semplice per gestire quest'operazione è quello di impiegare una procedura generale che esamina le informazioni presenti nel segnale d'interruzione, e invoca la procedura di gestione dello specifico segnale d'interruzione. D'altra parte, la gestione di un'interruzione deve essere molto rapida perciò, considerando che il numero dei possibili segnali d'interruzione è predefinito, si può usare una tabella di puntatori alle specifiche procedure. In questo modo, l'attivazione delle procedure di servizio delle interruzioni avviene in modo indiretto attraverso questa tabella, senza procedure intermedie. In genere la tabella di puntatori contenente gli indirizzi delle procedure di servizio delle interruzioni è mantenuta nella memoria bassa (per esempio, le prime 100 locazioni). L'accesso a questa sequenza d'indirizzi, detta **vettore delle interruzioni**, avviene per mezzo di un indice, codificato nello stesso segnale d'interruzione, allo scopo di fornire l'indirizzo della procedura di servizio relativa all'evento segnalato dall'interruzione. Sistemi operativi radicalmente differenti, come Windows e UNIX, usano lo stesso meccanismo di gestione delle interruzioni.



**Figura 1.3** Diagramma temporale delle interruzioni per un singolo processo che emette dati.

L'architettura di gestione delle interruzioni deve anche salvare l'indirizzo dell'istruzione interrotta. Molti sistemi di vecchia concezione si limitavano a memorizzare l'indirizzo dell'istruzione interrotta in una locazione fissa o indicizzata dal numero di dispositivo; architetture più recenti memorizzano l'indirizzo di ritorno nella pila (*stack*) di sistema. Se la procedura di gestione dell'interruzione richiede la modifica dello stato della CPU, per esempio modificando il contenuto di qualche registro, deve salvare esplicitamente lo stato corrente per poterlo ripristinare prima di restituire il controllo. Terminato il servizio dell'interruzione, l'indirizzo di ritorno precedentemente salvato viene carica nel **contatore di programma** (*program counter*), che contiene l'indirizzo della prossima istruzione da eseguire, consentendo la ripresa della computazione interrotta come se nulla fosse accaduto.

### 1.2.2 Struttura della memoria

La CPU può caricare istruzioni esclusivamente dalla memoria, quindi tutti i programmi da eseguire devono esservi caricati. I computer general-purpose eseguono la maggior parte dei programmi da una memoria riscrivibile, la memoria principale, chiamata anche **memoria ad accesso diretto** (*random access memory*, RAM). La memoria principale è realizzata solitamente con una tecnologia basata su semiconduttori chiamata **memoria dinamica ad accesso diretto** (*dynamic random access memory*, DRAM). I computer utilizzano anche altri tipi di memoria. Dal momento che la memoria di sola lettura (ROM) non può essere modificata, solo i programmi statici vi sono salvati. L'immutabilità della ROM è utile nelle cartucce per i videogiochi. Le EEPROM non possono essere cambiate di frequente, e per questo contengono per lo più programmi statici. Sulle EEPROM degli smartphone, ad esempio, sono memorizzati i programmi installati inizialmente dalla fabbrica.

Tutte le tipologie di memoria forniscono un vettore di parole. Ciascuna parola possiede un proprio indirizzo. L'interazione avviene per mezzo di una sequenza di istruzioni **load** e **store** opportunamente indirizzate. L'istruzione **load** trasferisce il contenuto di una parola della memoria centrale in uno dei registri interni della CPU, mentre **store** copia il contenuto di uno di questi registri nella locazione di memoria specificata. Oltre agli accessi dovuti alle operazioni **load** e **store**, che si richiedono in modo esplicito, la CPU preleva automaticamente dalla memoria centrale le istruzioni da eseguire.

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

In teoria, si vorrebbe che sia i programmi sia i dati da essi trattati potessero risiedere in modo permanente nella memoria centrale. Questo non è possibile per i seguenti due motivi:

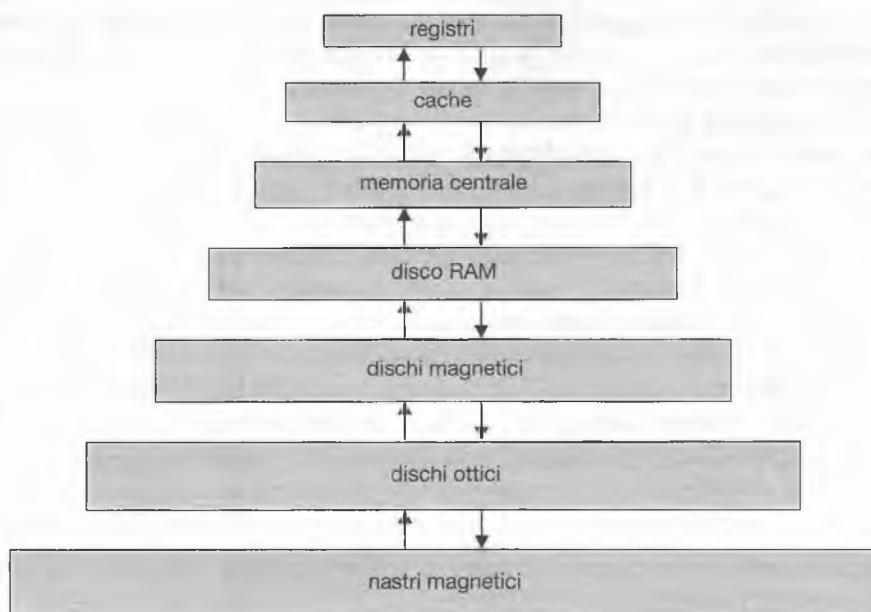
1. la capacità della memoria centrale non è di solito sufficiente a contenere in modo permanente tutti i programmi e i dati richiesti;
2. la memoria centrale è un dispositivo di memorizzazione *volatile*, sicché perde il proprio contenuto quando si spegne il sistema o si ha un'interruzione dell'alimentazione elettrica.

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

Il dispositivo più comunemente impiegato a questo scopo è l'unità a **disco magnetico**, adoperato per la memorizzazione sia di programmi sia di dati. La maggior parte dei programmi (elaboratori di testi, fogli di calcolo, programmi di consultazione del Web, compilatori, e così via) è mantenuta in un disco sino al momento del caricamento nella memoria, e fa uso di un disco come sorgente e destinazione delle informazioni elaborate. Come si spiega nel Capitolo 12, una corretta gestione delle unità a disco è di fondamentale importanza per un sistema di calcolo.

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

L'ampio ventaglio dei sistemi di memorizzazione disponibili in un elaboratore può essere ordinato secondo una scala gerarchica (Figura 1.4), sulla base della velocità e del costo. I gradini più alti ospitano i dispositivi più veloci, ma anche più dispendiosi. Andando verso il basso il costo per bit generalmente diminuisce, mentre il tempo di accesso tende ad aumentare. Si tratta di un compromesso ragionevole: se un certo sistema di memorizzazione, a parità di condizioni, fosse più veloce e nel contempo meno costoso rispetto ad altri, verrebbe meno qualunque motivo per usare la soluzione più lenta e costosa. In effetti, molti dispositivi di memorizzazione della prima ora, quali i sistemi a nastro perforato e le memorie a nuclei magnetici, sono ormai relegati nei musei, essendo stati soppiantati dai nastri magnetici e dalla me-



**Figura 1.4** Gerarchia dei dispositivi di memoria.

**moria a semiconduttori**, più rapidi ed economici. I dispositivi nei quattro livelli superiori della Figura 1.4 possono essere materialmente costruiti con la memoria a semiconduttori.

Oltre a caratterizzarsi per velocità e costo, i sistemi di memorizzazione si suddividono in volatili e non volatili. Come si è accennato, la **memoria volatile** comporta la perdita dei dati nel caso di interruzione dell'alimentazione. Qualora non si disponga di sistemi di salvataggio automatico dei dati (*sistemi di backup*) basati su dispendiose batterie o generatori elettrici, affinché siano preservati è necessario salvare i medesimi su **memoria non volatile**. Nella gerarchia della Figura 1.4, i dispositivi di memorizzazione al di sopra del disco RAM sono volatili, mentre gli ultimi tre in basso sono non volatili. Un **disco** RAM si può progettare per essere volatile o non volatile. Durante il normale funzionamento, il disco RAM memorizza i dati in un capiente vettore DRAM, che è volatile. Tuttavia, molti dischi RAM contengono, in posizione nascosta, un disco rigido magnetico e un alimentatore di riserva per la batteria. Nel caso di un'interruzione della corrente elettrica, il controllore del disco RAM copia i dati dalla RAM sul disco magnetico, per poi eseguire l'operazione inversa al ripristino della corrente elettrica. Una variante del disco RAM è rappresentata dalla memoria flash, impiegata in vari prodotti, dalle macchine fotografiche agli **assistenti digitali personali** (PDA) e ai robot; la sua diffusione come memoria rimovibile nei computer di uso generale è in costante aumento. La memoria flash è più lenta della DRAM, ma non necessita di alimentazione esterna. Un'altra possibilità per la memorizzazione non volatile è la NVRAM, vale a dire la DRAM provvista di batterie di scorta. Questo tipo di memoria arriva a eguagliare la DRAM in velocità, ma la sua non volatilità ha durata limitata.

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

### 1.2.3 Struttura di I/O

La memoria è solo uno dei numerosi dispositivi di I/O di un elaboratore. Una percentuale cospicua del codice di un sistema operativo è dedicata alla gestione dell'I/O; ciò è dovuto in parte alla sua importanza per il progetto di un sistema affidabile ed efficiente e in parte alla natura variabile dei dispositivi preposti all'ingresso e all'uscita dei dati. Presentiamo adesso una panoramica generale dell'I/O.

Un calcolatore d'uso generale è composto da una CPU e da un insieme di controllori di dispositivi connessi mediante un bus comune. Ciascun controllore deve occuparsi di un particolare tipo di dispositivo e, secondo la sua natura, può gestire uno o più dispositivi a esso connessi. Un controllore SCSI (*small computer-systems interface*), per esempio, è capace di controllare sette o più dispositivi. Un controllore di dispositivo dispone di una propria memoria interna, detta memoria di transito (*buffer*), e di un insieme di registri specializzati. Il controllore è responsabile del trasferimento dei dati tra i dispositivi periferici a esso connessi e la propria memoria di transito. I sistemi operativi in genere possiedono, per ogni controllore del dispositivo, un **driver del dispositivo** che si coordina con il controllore e funge da interfaccia uniforme con il resto del sistema.

Per avviare un'operazione di I/O, il driver del dispositivo carica i registri interessati all'interno del controllore, il quale, dal canto suo, esamina i contenuti di questi registri per scegliere l'azione da intraprendere (per esempio "leggi un carattere dalla tastiera"). Il controllore comincia a trasferire i dati dal dispositivo al proprio buffer locale. A trasferimento

completato, il controllore informa il driver, tramite un'interruzione, di avere terminato l'operazione. Il driver passa quindi il controllo al sistema operativo, restituendo i dati (o un puntatore a essi) se l'operazione è di lettura; per altre operazioni, il driver restituisce delle informazioni di stato.

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

Alcuni sistemi all'avanguardia hanno abbandonato la configurazione basata sul bus per adottare un'architettura incentrata sugli switch, in cui più dispositivi fisici possono integrare con varie parti del sistema concorrentemente, piuttosto che contendersi un unico bus condiviso. In tali circostanze, l'accesso diretto alla memoria risulta ancora più efficace. La Figura 1.5 mostra l'interazione di tutti i componenti di un elaboratore.

### 1.3 Architettura degli elaboratori

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

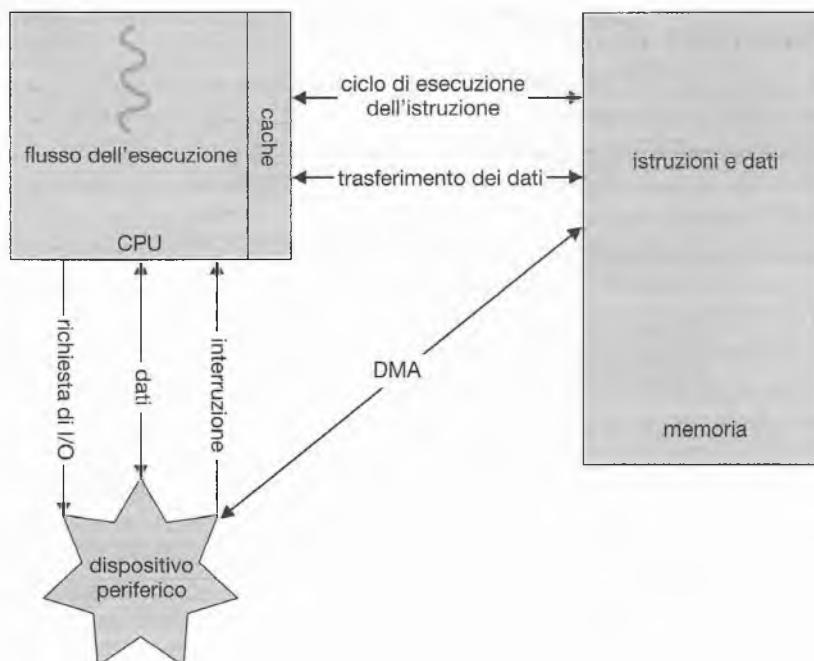


Figura 1.5 Funzionamento di un moderno sistema operativo.

### 1.3.1 Sistemi monoprocessoress

Sono molti i sistemi che usano un solo processore. La loro varietà può risultare stupefacente: si va dai PDA fino ai mainframe. Un sistema monoprocessoress è dotato di una CPU principale in grado di eseguire un insieme di istruzioni di natura generale, comprese quelle necessarie ai processi utenti. Quasi tutti i sistemi, inoltre, possiedono altri processori specializzati, deputati a compiti particolari. Essi possono assumere la forma di processori specifici di un dispositivo, quali i controllori del disco, della tastiera o del video; oppure, quando risiedono su mainframe, essere impiegati per finalità più generiche, come i processori dell'I/O che spostano dati da un dispositivo a un altro del sistema.

Tutti questi processori di tipo specifico sono dotati di un insieme ristretto di istruzioni, e non eseguono processi utenti. Talvolta sono guidati dal sistema operativo, che può inviare loro informazioni sul compito da espletare successivamente, e controllarne lo status. Prendiamo l'esempio di un microprocessore che funge da controllore del disco e che riceve dalla CPU un elenco di richieste; spetta a esso implementare una coda e un algoritmo di scheduling per gestirle. Questa strategia alleggerisce la CPU dal sovraccarico di lavoro che lo scheduling del disco può comportare. I PC ospitano un microprocessore all'interno della tastiera, per convertire la pressione di ciascun tasto nel codice appropriato da trasmettere alla CPU. In circostanze o sistemi differenti, i processori con finalità specifiche sono dispositivi di basso livello integrati nell'hardware. Il sistema operativo non può comunicare con questi processori, che svolgono in autonomia il proprio lavoro. L'utilizzo di microprocessori con finalità specifiche è comune e non trasforma un sistema monoprocessoress in un sistema multiprocessoress. Se è presente una sola CPU, si tratta di un sistema monoprocessoress.

### 1.3.2 Sistemi multiprocessoress

Sebbene i sistemi monoprocessoress siano ancora molto comuni, l'importanza dei sistemi multiprocessoress, conosciuti anche come sistemi paralleli o sistemi strettamente connessi (*tightly coupled system*), è in rapida crescita. Questo tipo di sistemi dispone di più unità d'elaborazione in stretta comunicazione, che condividono i canali di comunicazione all'interno del calcolatore (*bus*), i timer dei cicli di macchina (*clock*) e talvolta i dispositivi di memorizzazione e periferici.

Tali sistemi hanno tre vantaggi principali.

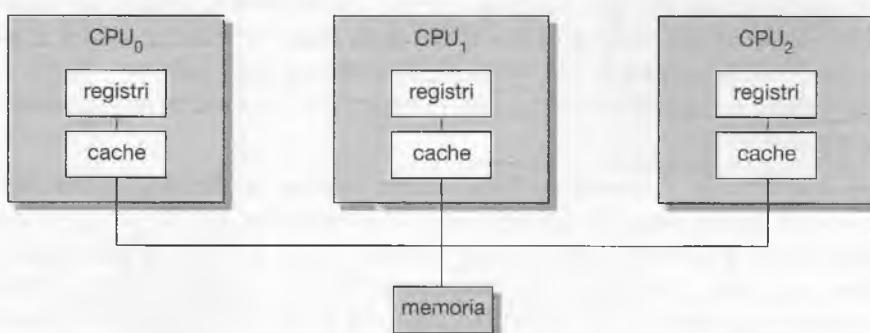
1. **Maggiore produttività (throughput).** Aumentando il numero di unità d'elaborazione è possibile svolgere un lavoro maggiore in meno tempo. Con  $n$  unità d'elaborazione la velocità non aumenta tuttavia di  $n$  volte, ma in misura minore. Infatti, se più unità d'elaborazione collaborano nell'esecuzione di un compito, il sistema operativo deve gestire le operazioni che garantiscono che tutti i componenti funzionino correttamente. Questo sovraccarico, unito alla contesa delle risorse condivise, riduce il guadagno atteso dalla disponibilità di più unità d'elaborazione; così come un gruppo di  $n$  programmatore che lavorano insieme non produce  $n$  volte quanto produrrebbe un solo programmatore.
2. **Economia di scala.** I sistemi multiprocessoress possono inoltre consentire risparmi rispetto a più sistemi dotati di una sola unità d'elaborazione, poiché nei primi si possono condividere dispositivi periferici, mezzi di registrazione dei dati e alimentatori elettrici. Se più programmi devono operare sullo stesso insieme di dati, è economicamente più conveniente registrarli in dischi condivisi da tutte le unità d'elaborazione, piuttosto che avere più calcolatori con i rispettivi dischi locali e più copie degli stessi dati.

3. **Incremento dell'affidabilità.** Se le funzioni si possono distribuire adeguatamente tra più unità d'elaborazione, un guasto di alcune di loro non blocca il sistema, semplicemente lo rallenta; ciascuna delle unità d'elaborazione rimanenti assume su di sé una parte del lavoro che svolgevano le unità d'elaborazione guaste; l'intero sistema non si ferma, ma funziona a una velocità ridotta.

Per molte applicazioni una maggiore affidabilità dell'elaboratore è di importanza cruciale. La capacità di continuare a offrire un servizio proporzionalmente commisurato, per qualità, ai dispositivi ancora in funzione è detta **degradazione controllata** (*graceful degradation*). Tali sistemi si spingono oltre la degradazione controllata e sono così definiti **tolleranti ai guasti** (*fault-tolerant*) perché, nonostante subiscano il danneggiamento di un qualunque singolo componente, continuano ugualmente a funzionare. La resistenza ai guasti, si noti, necessita di un meccanismo per il riconoscimento del danno, la sua diagnosi e, se è possibile, la riparazione. Il sistema HP NonStop (già noto come Tandem) duplica sia i dispositivi sia i programmi per garantire la continuità di funzionamento anche in caso di guasti. Esso è formato da coppie multiple della CPU, la cui attività è sincronizzata. Entrambi i processori di una coppia eseguono una data istruzione e confrontano i risultati. Risultati diversi segnalano un errore da parte di una delle CPU, e provocano il blocco di entrambe. Il processo che era in esecuzione è trasferito quindi a un'altra coppia di CPU, e riprende l'esecuzione a partire dall'istruzione a cui era avvenuto il blocco. Si tratta di una soluzione costosa, dato che implica l'uso di dispositivi dedicati e una notevole duplicazione delle risorse fisiche.

I sistemi multiprocessore attualmente in uso sono di due tipi. Alcuni impiegano la **multielaborazione asimmetrica** (*asymmetric multiprocessing*, AMP), in cui a ogni unità d'elaborazione si assegna un compito specifico. Un'unità d'elaborazione principale controlla il sistema, le altre attendono istruzioni dall'unità principale oppure hanno compiti predefiniti. Questo schema definisce una relazione gerarchica; l'unità d'elaborazione principale organizza e assegna il lavoro alle unità d'elaborazione secondarie.

Nei sistemi più comuni si ricorre alla **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP), in cui ogni processore è abilitato al compimento di tutte le operazioni del sistema. La tecnica SMP pone tutti i processori su un piano di parità; tra essi non vi è subordinazione gerarchica. La Figura 1.6 illustra una tipica architettura SMP. Un esempio della tecnica SMP è dato da Solaris, realizzato da Sun Microsystems quale versione commerciale di UNIX. Tale sistema può impiegare dozzine di processori, tutti gestiti dal sistema operativo Solaris. Il vantaggio offerto da questo modello è che molti processi sono eseguibili contemporaneamente ( $n$  processi se si hanno  $n$  CPU) senza causare un rilevante calo delle prestazioni. Per essere certi che i dati raggiungano le unità d'elaborazione giuste occorre però con-



**Figura 1.6** Architettura per la multielaborazione simmetrica.

trollare con molta attenzione le operazioni di I/O. Inoltre, poiché le unità d'elaborazione sono separate, una potrebbe essere inattiva mentre un'altra è sovraccarica, e ciò determinerebbe un'inefficienza evitabile se le unità d'elaborazione condividessero alcune strutture dati. Un sistema multiprocessore di questo tipo permetterebbe infatti di condividere dinamicamente processi e risorse (per esempio la memoria) tra le varie unità d'elaborazione, riducendo la varianza tra di loro. Come si vedrà nel Capitolo 6, un sistema di questo tipo deve essere scritto con molta cura. Tutti i sistemi operativi moderni (tra i quali il Windows, Windows XP, Mac OS X e Linux) ora gestiscono la multielaborazione simmetrica.

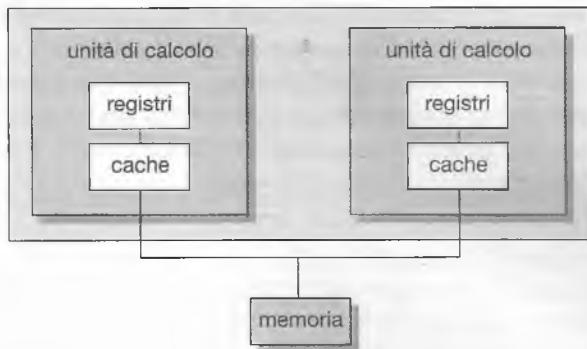
La differenza tra multielaborazione simmetrica e asimmetrica può derivare da caratteristiche sia dell'architettura del sistema sia da caratteristiche del sistema operativo: una speciale architettura può differenziare le unità d'elaborazione, oppure si può avere un sistema operativo che definisce una sola unità d'elaborazione primaria e più unità d'elaborazione secondarie. Per esempio, la versione 4 del sistema operativo SunOS della Sun consentiva la multielaborazione asimmetrica, mentre la versione 5 (Solaris) permette, sulla stessa architettura, la multielaborazione simmetrica.

Per aumentare la potenza di calcolo nella multielaborazione si aggiungono nuove CPU. Se la CPU ha un controllo di memoria integrato, allora l'aggiunta di CPU può aumentare la quantità di memoria indirizzabile dal sistema. D'altra parte, la multielaborazione può causare il cambiamento del modello di accesso alla memoria del sistema, trasformando un accesso uniforme alla memoria (UMA) in accesso non uniforme alla memoria (NUMA). Per accesso uniforme alla memoria si intende la situazione in cui l'accesso a una RAM da una qualsiasi CPU richiede lo stesso tempo. In caso di NUMA, invece, l'accesso ad alcune parti della memoria necessita di un tempo maggiore rispetto ad altre, con un conseguente peggioramento delle prestazioni. I sistemi operativi possono minimizzare la penalizzazione causata dal NUMA grazie a un'oculata gestione delle risorse, come discusso nel Paragrafo 9.5.4.

Una tendenza recente nella progettazione della CPU è raggruppare diverse unità di calcolo (*core*) in un singolo circuito. Si tratta, sostanzialmente, di circuiti integrati multiprocessore. Questi circuiti possono essere più efficienti rispetto a più circuiti dotati di una singola unità di calcolo, perché la comunicazione all'interno di un singolo circuito è più veloce rispetto a quella tra un circuito e un altro. Inoltre, un circuito dotato di diverse unità di calcolo usa meno potenza di diversi circuiti con una singola unità di calcolo. Ne segue che i sistemi a multipla unità di calcolo sono particolarmente indicati per sistemi server come banche dati e server web.

Nella Figura 1.7 è illustrata un'architettura a doppia unità di calcolo (*dual core*), con due unità sullo stesso circuito. In un'architettura di questo tipo ogni unità di calcolo ha il proprio insieme di registri e la propria cache; altre architetture possono prevedere l'utilizzo di una cache condivisa oppure una combinazione di cache locali e condivise. A prescindere dalle considerazioni riguardanti le architetture, come la competizione per l'uso della cache, della memoria e del bus, queste CPU a unità multipla appaiono al sistema operativo come *N* processori ordinari. Chi progetta sistemi operativi e chi programma applicazioni è quindi spinto all'impiego di questo tipo di CPU.

Infine, accenniamo a una delle ultime innovazioni, i cosiddetti **server blade**, che accolgono nello stesso contenitore fisico le schede del processore, dell'I/O e della rete. A differenza dei tradizionali sistemi multiprocessore, nei server blade ogni scheda madre (una scheda, cioè, che ospita una CPU) avvia ed esegue in maniera indipendente il proprio sistema operativo. Alcune di queste schede, poi, possono essere a loro volta multiprocessore, il che rende più labile la distinzione tra tipi diversi di computer. Si può affermare, in sintesi, che tali server sono costituiti da svariati sistemi multiprocessore indipendenti.



**Figura 1.7** Architettura a doppia unità di calcolo, con due unità sullo stesso circuito.

### 1.3.3 Cluster di elaboratori

Come per i sistemi multiprocessore, i cluster di elaboratori (*clustered systems*) o cluster sono basati sull'uso congiunto di più unità d'elaborazione riunite per lo svolgimento di attività d'elaborazione comuni. Differiscono dai sistemi paralleli per il fatto che sono composti di due o più calcolatori completi collegati tra loro. In realtà per tali sistemi non esiste una definizione precisa; c'è un dibattito aperto tra i fornitori delle varie offerte commerciali su tale definizione e sul perché una soluzione sia migliore di un'altra. La definizione generalmente accettata è che si tratta di calcolatori che condividono la memoria di massa, connessi per mezzo di una rete locale (LAN), (Paragrafo 1.10), o connessioni più veloci come per esempio InfiniBand.

Di solito si adotta questo tipo di soluzione per offrire un'elevata disponibilità. Ciascun calcolatore esegue una serie di programmi che forma uno strato di gestione del cluster; ogni nodo può tenere sotto controllo (attraverso la LAN) uno o più degli altri nodi. Se si presenta un malfunzionamento, il calcolatore che svolge il controllo può appropriarsi dei mezzi di memorizzazione del calcolatore malfunzionante e riavviare le applicazioni che erano in esecuzione. Gli utenti e i client delle applicazioni notano solo una breve interruzione del servizio.

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

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

### CLUSTER "BEOWULF"

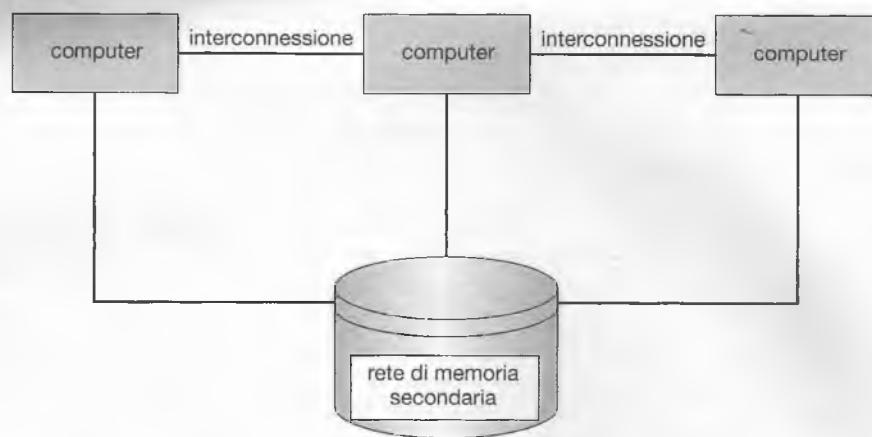
I cluster di tipo Beowulf sono progettati per risolvere problemi di calcolo che richiedono elevate prestazioni. Sono costruiti utilizzando prodotti hardware, ad esempio dei personal computer, collegati da una semplice rete locale. La peculiarità più interessante dei cluster Beowulf consiste nel fatto che non utilizzano degli specifici pacchetti software, ma piuttosto una serie di librerie di software open-source che permettono ai nodi di elaborazione del cluster di comunicare. Esiste quindi una grande varietà di approcci per la costruzione di un cluster Beowulf, sebbene i nodi di elaborazione montino solitamente il sistema operativo Linux. I cluster Beowulf, non richiedendo hardware specifico e impiegando software open-source disponibile gratuitamente, offrono una soluzione a basso costo per la costruzione di un cluster ad alte prestazioni. Alcuni cluster Beowulf costituiti da vecchi personal computer sono formati da centinaia di nodi di elaborazione e utilizzati nel calcolo scientifico per risolvere problemi molto impegnativi dal punto di vista computazionale.

Altre forme sono i cluster di sistemi paralleli e quelli di sistemi connessi attraverso reti geografiche (WAN). I primi permettono a più calcolatori di accedere agli stessi dati nella memoria di massa condivisa. Poiché la maggior parte dei sistemi operativi non consente quest'accesso simultaneo ai dati da parte di più calcolatori, si ricorre a programmi specifici e particolari versioni delle applicazioni. Per esempio, l'Oracle Real Application Cluster è una versione del sistema di gestione delle basi di dati Oracle, progettata per funzionare in cluster di sistemi paralleli. Ogni calcolatore esegue l'applicazione Oracle e uno strato di programmi controlla l'accesso ai dischi condivisi, in questo modo ogni calcolatore del sistema ha accesso alla base di dati. Per ottenere questo accesso condiviso ai dati, il sistema deve anche prevedere il controllo dell'accesso e la mutua esclusione, in modo da evitare sul nascere i conflitti tra operazioni. Questa funzione, detta **gestione distribuita degli accessi** (*distributed lock manager*, DLM), è attiva in alcuni cluster di elaboratori.

La ricerca in quest'ambito sta attraversando una fase di rapida evoluzione. Alcuni prodotti di questo genere ospitano dozzine di sistemi in un solo cluster, e possono accorpare in un unico insieme nodi distanti chilometri. Tali progressi si devono in buona misura a **reti di memoria secondaria** (*storage-area network*, SAN), illustrati nel Paragrafo 12.3.3, grazie alle quali è possibile far sì che molti sistemi accedano a un'unica riserva di spazio per archiviare informazioni. Se le applicazioni e i relativi dati sono memorizzati in una SAN, il software del cluster può smistare l'applicazione verso una qualunque delle macchine che vi hanno accesso. Qualora venga meno una di loro, può subentrare qualsiasi altra macchina. Se una base di dati è implementata su un cluster di sistemi, decine di macchine possono condividerne il contenuto, con notevole aumento delle prestazioni e dell'affidabilità. La Figura 1.8 rappresenta la struttura generale di un cluster.

## 1.4 Struttura del sistema operativo

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

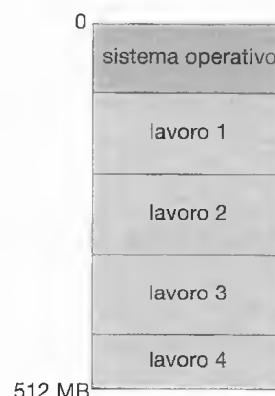


**Figura 1.8** Struttura generale di un cluster.

Fra le più importanti caratteristiche dei sistemi operativi vi è la multiprogrammazione. In generale, un singolo utente non è in grado di tenere costantemente occupata la CPU e i dispositivi di I/O: la multiprogrammazione consente di aumentare la percentuale d'utilizzo della CPU, organizzando i lavori in modo tale da mantenerla in continua attività.

L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale diversi lavori (Figura 1.9). Dato che, in genere, la memoria centrale è troppo piccola per contenere tutti i programmi da eseguire, questi vengono collocati inizialmente sul disco in un'area apposita, detta **job pool**, contenente tutti i processi in attesa di essere allocati nella memoria centrale.

L'insieme dei programmi caricati in memoria è generalmente un sottoinsieme dei lavori contenuti nel **job pool**. Il sistema operativo ne sceglie uno tra quelli contenuti nella memoria e inizia l'esecuzione: a un certo punto potrebbe trovarsi nell'attesa di qualche evento, come il completamento di un'operazione di I/O. In questi casi, in un sistema non multiprogrammato, la CPU rimarrebbe inattiva. In un sistema con multiprogrammazione, invece, il sistema operativo passa semplicemente a un altro lavoro e lo esegue. Quando il primo lavoro ha terminato l'attesa, la CPU ne riprende l'esecuzione. Finché c'è almeno un lavoro da eseguire, la CPU non è mai inattiva.



**Figura 1.9** Configurazione della memoria per un sistema con multiprogrammazione.

Ritroviamo quest'idea anche in altre circostanze della vita comune. Un avvocato, per esempio, non lavora per un solo cliente alla volta: mentre un caso aspetta di essere dibattuto o si attende la stesura dei relativi documenti, l'avvocato può lavorare a un altro caso; se ha abbastanza clienti, non sarà mai inattivo per mancanza di lavoro. (Gli avvocati inattivi tendono a trasformarsi in politici, quindi tenerli occupati ha un certo valore sociale.)

Un sistema con multiprogrammazione fornisce un ambiente in cui le risorse del sistema (per esempio CPU, memoria, dispositivi periferici) sono impiegate in modo efficiente, ma non rappresenta un sistema d'interazione con l'utente. La **partizione del tempo d'elaborazione** (*time sharing* o *multitasking*) è un'estensione logica della multiprogrammazione; la CPU esegue più lavori commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente l'interazione col proprio programma durante la sua esecuzione.

Un **sistema di calcolo interattivo** permette la comunicazione diretta tra utente e sistema. L'utente impartisce le istruzioni direttamente al sistema operativo oppure a un programma, attraverso una tastiera o un mouse, e attende una risposta immediata. Il **tempo di risposta** dovrebbe perciò essere breve, in genere meno di un secondo.

Un sistema operativo a partizione del tempo d'elaborazione permette a più utenti di condividere contemporaneamente il calcolatore. Poiché le azioni e i comandi eseguiti in un sistema a partizione del tempo sono tendenzialmente brevi, a ciascun utente basta poter usare una piccola parte del tempo di calcolo della CPU. Il sistema passa rapidamente da un utente all'altro, quindi ogni utente ha l'impressione di disporre dell'intero calcolatore, che in realtà è condiviso da molti utenti.

Per assicurare a ciascun utente una piccola frazione del tempo di calcolo, un sistema operativo a partizione del tempo d'elaborazione si avvale dello scheduling della CPU e della multiprogrammazione. Ciascun utente dispone di almeno un proprio programma in memoria. Un programma caricato in memoria e predisposto per la fase d'esecuzione è noto come **processo**. Normalmente un processo, durante la sua esecuzione, impegnà la CPU per un breve periodo di tempo prima di richiedere operazioni di I/O o di terminare; tali operazioni possono essere interattive, cioè i risultati sono inviati a uno schermo a disposizione dell'utente, che immette dati tramite una tastiera, un mouse, o altro. I tempi di tali operazioni risentono della lentezza del lavoro umano; l'immissione di dati e comandi tramite una tastiera, per esempio, è limitata dalla velocità di battitura dell'utente, che, per elevata che sia, è sempre molto bassa rispetto ai tempi d'elaborazione di un calcolatore: sette caratteri al secondo sono tanti per una persona, ma pochissimi per un calcolatore. Anziché lasciare inattiva la CPU, durante l'immissione interattiva il sistema operativo commuta rapidamente la CPU al programma di un altro utente.

La partizione del tempo di elaborazione e la multiprogrammazione richiedono la contemporanea presenza di diversi processi in memoria. Se alcuni processi sono pronti per il trasferimento in memoria centrale, ma lo spazio disponibile non è sufficiente per accoglierli tutti, il sistema deve fare una selezione; questa scelta, illustrata nel Capitolo 5, si chiama **job scheduling**, ossia *pianificazione dei lavori*. Quando il sistema operativo seleziona un processo dall'insieme dei lavori, carica quel processo in memoria perché sia eseguito. La coesistenza di un certo numero di programmi in memoria nello stesso lasso di tempo richiede una qualche forma di gestione della memoria, argomento esposto nei Capitoli 8 e 9. Inoltre, l'esistenza di diversi processi pronti per l'esecuzione nello stesso istante impone al sistema di scegliere solo alcuni di essi: i problemi inerenti a questa scelta, ovvero lo **scheduling della CPU** (*pianificazione della CPU*), sono descritti nel Capitolo 5. Infine, l'esecuzione corrente di processi multipli rende necessario limitare il più possibile le loro interferenze reciproche sotto ogni aspetto del funzionamento del sistema, compresi lo scheduling dei pro-

cessi e la gestione del disco e della memoria. Tali problematiche sono affrontate di volta in volta attraverso il libro.

In un sistema basato sulla partizione del tempo di elaborazione, il sistema operativo deve garantire tempi di risposta accettabili: questa finalità è raggiunta, in alcuni casi, grazie alla tecnica detta **swapping** (*avvicendamento*), che consente di scambiare i processi presenti in memoria con quelli che risiedono su disco e viceversa. Un metodo più comune per ottenere il medesimo risultato è la **memoria virtuale**, tecnica che consente l'esecuzione di lavori d'elaborazione anche non interamente caricati nella memoria (Capitolo 9). Il più evidente vantaggio della memoria virtuale è che i programmi possono avere dimensioni maggiori della **memoria fisica**; inoltre, essa astrae la memoria centrale in un grande e uniforme vettore, separando la **memoria logica**, vista dall'utente, dalla memoria fisica, sollevando i programmatore dai problemi legati ai limiti della memoria.

I sistemi a partizione del tempo devono inoltre fornire un *file system* (Capitoli 10 e 11) residente in un insieme di dischi, i quali a loro volta necessitano di una gestione (Capitolo 12). I sistemi a partizione del tempo dispongono di meccanismi per la protezione delle risorse rispetto a utenti non autorizzati (Capitolo 14). Per assicurare che tale esecuzione sia disciplinata il sistema deve fornire meccanismi per la comunicazione e sincronizzazione dei processi (Capitolo 6) e garantire che i processi non s'inceppino in una situazione di stallo, in un'indefinita attesa reciproca (Capitolo 7).

## 1.5 Attività del sistema operativo

Come si è avuto modo di accennare, i moderni sistemi operativi sono guidati dalle interruzioni. Quando i dispositivi dell'I/O non richiedono alcun servizio, in assenza di processi da eseguire e di utenti a cui rispondere, il sistema operativo rimane inerte e attende che accada qualcosa. Quasi sempre, è un'interruzione o un'eccezione a segnalare gli eventi. Un segnale di eccezione (*trap* o *exception*) indica che si è verificata un'interruzione generata da un programma, dovuta a un errore (per esempio, una divisione per zero o l'accesso illegale alla memoria), o alla richiesta di erogazione, da parte di un programma utente, di uno dei servizi del sistema operativo. La struttura generale di un sistema operativo è definita proprio in quanto esso è guidato dalle interruzioni. A ciascun tipo di interruzione corrispondono nel sistema singoli segmenti di codice, che determinano la reazione all'interruzione; apposite routine per il servizio delle interruzioni hanno il compito di fornire loro una risposta adeguata.

Dal momento che il sistema operativo e gli utenti condividono la dotazione di programmi e dispositivi dell'elaboratore, è necessario assicurarsi che un errore provocato da un programma utente non possa arrecare danno ad altri programmi. In un ambiente condiviso, infatti, l'errore di un solo programma ha un effetto negativo potenziale sugli altri processi. Se, per esempio, un processo rimane bloccato in un ciclo infinito, potrebbe essere impedito il corretto funzionamento di molti altri processi. I sistemi multiprogrammati, inoltre, sono esposti a rischi più sottili; si pensi a un programma viziato da errori, capace di modificare interamente un altro programma, i suoi dati e persino lo stesso sistema operativo.

Senza un'efficace protezione da errori come quelli menzionati, il calcolatore è costretto a eseguire un solo processo per volta, a meno di non voler considerare sospetti tutti i dati in uscita. Un sistema operativo progettato in maniera soddisfacente deve evitare che un programma viziato da errori (oppure volutamente dannoso) possa indurre un'esecuzione scorretta di altri programmi.

### 1.5.1 Dupliche modalità di funzionamento

Per garantire il corretto funzionamento del sistema è necessario distinguere tra l'esecuzione di codice del sistema operativo e di codice definito dall'utente. Il metodo seguito da molti sistemi operativi consiste nel disporre di specifiche caratteristiche dell'architettura del sistema che consentono di gestire differenti modalità di funzionamento.

Sono necessarie almeno due diverse modalità: **modalità utente** e **modalità di sistema** (detta anche *modalità kernel*, *modalità supervisore*, *modalità monitor* o *modalità privilegiata*). Per indicare quale sia la modalità attivo, l'architettura della CPU deve essere dotata di un bit, chiamato appunto **bit di modalità**: di sistema (0) o utente (1). Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente. Quando l'elaboratore agisce per conto di un'applicazione utente, il sistema è in modalità utente. Tuttavia, allorquando l'applicazione utente rivolga una richiesta di servizio al sistema operativo (tramite una chiamata di sistema), per soddisfare la richiesta questi deve passare dalla modalità utente alla modalità di sistema. Ciò è esemplificato dalla Figura 1.10. Come vedremo, questa miglioria architettonica agevola il funzionamento del sistema anche in altre circostanze.

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

La duplice modalità di funzionamento (*dual mode*) consente la protezione del sistema operativo rispetto al comportamento degli utenti e viceversa. Questo livello di protezione si ottiene definendo come **istruzioni privilegiate** le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché la CPU consente l'esecuzione di queste istruzioni soltanto nella modalità di sistema, se si tenta di far eseguire in modalità utente un'istruzione privilegiata, la CPU non la esegue, ma la tratta come un'istruzione illegale inviando un segnale di eccezione al sistema operativo.

Un esempio di istruzione privilegiata è dato dall'istruzione per passare alla modalità *kernel*. Altri esempi si riferiscono al controllo dell'I/O, alla gestione del timer (*temporizzatore*) e delle interruzioni. Come si vedrà in seguito, vi sono molte istruzioni privilegiate accessorie.

Possiamo ora esaminare il ciclo di vita di un'istruzione e la sua esecuzione in un elaboratore. All'inizio il controllo appartiene al sistema operativo, dove le istruzioni sono eseguite in modalità di sistema. Nel momento in cui il controllo è ceduto a un'applicazione utente si entra in modalità utente. Alla fine, il controllo è restituito al sistema operativo mediante un'interruzione, un'eccezione o una chiamata di sistema.

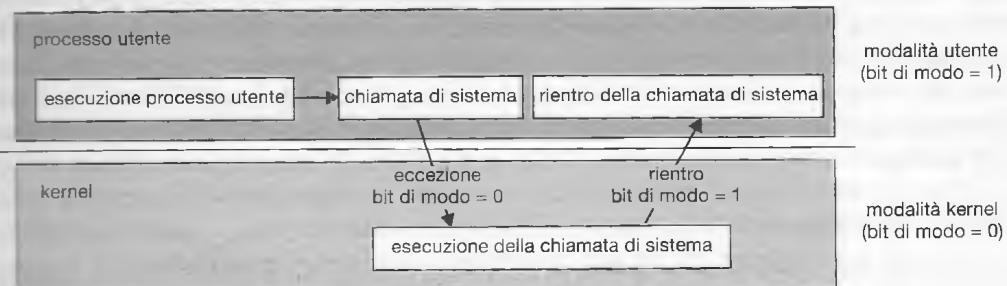


Figura 1.10 Transizione da modalità utente a modalità kernel.

Le chiamate di sistema sono gli strumenti con cui un programma utente richiede al sistema operativo di compiere operazioni a esso riservate, per conto del programma utente. Vi sono vari modi per generare una chiamata di sistema, a seconda delle funzionalità di cui dispone il processore della macchina. In ogni caso, però, le chiamate di sistema sono il mezzo utilizzato dai processi per sollecitare all'azione il sistema operativo. Una chiamata di sistema è solitamente realizzata come un'eccezione che rimanda a un indirizzo specifico nel vettore delle interruzioni. A tale eccezione si può dare esecuzione con un'istruzione `trap` generica, sebbene alcuni sistemi (come quelli appartenenti alla famiglia MIPS R2000) abbiano un'istruzione dedicata `syscall`.

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

Se la CPU non dispone di queste due modalità di funzionamento il sistema operativo può andare incontro a serie limitazioni. Il sistema MS-DOS per esempio è stato sviluppato per l'architettura 8088 Intel priva del bit di modo e, quindi, di entrambe le modalità. Un programma utente potrebbe cancellare il sistema operativo scrivendo nuove informazioni nelle locazioni in cui questo è memorizzato, e più programmi potrebbero scrivere contemporaneamente nello stesso dispositivo, con l'eventualità di ottenere risultati disastrosi. Versioni più recenti e progredite delle CPU Intel, come il Pentium, sono dotate della duplice modalità di funzionamento. I più recenti sistemi operativi sviluppati per tali architetture, come Microsoft Windows 2000, Windows XP e i sistemi Solaris x86 traggono vantaggio da questa caratteristica e garantiscono una maggiore protezione del sistema operativo.

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

### 1.5.2 Timer

Occorre assicurare che il sistema operativo mantenga il controllo dell'elaborazione, cioè impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema

senza più restituire il controllo al sistema operativo. A tale scopo si può usare un timer, programmabile affinché invii un segnale d'interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi (per esempio, di 1/60 di secondo) o variabili (per esempio, da un millisecondo a un secondo). Un timer **variabile** di solito si realizza mediante un generatore di impulsi a frequenza fissa e un contatore. Il sistema operativo assegna un valore al contatore, che si decrementa a ogni impulso e quando raggiunge il valore 0 si genera un segnale d'interruzione. Per esempio, un contatore di 10 bit con un generatore di impulsi con periodo di 1 millisecondo consente la generazione di interruzioni a intervalli compresi tra 1 e 1024 millisecondi, con incrementi di 1 millisecondo.

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

La presenza di un timer garantisce quindi che nessun programma utente possa essere eseguito troppo a lungo. Una tecnica semplice consiste nell'impostare un contatore con un valore pari al tempo concesso al programma per la propria esecuzione. Per esempio, se il programma richiede 7 minuti si dovrebbe impostare il contatore al valore 420. Il timer genererà un'interruzione ogni secondo e il contatore sarà decrementato di 1; fintanto che il valore resta positivo, il controllo ritorna al programma utente; quando il contatore raggiunge un valore negativo, il sistema operativo termina l'esecuzione del programma per il superamento del tempo a esso assegnato.

## 1.6 Gestione dei processi

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

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

Un programma di per sé non è un processo d'elaborazione; un programma è un'entità passiva, come il contenuto di un file memorizzato in un disco, mentre un processo è un'en-

tità *attiva*, con un **contatore di programma** che indica la successiva istruzione da eseguire (i thread sono trattati nel Capitolo 4). L'esecuzione di un processo deve essere sequenziale: la CPU esegue le istruzioni del processo una dopo l'altra, finché il processo termina; inoltre in ogni istante si esegue al massimo un'istruzione del processo. Quindi, anche se due processi possono corrispondere allo stesso programma, si considerano in ogni caso due sequenze d'esecuzione separate. Un processo multithread possiede più contatori di programma, ognuno dei quali punta all'istruzione successiva da eseguire per un dato thread. Il processo è l'unità di lavoro di un sistema. Tale sistema è costituito di un gruppo di processi, alcuni dei quali del sistema operativo (che eseguono codice di sistema), mentre altri sono processi utenti (che eseguono codice utente). Tutti questi processi si possono potenzialmente eseguire in modo concorrente, avvicendandosi nell'uso della CPU.

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

- ◆ creazione e cancellazione dei processi utenti e di sistema;
- ◆ sospensione e ripristino dei processi;
- ◆ fornitura di meccanismi per la sincronizzazione dei processi;
- ◆ fornitura di meccanismi per la comunicazione tra processi;
- ◆ fornitura di meccanismi per la gestione delle situazioni di stallo (*deadlock*).

Le tecniche di gestione dei processi sono trattate in modo approfondito nei Capitoli dal 3 al 6.

## 1.7 Gestione della memoria

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

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

Per migliorare l'utilizzo della CPU e la rapidità con la quale il calcolatore risponde ai propri utenti i computer a uso generale devono tenere molti programmi in memoria. Esistono diversi schemi di gestione della memoria; l'efficacia di ogni algoritmo dipende dalla situazione specifica. Poiché ogni algoritmo richiede specifiche caratteristiche, la scelta di un particolare schema di gestione della memoria dipende da molti fattori, principalmente dal tipo di *architettura* del sistema.

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

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

Le tecniche di gestione della memoria sono trattate in profondità nei Capitoli 8 e 9.

## 1.8 Gestione della memoria di massa

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

### 1.8.1 Gestione dei file

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

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

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

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

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

Le tecniche di gestione dei file sono trattate nei Capitoli 10 e 11.

### 1.8.2 Gestione della memoria di massa

Lo scopo principale di un sistema di calcolo è quello di eseguire programmi. Durante l'esecuzione, i programmi, insieme con i dati cui accedono, devono trovarsi nella memoria centrale. Giacché la memoria centrale è troppo piccola per contenere tutti i dati e tutti i programmi, e il suo contenuto va perduto se il sistema si spegne, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. La maggior parte dei moderni sistemi di calcolo impiega i dischi come principale mezzo di memorizzazione secondaria, sia per i programmi sia per i dati. I dischi contengono la maggior parte dei programmi, compresi i compilatori, gli assemblatori, le procedure di ordinamento e gli editor. Questi programmi rimangono nel disco fino al momento del caricamento in memoria e si servono del disco sia come sorgente sia come destinazione delle loro computazioni; per tale ragione è fondamentale che la registrazione nei dischi sia gestita correttamente. Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei dischi:

- ◆ gestione dello spazio libero;
- ◆ assegnazione dello spazio;
- ◆ scheduling del disco.

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

Vi sono però svariati frangenti in cui tornano utili dispositivi di memorizzazione più lenti, meno costosi e talora più capienti della memoria secondaria. Le copie di riserva dei dischi di sistema, i dati usati raramente e gli archivi a lungo termine sono alcuni esempi. Le unità a nastro magnetico, i CD e i DVD, sono tipici dispositivi di memoria terziaria. I media (nastri e i dischi ottici) comprendono i formati monoscrivibili WORM (*write once, read many times*) e riscrivibili RW (*read and write*).

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

Le tecniche di gestione della memoria secondaria e terziaria saranno approfondite nel Capitolo 12.

### 1.8.3 Cache

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

Inoltre, i registri programmabili presenti all'interno della CPU, come i registri indice, rappresentano per la memoria centrale una cache ad alta velocità. Il programmatore (o il compilatore) codifica gli algoritmi di assegnazione e aggiornamento dei registri in modo da stabilire quali informazioni mantenere nei registri e quali nella memoria centrale. Esistono anche cache che sono interamente gestite dall'architettura del sistema: la maggior parte dei

sistemi è dotata per esempio di una cache per la memorizzazione delle istruzioni che presumibilmente saranno eseguite dopo l'istruzione corrente. Senza di essa, la CPU dovrebbe attendere parecchi cicli prima che un'istruzione sia prelevata dalla memoria. Per simili motivi, la maggior parte dei sistemi è dotata di una o più cache di dati nella gerarchia delle memorie. In questo testo non ci si occupa di tali dispositivi, poiché sono componenti non controllabili dal sistema operativo.

Data la capacità limitata di questi dispositivi, la gestione della cache è un importante problema di progettazione. Da un'attenta selezione delle dimensioni e dei criteri di aggiornamento della cache può conseguire un notevole incremento delle prestazioni del sistema. La Figura 1.11 presenta un confronto tra le prestazioni di vari dispositivi per la memorizzazione in piccoli server e potenti stazioni di lavoro. La necessità delle memorie cache emerge chiaramente. Nel Capitolo 9 si discutono alcuni algoritmi per la sostituzione degli elementi contenuti nelle cache controllabili dal sistema operativo.

La memoria centrale si può considerare una cache per la memoria secondaria, giacché i dati in essa contenuti devono essere riportati nella memoria centrale per poter essere usati e qui devono risiedere prima di essere trasferiti nella memoria secondaria. I dati del file system, che risiedono in modo permanente nella memoria secondaria, possono apparire a diversi livelli della gerarchia di memoria. A livello più alto, il sistema operativo può mantenere una cache dei dati del file system in memoria centrale. Anche i dischi RAM si possono impiegare come veloci sistemi di memoria volatile cui si accede tramite l'interfaccia del file system. La memorizzazione secondaria si effettua generalmente in dischi magnetici; copie di riserva del loro contenuto spesso si registrano in nastri magnetici o dischi rimovibili. Alcuni sistemi, per ridurre i costi di memorizzazione, archiviano automaticamente i vecchi file di dati trasferendoli dalla memoria secondaria alla memoria terziaria, costituita per esempio da tape-box di nastri (Capitolo 12).

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

Livello	1	2	3	4
Nome	registri	cache	memoria centrale	disco
Dimensione tipica	< 1 KB	< 16 MB	< 16 GB	> 100 GB
Tecnologia	memoria dedicata con porte multiple (CMOS)	CMOS SRAM (on-chip o off-chip)	CMOS DRAM	disco magnetico
Tempo d'accesso (ns)	0,25 – 0,5	0,5 – 25	80 – 250	5.000.000
Aampiezza di banda (MB/s)	20.000 – 100.000	5000 – 10.000	1000 – 5000	20 – 150
Gestito da	compilatore	hardware	sistema operativo	sistema operativo
Supportato da	cache	memoria centrale	disco	CD o nastro

Figura 1.11 Prestazioni relative a varie forme di archiviazione dei dati.



**Figura 1.12** Migrazione di un intero A da un disco a un registro.

In una struttura gerarchica come quella appena introdotta, può accadere che gli stessi dati siano mantenuti contemporaneamente in diversi livelli del sistema di memorizzazione. Per esempio, si supponga di incrementare di 1 il valore di un numero intero A contenuto in un file B, registrato in un disco magnetico. L'operazione d'incremento prevede innanzitutto l'esecuzione di un'operazione di I/O per copiare nella memoria centrale il blocco di disco contenente il valore di A. Questa operazione è di solito seguita dalla copiatura di A all'interno della cache, e di qui in uno dei registri interni della CPU. Quindi, esistono diverse copie di A memorizzate nei vari dispositivi coinvolti nel trasferimento: nel disco magnetico, nella memoria centrale, nella cache e in un registro interno della CPU (Figura 1.12). Dopo l'incremento del valore della copia contenuta nel registro interno, il valore di A sarà diverso da quello assunto dalle altre copie della stessa variabile. Le diverse copie di A avranno lo stesso valore solamente dopo che il nuovo valore sarà stato riportato dal registro interno della CPU alla copia di A residente nel disco.

In un ambiente di calcolo che ammette l'esecuzione di un solo processo alla volta, questo modo di operare non pone particolari difficoltà, poiché ogni accesso ad A coinvolge la copia posta al livello più alto della gerarchia. Viceversa, nei sistemi a partizione del tempo d'elaborazione, in cui il controllo della CPU passa da un processo all'altro, è necessario prestare una particolare attenzione al fine di assicurare che qualsiasi processo che desideri accedere ad A ottenga dal sistema il valore della variabile aggiornato più di recente.

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

In un sistema distribuito la situazione diventa ancora più complessa, poiché può accadere che più copie (chiamate in questo caso **repliche**) dello stesso file siano mantenute in differenti calcolatori, dislocati in luoghi fisici diversi. Essendoci la possibilità di accedere e modificare in modo concorrente queste repliche, è necessario fare in modo che ogni modifica a una qualunque replica si rifletta quanto prima sulle altre. Nel Capitolo 17 si discutono diversi metodi che consentono di soddisfare questo requisito.

#### 1.8.4 Sistemi di I/O

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

- ◆ un componente di gestione della memoria comprendente la gestione delle regioni della memoria riservate ai trasferimenti di I/O (*buffer*), la gestione delle cache e la gestione asincrona delle operazioni di I/O e dell'esecuzione di più processi (*spooling*);
- ◆ un'interfaccia generale per i driver dei dispositivi;
- ◆ i driver per gli specifici dispositivi.

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

Nel Paragrafo 1.2.3 è stato presentato l'uso dei gestori dei segnali d'interruzione e dei driver dei dispositivi per la costruzione di sottosistemi di I/O efficienti. Nel Capitolo 13 si discute per esteso il modo in cui il sottosistema di I/O interagisce con gli altri componenti del sistema, come gestisce i dispositivi, trasferisce i dati e individua il completamento delle operazioni di I/O.

## 1.9 Protezione e sicurezza

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

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

La protezione può migliorare l'affidabilità rilevando errori nascosti nelle interfacce tra i componenti dei sottosistemi. Il beneficio che spesso deriva da una tempestiva rilevazione degli errori nelle interfacce è di prevenire la contaminazione di un sottosistema sano a opera di un altro sottosistema infetto. Non proteggere una risorsa significa lasciare via libera al suo utilizzo (o all'utilizzo scorretto) da parte di utenti incompetenti o non autorizzati. Un sistema informato da strategie di protezione offre i mezzi per distinguere l'uso autorizzato da quello non autorizzato, come si vedrà nel Capitolo 14.

Pur essendo dotato di protezione adeguata, un sistema può rimanere esposto agli accessi abusivi, rischiando malfunzionamenti. Si consideri un'utente le cui informazioni di autenticazione siano state trafugate. I suoi dati rischiano di essere copiati o cancellati, anche se è attiva la protezione dei file e della memoria. È compito della **sicurezza** difendere il sistema da attacchi provenienti dall'interno o dall'esterno. La varietà di minacce conosciute è enorme: si va dai virus e i worm, agli attacchi che paralizzano il servizio (appropriandosi di tutte le risorse del sistema e dunque escludendo da esso i legittimi utenti), al furto d'identità e la sottrazione del servizio (uso non autorizzato di un sistema). Per alcuni sistemi si può tenere che l'attività di prevenzione da attacchi siffatti appartenga a una funzione del sistema, mentre ve ne sono altri che delegano la difesa a regole esterne o a programmi aggiuntivi. A causa dell'allarmante incremento di incidenti legati alla sicurezza, le problematiche della sicurezza del sistema operativo si collocano in un settore che vede crescere rapidamente la ricerca e la sperimentazione. La sicurezza è analizzata nel Capitolo 15.

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

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

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

## 1.10 Sistemi distribuiti

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

Una rete si può considerare, in parole semplici, come un canale di comunicazione tra due o più sistemi. I sistemi distribuiti si basano sulle reti per realizzare le proprie funzioni, sfruttano la capacità di comunicazione per cooperare nella soluzione dei problemi di calcolo e per fornire agli utenti un ricco insieme di funzioni. Le reti differiscono per i protocolli usati, per le distanze tra i nodi e per il mezzo attraverso il quale avviene la comunicazione. Sebbene siano largamente usati sia il protocollo ATM sia altri protocolli, il più diffuso protocollo di comunicazione è il TCP/IP. Anche la disponibilità dei protocolli di rete è piuttosto varia nei diversi sistemi operativi. La maggior parte di essi, inclusi i sistemi Windows e UNIX, impiega il TCP/IP. Alcuni sistemi dispongono di propri protocolli che soddisfano esigenze specifiche. Affinché un sistema operativo possa gestire un protocollo di rete è necessa-

ria la presenza di un dispositivo d'interfaccia – un adattatore di rete, per esempio – con un driver per gestirlo, oltre al software per la gestione dei dati.

Le reti si classificano secondo le distanze tra i loro nodi: una **rete locale** (LAN) comprende nodi all'interno della stessa stanza, piano o edificio; una **rete geografica** (WAN) si estende a gruppi di edifici, città, o al territorio di una regione o di uno stato. Una società multinazionale, per esempio, potrebbe disporre di una rete WAN per connettere i propri uffici nel mondo. Queste reti possono funzionare con uno o più protocolli e il continuo sviluppo di nuove tecnologie fa sì che si definiscano nuovi tipi di reti. Le **reti metropolitane** (MAN) per esempio collegano gli edifici di un'intera città; i dispositivi BlueTooth e 802.11 comunicano a breve distanza, dell'ordine delle decine di metri, creando essenzialmente una **micronerete** (*small-area network*), simile a quelle domestiche.

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

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

I Capitoli dal 16 al 18 sono dedicati alle reti di calcolatori e ai sistemi distribuiti.

## 1.11 Sistemi a orientamento specifico

Gli argomenti esaminati finora si riferivano a sistemi di calcolo più familiari, ovvero quelli con finalità generali. Esistono, d'altra parte, varie categorie di sistemi di calcolo, inerenti a particolari aree della computazione, le cui funzioni sono più limitate.

### 1.11.1 Sistemi integrati real-time

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

Ciò che contraddistingue i sistemi integrati è la loro grande variabilità. Talvolta possono essere elaboratori di uso generale con sistemi operativi standard – come UNIX – che sfruttano applicazioni create appositamente per implementare una funzionalità. Altri sono dispositivi meccanici che ospitano un sistema operativo integrato a obiettivo specifico, che

consente di ottenere proprio la funzionalità desiderata. Inoltre, esistono dispositivi meccanici che hanno al loro interno circuiti integrati per applicazioni specifiche (ASIC), capaci di svolgere il loro lavoro senza un sistema operativo.

La diffusione dei sistemi integrati è in continua espansione. Indubbiamente, sono destinate a crescere anche le potenzialità applicative di questi congegni, sia come unità indipendenti sia in qualità di membri delle reti e di Internet. È già possibile l'automatizzazione di intere abitazioni, cosicché un computer centrale – sia che si tratti di un computer con finalità generali o di un sistema integrato – sia in grado di controllare il riscaldamento, l'illuminazione o i sistemi di allarme. Attraverso Internet, la propria casa può essere riscaldata a distanza prima di rientrarvi. Un giorno, il frigorifero potrà forse decidere autonomamente di chiamare il droghiere per il rifornimento del latte.

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

Questo tipo di sistema ha l'obbligo di rispettare categoricamente i propri vincoli di tempo, che sono definiti con precisione. L'elaborazione *deve* avvenire entro i limiti prestabiliti: in caso contrario, il sistema andrà in crisi. Per esempio, non serve a nulla che il braccio di un robot riceva l'ordine di fermarsi solo *dopo* essersi scontrato con l'automobile che era impegnato a costruire. Le prestazioni di un sistema real-time sono soddisfacenti solo se esso, generando il risultato corretto, rispetta precise scadenze. Si confronti questo tipo di sistema con un sistema a tempo ripartito (*time-sharing system*), in cui è auspicabile una risposta rapida (ma non obbligatoria), o con un sistema a lotti, che potrebbe essere del tutto esente da vincoli di tempo.

Dei sistemi integrati in tempo reale si riparerà, con dovizia di dettagli, nel Capitolo 19. Il Capitolo 5 esamina la tipologia di scheduling da adottare affinché un sistema operativo possa eseguire applicazioni real-time. La gestione della memoria nell'ambito dell'elaborazione in tempo reale è illustrata nel Capitolo 9. Infine, il Capitolo 22 è dedicato ai componenti real-time del sistema operativo Windows XP.

### 1.11.2 Sistemi multimediali

Gran parte dei sistemi operativi è stata concepita per la gestione di dati tradizionali, quali file di testo, programmi, documenti di videoscrittura e fogli di calcolo. Tuttavia, per effetto di una recente tendenza della tecnologia, negli elaboratori fanno la loro comparsa sempre più spesso i dati multimediali o dati a trasmissione continua. Essi consistono sia di file audio e video che di file tradizionali. Questi dati si distinguono da quelli tradizionali perché la loro trasmissione all'utente finale deve attenersi a precise frequenze: per esempio, nel caso di un video, 30 fotogrammi al secondo.

Multimediale è sinonimo, al giorno d'oggi, di varie applicazioni che incontrano il gradimento del pubblico. Alcune di esse sono i file audio MP3, i film in DVD, la video-conferenza, i video clip contenenti anteprime cinematografiche o notiziari. Tra le applicazioni multimediali citiamo anche gli eventi trasmessi in diretta sul World Wide Web, relativi a discorsi o gare sportive, e persino le webcam dal vivo con cui uno spettatore a Manhattan può osservare gli avventori di un caffè parigino. Le applicazioni multimediali non devono neces-

sariamente essere o solo audio, o solo video; infatti, un'applicazione multimediale sovente comprende una combinazione dei due. Un film, per esempio, si compone di tracce audio e video separate. Né queste applicazioni sono destinate unicamente ai personal computer; sempre di più, esse sono dirette a prodotti più piccoli quali gli assistenti personali digitali (PDA) e i telefoni cellulari. Un intermediario sui mercati finanziari, a titolo di esempio, potrà seguire l'andamento delle azioni in tempo reale sul proprio PDA, tramite una connessione senza fili.

Nel Capitolo 20 prenderemo in analisi le esigenze delle applicazioni multimediali. Apprenderemo ciò che distingue i dati multimediali da quelli tradizionali, sottolineando l'impatto di tali differenze sulla progettazione di sistemi operativi idonei a soddisfare i requisiti dei sistemi multimediali.

### 1.11.3 Sistemi palmari

I sistemi palmari (*handheld systems*) comprendono gli assistenti digitali, noti come PDA (*personal digital assistant*), come i computer palmari e tascabili (*palm-pilot*) o i telefoni cellulari, molti dei quali adottano sistemi operativi integrati a orientamento specifico. I progettisti di sistemi e applicazioni per i sistemi palmari devono affrontare molti problemi, alcuni dei quali correlati alle dimensioni di tali dispositivi. I PDA più diffusi sono lunghi circa dodici centimetri, larghi circa otto, e pesano tra cento e duecentocinquanta grammi. A causa delle piccole dimensioni, la maggior parte dei dispositivi palmari dispone di una memoria limitata, unità d'elaborazione lente e schermi piccoli.

La quantità di memoria di molti dispositivi palmari varia a seconda del modello, ma in genere oscilla tra 1 MB e 1 GB: assai meno di quella di un comune PC o stazione di lavoro, che può essere di diversi gigabyte. Per questo motivo, il sistema operativo e le applicazioni devono gestire la memoria in modo efficiente. Ciò implica, tra l'altro, restituire al gestore della memoria l'intera memoria assegnata, una volta che questa non è più usata. Il Capitolo 9 tratta le tecniche di memoria virtuale, che permettono ai programmati di scrivere programmi che si comportano come se il sistema avesse più memoria di quella fisicamente disponibile. Attualmente, molti dispositivi palmari non impiegano tecniche di memoria virtuale, quindi impongono ai programmati di lavorare entro i limiti della memoria fisica.

Un secondo aspetto che riguarda i programmati di questi dispositivi è la velocità delle CPU, che di solito è solo una frazione di quella di una CPU per PC. D'altra parte, le CPU più veloci consumano più energia, quindi richiederebbero l'impiego di cluster di dimensioni maggiori e la necessità di cambiarle (o ricaricarle) più spesso. Per ridurre al minimo le dimensioni dei dispositivi palmari si usano di solito CPU più piccole, più lente e che consumano meno energia. Quindi, il sistema operativo e le applicazioni si devono progettare in modo da non gravare eccessivamente la CPU.

Un ulteriore problema da affrontare da parte degli sviluppatori di questi dispositivi riguarda le questioni di I/O: a causa dello spazio esiguo, i metodi per l'inserimento dei dati sono limitati all'uso di piccole tastiere, al riconoscimento della scrittura, o a ridotte tastiere simulate sullo schermo. Poiché lo schermo è di modeste dimensioni, le opzioni disponibili per la rappresentazione dei dati in uscita sono limitate. Mentre nei PC sono comuni schermi di 30 pollici, quelli dei dispositivi palmari di solito non superano i 6 × 9 centimetri. Attività comuni come la lettura della posta elettronica o la consultazione del Web si devono condensare in schermi molto piccoli. Un metodo per mostrare il contenuto di pagine web è il servizio di *ritagli di pagine* (*web clipping*), che prevede l'invio di sottoinsiemi delle pagine da mostrare nel dispositivo palmare.

Alcuni dispositivi palmari possono servirsi di tecnologie di comunicazione senza fili, come il sistema BlueTooth o 802.11, che consentono l'accesso al servizio di posta elettronica e al Web. I telefoni cellulari che permettono la connessione alla rete Internet appartengono a questa categoria. Tuttavia, molti PDA attualmente non prevedono l'accesso senza fili; i dati da trasferire a questi dispositivi di solito si trasmettono prima a un PC o a una stazione di lavoro e successivamente al PDA. Alcuni permettono la copiatura diretta dei dati tra dispositivi per mezzo di una connessione a raggi infrarossi.

Generalmente, l'utilità e la portabilità dei PDA compensano i limiti delle loro funzioni. La loro diffusione è in continua espansione rispetto alla connessione alle reti e la disponibilità di altri accessori – macchine fotografiche, lettori MP3, ecc. – li rendono sempre più utili.

## 1.12 Ambiente d'elaborazione

Finora è stata offerta una panoramica dell'organizzazione dei sistemi di elaborazione e dei principali componenti di un sistema operativo. Il capitolo si conclude con una breve introduzione del loro uso in ambienti d'elaborazione diversi.

### 1.12.1 Elaborazione tradizionale

Con l'evoluzione delle tecniche d'elaborazione, i confini tra molti ambienti d'elaborazione tradizionale diventano sempre più sfumati. Si consideri per esempio un tipico ambiente d'ufficio: solo pochi anni fa consisteva di PC connessi in rete, con server che fornivano servizi di accesso ai file e di stampa. L'accesso remoto era difficoltoso e la portabilità si otteneva grazie ai calcolatori portatili che permettevano il trasferimento di una parte dello spazio di lavoro dell'utente. I terminali connessi ai mainframe erano ancora i più diffusi nelle grandi aziende, con ancora meno funzioni d'accesso remoto e di portabilità.

Attualmente si tende ad avere più modi d'accesso a questi ambienti; le tecnologie del Web stanno estendendo i confini del calcolo tradizionale, le aziende realizzano i cosiddetti **portali** che permettono l'accesso tramite il Web ai propri server interni. I calcolatori di rete sono essenzialmente terminali adatti all'elaborazione basata sul Web. I sistemi palmari possono sincronizzarsi con i PC per consentire un uso estremamente portatile delle informazioni aziendali; i PDA permettono anche la connessione a reti senza fili per accedere al portale web dell'azienda (e alle tantissime altre risorse del Web).

In casa, la maggior parte degli utenti aveva un solo calcolatore con una lenta connessione via modem all'ufficio, alla rete Internet, o a entrambi. Le connessioni di rete veloci, un tempo possibili a costi molto alti, sono ora disponibili a prezzi abbastanza contenuti e permettono l'accesso a maggiori quantità di dati. Queste connessioni veloci consentono ai calcolatori di casa di trasformarsi in server web e di formare reti con stampanti, PC client e server. Alcuni ambienti d'elaborazione domestici sono dotati anche di **firewall** (*barriera antintrusione*) che proteggono dagli attacchi informatici esterni; si tratta di sistemi che solo qualche anno fa costavano migliaia di euro e dieci anni fa nemmeno esistevano.

Nella seconda metà del '900 le risorse elettroniche di calcolo erano scarse (e prima ancora, non esistevano!). C'è stato un periodo di tempo in cui i sistemi erano o a lotti oppure interattivi. I sistemi a lotti elaboravano i processi all'ingrosso, per così dire, con un input predeterminato (da file o da altre fonti). I sistemi interattivi aspettavano di ricevere i dati in ingresso dagli utenti. Per ottenere la massima resa dall'elaboratore, diversi utenti si alternavano su questi sistemi. I sistemi a tempo ripartito impiegavano un timer e algoritmi di sche-

duling per assegnare rapidamente i processi alla CPU, attribuendo a ogni utente una parte delle risorse.

Attualmente, i tradizionali sistemi a tempo ripartito sono rari. La corrispondente strategia di scheduling è tuttora usata dai server e nelle stazioni di lavoro, ma spesso tutti i processi fanno capo allo stesso utente (o a un utente singolo e al sistema operativo). I processi dell'utente, e i processi del sistema che forniscono servizi all'utente, sono gestiti in modo da ricevere entrambi, frequentemente, una fetta del tempo a disposizione. Ciò si può notare, per esempio, osservando le finestre esistenti durante il lavoro di un utente al calcolatore, e notando che molte fra loro possono eseguire nel contempo operazioni diverse.

### 1.12.2 Computazione client-server

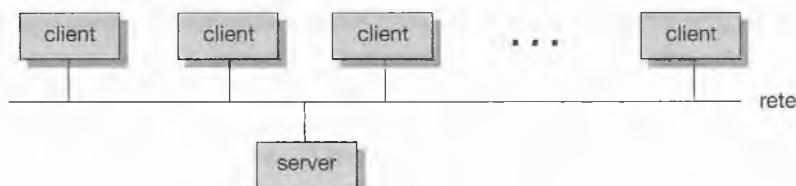
Di pari passo all'aumento di velocità, potenza ed economicità dei PC, i progettisti hanno abbandonato il modello di architettura centralizzata per i sistemi. I PC stanno prendendo il posto dei terminali connessi ai grandi sistemi centralizzati. A riprova di questo fenomeno, accade sempre più spesso che i PC gestiscano in proprio la funzionalità d'interfaccia con l'utente, di cui prima si occupavano i sistemi a livello centrale. Ecco perché molti sistemi odierni fungono da sistemi server per le richieste che ricevono dai sistemi client. Questa variante specialistica dei sistemi distribuiti, che prende il nome di sistema client-server, ha la struttura generale rappresentata nella Figura 1.13.

Schematicamente possiamo suddividere i sistemi server come server per l'elaborazione e file server.

- ◆ I **server per l'elaborazione** forniscono un'interfaccia a cui i client possono inviare una richiesta (per esempio, la lettura di alcuni dati); da parte sua, il server esegue l'azione richiesta e restituisce i risultati al client. Un server che ospita una base di dati a cui i client possono attingere costituisce un esempio di tale sistema.
- ◆ I **file server** offrono un'interfaccia al file system che consente al client creazione, aggiornamento, lettura e cancellazione dei file. Un esempio di questo sistema è dato da un server web che trasferisce i file richiestigli dai browser dei client.

### 1.12.3 Computazione peer-to-peer

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



**Figura 1.13** Struttura generale di un sistema client-server.

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

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

Le reti paritetiche hanno avuto larga diffusione dalla fine degli anni '90 grazie a diversi servizi di condivisione dei file, quali Napster e Gnutella, che permettono agli utenti lo scambio reciproco di file. Il sistema Napster utilizza una modalità operativa simile al primo tipo descritto precedentemente: un server centrale conserva il registro di tutti i file prelevabili dai nodi della rete Napster, e lo scambio effettivo dei file avviene su iniziativa dei nodi stessi. Il sistema Gnutella adotta una tecnica simile al secondo tipo descritto: un client fa pervenire le richieste di file agli altri nodi del sistema, quindi i nodi che possono soddisfare la richiesta rispondono direttamente al client. In prospettiva futura lo scambio dei file lascia adito a dubbi, visto che molti di loro sono tutelati dalle leggi sui diritti d'autore (i brani musicali, per esempio) che ne regolamentano la distribuzione. In ogni caso, la tecnologia peer-to-peer giocherà, con ogni probabilità, un ruolo determinante nell'ambito di molti servizi, quali la ricerca all'interno di una rete, lo scambio di file e la posta elettronica.

#### 1.12.4 Computazione basata sul Web

La rete Internet è diventata onnipresente, come dimostra il fatto che vi accedono molti più dispositivi di quanti se ne potessero immaginare solo fino a qualche anno fa. I PC, insieme alle stazioni di lavoro, ai PDA e persino ai telefoni cellulari, sono tuttora i dispositivi d'eccellenza per la connessione al Web.

La computazione basata sul Web ha accentuato l'importanza dell'interconnessione in rete. Alcuni dispositivi che prima non avevano la possibilità di collegarsi, garantiscono ora l'accesso, con o senza l'ausilio di cavi. I dispositivi che già godevano della connessione possono ora usufruire di collegamenti di qualità e velocità superiori, che derivano da una migliore tecnologia delle reti, dal perfezionamento dei programmi che gestiscono le reti, o da entrambi i fattori.

La computazione basata sul Web ha dato origine a nuovi dispositivi, quali i **bilanciatori del carico** (*load balancers*), che distribuiscono le connessioni di rete su una serie di server simili. I sistemi operativi come Windows 95, che aveva il ruolo di client web, si sono evoluti in Linux e Windows XP, che possono fungere sia da server web che da client web. Più in generale, il Web ha avuto come effetto collaterale un incremento di complessità dei dispositivi, che devono adesso garantire la possibilità di operare sul Web.

## 1.13 Sistemi operativi open-source

Lo studio dei sistemi operativi, come già sottolineato, è semplificato dalla disponibilità di un vasto numero di programmi open-source. I **sistemi operativi open-source** sono disponibili in formato sorgente anziché come codice binario compilato. Linux è il più diffuso sistema operativo open-source, mentre Microsoft Windows è un ben noto esempio dell'approccio opposto, a **sorgente chiuso** (*closed source*). Avere a disposizione il codice sorgente permette al programmatore di produrre il codice binario, eseguibile da un sistema. Il processo inverso, chiamato processo di **reverse-engineering**, che permette di ricavare il codice sorgente partendo dal binario, è molto più oneroso; molti elementi utili, ad esempio i commenti, non possono essere ripristinati. Apprendere il funzionamento dei sistemi operativi esaminandone il codice sorgente originale, piuttosto che leggendo descrizioni di quel codice sorgente, può essere molto utile. Avendo a disposizione il codice sorgente, uno studente può modificare il sistema operativo per poi compilare ed eseguire il codice, verificando così i cambiamenti che vi ha apportato. Procedere in questo modo è sicuramente di notevole aiuto per l'apprendimento. Questo libro include degli esercizi che richiedono la modifica del codice sorgente di un sistema operativo. Alcuni algoritmi, inoltre, sono descritti ad alto livello, per essere sicuri di coprire tutti gli argomenti importanti che riguardano i sistemi operativi. Nel libro si possono inoltre trovare riferimenti ad esempi di codice open-source, per eventuali approfondimenti.

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

### 1.13.1 Storia

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

Successivamente le società informatiche cercarono di limitare l'utilizzo del loro software a computer autorizzati e clienti paganti. Riuscirono a raggiungere questo obiettivo rendendo disponibili solo i file in binario compilati a partire dal codice sorgente, ma non il codice stesso. Esse protessero così allo stesso tempo il proprio codice e le proprie idee dai concorrenti. Altra questione è quella che riguarda il materiale protetto da copyright. I sistemi operativi e altri programmi possono limitare la possibilità di accedere a film, musica e a libri

elettronici solo a computer autorizzati. Tale protezione o gestione dei diritti digitali (*digital rights management*, DRM) non sarebbe efficace se il codice sorgente che implementa questi limiti fosse pubblicato. Le leggi di molti paesi, incluso il Digital Millennium Copyright Act (DMCA) negli Stati Uniti, rendono illegale ricavare un codice DRM tramite il reverse-engineering o provare a eludere la protezione del materiale.

Per contrastare la limitazione nell'utilizzo e nella redistribuzione di software, nel 1983 Richard Stallman diede vita al progetto GNU con la finalità di creare un sistema operativo gratuito, open-source e compatibile con UNIX. Nel 1985 pubblicò il manifesto GNU, sostenendo che tutti i software dovrebbero essere gratuiti e open-source. Costituì inoltre la Free Software Foundation (FSF) con lo scopo di incoraggiare il libero scambio dei codici sorgente e il libero utilizzo del software. Anziché depositare il proprio software, la FSF distribuisce il software incoraggiandone la condivisione e il miglioramento. La General Public License della GNU (GPL) codifica questa distribuzione ed è un'autorizzazione pubblica per il rilascio di software. Fondamentalmente la GPL richiede che il codice sorgente sia distribuito assieme all'eseguibile binario e che qualsiasi cambiamento apportato a quel sorgente sia anch'esso reso disponibile con la stessa autorizzazione GPL.

### 1.13.2 Linux

Consideriamo GNU/Linux come esempio di sistema operativo open-source. Il progetto GNU produsse molti strumenti compatibili con UNIX, inclusi compilatori, editor, utilità, ma non ha mai distribuito un kernel. Nel 1991 uno studente finlandese, Linus Torvalds, rilasciò un kernel rudimentale simile a UNIX utilizzando compilatori e strumenti di GNU e invitò gli interessati a contribuire allo sviluppo. Con l'avvento di Internet, chiunque fosse interessato al progetto poteva scaricare il codice sorgente, modificarlo e sottoporre i cambiamenti a Torvalds. Il rilascio settimanale di aggiornamenti permise a questo sistema operativo, il cosiddetto Linux, di crescere rapidamente, avvalendosi delle migliorie apportate da migliaia di programmatore.

Il sistema operativo GNU/Linux che ne è risultato ha creato centinaia di singole distribuzioni del sistema, ovvero versioni personalizzate. Le principali distribuzioni includono Red Hat, SUSE, Fedora, Debian, Slackware e Ubuntu. Le distribuzioni differiscono nelle funzionalità, nelle applicazioni installate, nel supporto hardware, nell'interfaccia e negli obiettivi. Ad esempio, Red Hat Enterprise Linux è indirizzato al grande uso commerciale. PCLinuxOS è un LiveCD, un sistema operativo che può essere avviato ed eseguito da un CD-ROM senza essere installato sul disco fisso. Una variante di PCLinuxOS, "PCLinuxOS Supergamer DVD", è un LiveDVD che include driver per la grafica e giochi. Un giocatore può farlo funzionare su qualsiasi sistema compatibile semplicemente avviandolo dal DVD; al termine del gioco il riavvio del sistema ripristina il sistema operativo originario.

L'accesso al codice sorgente di Linux varia secondo la versione. Qui prendiamo in considerazione Ubuntu, una distribuzione di Linux disponibile in molte tipologie, comprese quelle destinate ai desktop, ai server o agli studenti. Il suo fondatore si fa carico delle spese per la stampa e la spedizione dei DVD con il codice binario e sorgente (il che contribuisce a diffondere la distribuzione). I seguenti passi permettono di esplorare il codice sorgente del kernel Ubuntu su sistemi che supportano lo strumento gratuito "WMware Player".

Scaricate il software di virtualizzazione (player) da:

- ◆ <http://www.wmware.com/download/player/> e installatelo sul vostro sistema.
- ◆ Scaricate una macchina virtuale contenente Ubuntu. Centinaia di immagini di macchine virtuali (*appliances*), preinstallate con sistemi operativi e applicazioni, sono disponibili su WMware all'indirizzo <http://www.wmware.com/appliances/>.

- ◆ Avviate la macchina virtuale in ambiente WMware player.
- ◆ Ottenete il codice sorgente della versione del kernel che vi interessa, ad esempio 2.6, eseguendo `wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.1.tar.bz2`.
- ◆ Decomprimete il file scaricato con `tar xjf linux-2.6.18.1.tar.bz2`.
- ◆ Esplorate il codice sorgente del kernel Ubuntu, che si trova ora in `./linux-2.6.18.1`.

Per ulteriori informazioni su Linux si veda il Capitolo 21. Per quanto riguarda le macchine virtuali, si veda il Paragrafo 2.8.

### 1.13.3 UNIX BSD

Rispetto a Linux UNIX BSD ha una storia più lunga e complicata. La sua creazione, derivata dallo UNIX di AT&T, risale al 1978 e le sue prime versioni vennero distribuite dalla Università della California a Berkeley (UCB) in codice sorgente e in formato binario, ma non erano open-source, perché era necessaria una licenza della AT&T. Lo sviluppo di UNIX BSD venne rallentato nei successivi anni da una querela della AT&T, ma alla fine una versione completa e open-source del sistema, la 4.4BSD-lite, venne rilasciata nel 1994.

Esattamente come nel caso di Linux, ci sono diverse distribuzioni di UNIX BSD, tra le quali FreeBSD, NetBSD, OpenBSD e DragonflyBSD. Per esaminare nel dettaglio il codice sorgente di FreeBSD è sufficiente scaricare l'immagine della versione desiderata e caricarla in VMWare, come descritto in precedenza per Linux Ubuntu. Il codice sorgente è allegato alla distribuzione e lo si può trovare in `/usr/src`. Il codice sorgente del kernel si trova in `/usr/src/sys`. Per esaminare, ad esempio, il codice che implementa la memoria virtuale nel kernel di FreeBSD, è sufficiente andare in `/usr/src/sys/vm`.

Darwin, il cuore del kernel di MAC, è basato su UNIX BSD ed è anch'esso open-source. Il sorgente è disponibile all'indirizzo <http://www.opensource.apple.com/darwinsource/>. Lo stesso sito contiene tutte le componenti open-source delle distribuzioni di MAC. Il nome del pacchetto contenente il kernel è "xnu". Il codice sorgente del kernel di MAC nella sua revisione 1228 (il kernel di Leopard MAC) si può trovare all'indirizzo <http://www.opensource.apple.com/darwinsource/tarballs/apsl/xnu-1228.tar.gz>. Apple fornisce anche diversi strumenti per sviluppatori, documentazione e supporto all'indirizzo <http://connect.apple.com>.

### 1.13.4 Solaris

Solaris è il sistema operativo commerciale basato su UNIX della Sun Microsystems. In origine il sistema operativo della Sun, il SunOS, si basava su UNIX BSD. A partire dal 1991 Sun iniziò a utilizzare come base del suo sistema operativo UNIX System V di AT&T. Nel 2005 Sun rese disponibile parte del sistema Solaris; col tempo la società ha continuato a fare aggiunte al codice open-source. Purtroppo Solaris non è completamente open-source perché parte del codice è ancora di proprietà della AT&T e di altre società. Tuttavia Solaris può essere compilato dal codice sorgente e collegato (tramite il linker) al codice binario delle componenti a sorgente chiuso. Anche Solaris, quindi, può essere esplorato, modificato, compilato e testato.

Il codice sorgente è disponibile all'indirizzo <http://opensolaris.org/os/downloads/>. Sono disponibili anche distribuzioni precompilate basate sul codice sorgente, nonché gruppi di discussione. Non è necessario scaricare l'intero pacchetto del codice sorgente dal sito, perché Sun permette ai visitatori di esplorare il codice sorgente online con un apposito browser.

### 1.13.5 Utility

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

GNU/Linux, UNIX BSD e Solaris sono tutti sistemi operativi open-source, ma ognuno ha obiettivi, funzionalità, licenze e scopi propri. A volte le licenze non si escludono reciprocamente e danno vita a una sorta di impollinazione incrociata che contribuisce a un rapido miglioramento dei progetti riguardanti i sistemi operativi. Ad esempio, diverse componenti di rilievo di Solaris sono state trasferite su UNIX BSD. I vantaggi del software gratuito e dell'open-source possono portare a un aumento del numero e della qualità dei progetti open-source, comportando un incremento del numero di individui e società che li utilizzano.

## 1.14 Sommario

Un sistema operativo è il software che gestisce l'hardware di un calcolatore, e fornisce un ambiente all'interno del quale siano eseguibili le applicazioni. Forse l'aspetto più concreto dei sistemi operativi è l'interfaccia d'accesso al computer che essi forniscono all'utente.

Per essere eseguiti, i programmi devono risiedere nella memoria centrale del calcolatore. Questa è infatti la sola area di memoria di grandi dimensioni direttamente accessibile dalla CPU. Essa è un vettore, di parole o byte, di dimensioni variabili tra i milioni e i miliardi di byte. Ciascuna parola possiede il proprio indirizzo. La memoria centrale è un dispositivo volatile, poiché perde il proprio contenuto quando manca l'alimentazione elettrica. La maggior parte dei sistemi di calcolo dispone di una memoria secondaria come estensione della memoria centrale. Il requisito fondamentale della memoria secondaria è la capacità di memorizzare in modo permanente grandi quantità di dati. Il più comune dispositivo di memoria secondaria è il disco magnetico; si tratta di un dispositivo di memoria non volatile che può memorizzare sia dati sia programmi e che consente l'accesso diretto ai suoi elementi.

I sistemi di memorizzazione di un calcolatore si possono organizzare in modo gerarchico secondo la velocità e il costo. I livelli più alti rappresentano i dispositivi più rapidi, ma più costosi. Scendendo nella gerarchia, il costo per bit generalmente decresce, mentre di solito aumentano i tempi d'accesso.

Alla base della progettazione di un sistema di elaborazione sono possibili approcci diversi. Una prima scelta deve essere operata tra i sistemi a processore unico e quelli con due o più processori; in quest'ultimo caso, la memoria fisica e i dispositivi periferici sono condivisi da tutti i processori. Lo schema più comune tra i sistemi multiprocessore è rappresentato dalla multielaborazione simmetrica (SMP), che vede tutti i processori su un piano di parità: essi elaborano in piena indipendenza gli uni dagli altri. Una forma specializzata di multielaborazione è invece rappresentata dai cluster di elaboratori (*clustered systems*), ovvero un certo numero di elaboratori connessi da una rete locale.

Per utilizzare al meglio la CPU, i sistemi operativi moderni impiegano la multiprogrammazione, grazie alla quale diversi processi possono occupare contemporaneamente la memoria, assicurando che la CPU non resti mai inattiva. Con i sistemi a tempo ripartito il concetto di multiprogrammazione è stato ulteriormente esteso, per mezzo di algoritmi di scheduling della CPU che fanno la spola tra un processo e l'altro, dando così l'impressione che molti processi siano in esecuzione allo stesso tempo.

Il sistema operativo deve assicurare il corretto funzionamento del calcolatore. Per evitare che i programmi utenti interferiscano tra di loro e col sistema operativo, la CPU ha due modalità di funzionamento: la modalità utente e la modalità di sistema. Diverse istruzioni (come quelle di I/O e di arresto del calcolatore) sono privilegiate e si possono eseguire solamente in modalità di sistema. Anche l'area della memoria in cui risiede il sistema operativo deve essere protetta dai tentativi di modifiche da parte degli utenti. La presenza di un timer evita il verificarsi di cicli infiniti. Queste funzioni (duplice modalità di funzionamento, istruzioni privilegiate, protezione della memoria, interruzioni del timer) costituiscono gli elementi fondamentali impiegati dal sistema operativo per ottenere un corretto funzionamento del sistema.

Un processo (*process* o *job*) è l'unità fondamentale di lavoro in un sistema operativo. La gestione dei processi comprende aspetti come la loro creazione e cancellazione, nonché la messa a punto di meccanismi per la comunicazione reciproca e la sincronizzazione dei processi. Un sistema operativo, nel gestire la memoria, si cura di registrare quali parti di essa vengano usate e da chi. È sempre al sistema operativo, inoltre, che spetta l'allocazione dinamica e il rilascio dello spazio di memoria. Esso gestisce anche l'archiviazione dei dati: dai file system per i file e le directory, ai dispositivi per la memorizzazione di massa.

Altre due indispensabili funzioni dei sistemi operativi sono le strategie di protezione e le politiche per la sicurezza del sistema. La protezione è garantita da una serie di meccanismi per il controllo dell'accesso, da parte dei processi o degli utenti, alle risorse che il sistema mette a disposizione. Alle misure di sicurezza è invece affidata la difesa dell'elaboratore da attacchi esterni o interni.

I sistemi distribuiti conferiscono agli utenti la possibilità di condividere risorse che giacciono a grande distanza le une dalle altre, su una rete di terminali connessi. I servizi possono essere offerti sia con il modello client-server che con il modello peer-to-peer (ossia, da pari a pari). In un cluster, poiché i dati risiedono in una memoria condivisa, l'elaborazione può essere compiuta da più macchine; di conseguenza, anche se alcuni membri del cluster dovessero subire dei guasti, l'elaborazione può essere portata avanti dagli altri.

I due tipi fondamentali di reti sono le LAN e le WAN. Le prime consentono l'interazione tra processori collocati in un raggio geografico ristretto, mentre le seconde mettono in comunicazione processori separati da distanze più grandi. Le LAN sono generalmente più veloci delle WAN.

Vi sono diversi sistemi di calcolo che perseguono scopi specifici. Ne fanno parte i sistemi operativi real-time destinati ad ambienti integrati quali i prodotti commerciali, le automobili e la robotica. I sistemi operativi real-time sono vincolati da scadenze temporali ben definite, che non tollerano ritardi. L'elaborazione deve essere completata entro i limiti prestabiliti, pena il fallimento del sistema. I sistemi multimediali riguardano la trasmissione di dati multimediali, e hanno spesso requisiti particolari per la visualizzazione o la riproduzione di filmati e tracce audio, o casi in cui audio e video sono sincronizzati.

In tempi recenti, l'influenza di Internet e del Web ha incoraggiato lo sviluppo di sistemi operativi moderni che includono come parti integranti elementi quali i browser web, il software per l'accesso alla rete e i programmi per la comunicazione in rete.

Il movimento a sostegno del software gratuito ha creato migliaia di progetti open-source, inclusi sistemi operativi. Grazie a questi progetti gli studenti hanno la possibilità di utilizzare il codice sorgente come strumento di apprendimento. Possono infatti modificare i programmi e testarli, individuare bachi ed eliminarli, e al tempo stesso esplorare sistemi operativi maturi e completamente definiti, compilatori, strumenti, interfacce e altri tipi di programmi.

GNU/Linux, UNIX BSD e Solaris sono tutti sistemi operativi open-source, ma ognuno ha obiettivi, utilità, licenze e scopi propri. A volte le licenze non si escludono reciprocamente e danno vita a una sorta di impollinazione incrociata che contribuisce a un rapido miglioramento dei progetti riguardanti i sistemi operativi. Ad esempio, diverse componenti di rilievo di Solaris sono state trasferite su UNIX BSD. I vantaggi del software gratuito e dell'open-source possono portare a un aumento del numero e della qualità dei progetti open-source, comportando un incremento del numero di individui e società che li utilizzano.

## Esercizi pratici

- 1.1 Quali sono i tre scopi principali di un sistema operativo?
- 1.2 Quali sono le differenze principali tra i sistemi operativi per mainframe e quelli per personal computer?
- 1.3 Elicate le quattro fasi necessarie per eseguire un programma su una macchina completamente dedicata, ovvero un computer sul quale viene eseguito solo quel programma.
- 1.4 Abbiamo sottolineato come il sistema operativo sia volto all'uso efficiente dell'hardware. Quando è opportuno che il sistema operativo rinunci a questo principio e "sprechi" risorse? Perché un sistema simile non può essere considerato davvero inefficiente?
- 1.5 Qual è la difficoltà principale che deve superare un programmatore nello scrivere un sistema operativo per un ambiente real-time?
- 1.6 Considerate le varie definizioni di sistema operativo. Valutate se sia opportuno che il sistema operativo includa o meno applicazioni quali browser e programmi di posta elettronica. Argomentate entrambe le possibilità, fornendo delle motivazioni.
- 1.7 Come funziona la distinzione tra modalità di sistema (modalità kernel) e modalità utente quale rudimentale forma di protezione (sicurezza) del sistema?
- 1.8 Quale delle seguenti istruzioni dovrebbe essere privilegiata?
  - a. Impostare il timer.
  - b. Leggere il clock.
  - c. Cancellare la memoria.
  - d. Invocare un'istruzione trap.
  - e. Disattivare le interruzioni.
  - f. Modificare le informazioni nella tabella che indica lo status dei dispositivi.
  - g. Passare da modalità di sistema a modalità utente.
  - h. Accedere a un dispositivo I/O.

- 1.9 Alcuni dei primi computer proteggevano il sistema operativo posizionandolo in una partizione della memoria che non poteva essere modificata né dall'utente né dal sistema operativo. Descrivete due difficoltà che secondo voi potrebbero sorgere da uno schema simile.
- 1.10 Alcune CPU offrono più di due modalità di operazione. Quali sono due possibili impieghi di queste modalità multiple?
- 1.11 I timer potrebbero essere utilizzati anche per calcolare l'ora corrente. Spiegate brevemente come.
- 1.12 Internet è una rete LAN o WAN?

## Esercizi

- 1.13 In un ambiente multiprogrammato e a tempo ripartito, diversi utenti condividono il sistema simultaneamente. Tale situazione può generare alcuni problemi di sicurezza.
  - a. Individuate due possibili problemi.
  - b. Si può garantire il medesimo grado di sicurezza in una macchina a tempo ripartito e in una macchina dedicata? Motivate la risposta.
- 1.14 La tematica dell'utilizzo delle risorse è una costante dei sistemi operativi, seppure in modi diversi a seconda del tipo di sistema considerato. Si dettaglino le risorse da gestire con cura nelle seguenti situazioni:
  - a. mainframe o minicomputer;
  - b. stazioni di lavoro connesse a server;
  - c. computer palmari.
- 1.15 Quando e perché sarebbe più conveniente per un utente scegliere un sistema a tempo ripartito anziché un PC o una stazione di lavoro individuale?
- 1.16 Quale tra le funzionalità citate in basso deve possedere un sistema operativo in queste situazioni: (a) dispositivi palmari e (b) sistemi real-time.
  - a. Programmazione a lotti.
  - b. Memoria virtuale.
  - c. Tempo ripartito.
- 1.17 Descrivete le differenze tra multielaborazione simmetrica e asimmetrica. Elencate tre vantaggi e uno svantaggio dei sistemi multiprocessore.
- 1.18 Quali differenze presentano i cluster di elaboratori rispetto ai sistemi per la multielaborazione? Che cosa è necessario perché due macchine appartenenti a un cluster, con il loro contributo congiunto, offrano un servizio altamente affidabile?
- 1.19 Tratteggiate le differenze tra il modello client-server e quello peer-to-peer.
- 1.20 Ipotizziamo che sui due nodi di un cluster di elaboratori sia attiva una base di dati. Descrivete due modalità con cui il software per la gestione del cluster può regolare l'accesso ai dati sul disco, analizzando i pro e i contro di ognuna.
- 1.21 In che cosa i computer di rete si differenziano dagli elaboratori tradizionali? Enumerate qualche caso concreto in cui sia preferibile utilizzare un computer di rete.

- 1.22 Qual è lo scopo delle interruzioni? Quali differenze vi sono tra un'eccezione e un'interruzione? Un programma utente può generare un'eccezione di proposito? In caso affermativo, con quale scopo?
- 1.23 L'accesso diretto alla memoria (DMA) è usato per dispositivi I/O ad alta velocità per evitare di sovraccaricare la CPU.
- a. In che modo la CPU si interfaccia con il dispositivo per coordinare il trasferimento?
  - b. In che modo la CPU apprende che il trasferimento in memoria è completo?
  - c. La CPU è abilitata all'esecuzione di altri programmi mentre il controllore DMA procede al trasferimento dei dati. Può tale trasferimento interferire con la corretta esecuzione dei programmi utenti? Se la risposta è positiva, descrivete in quale forma può sorgere l'interferenza.
- 1.24 L'architettura di alcuni sistemi di calcolo non possiede una modalità di funzionamento riservata al sistema operativo. Spiegate se per questi calcolatori è possibile realizzare sistemi operativi sicuri.
- 1.25 Fornite due motivi che dimostrino l'utilità delle cache, quali problemi risolvono e quali provocano. Potendo disporre di una cache delle dimensioni del dispositivo cui è associata (per esempio una cache delle dimensioni di un disco) spiegate se sia possibile usarla eliminando il dispositivo.
- 1.26 Considerate un sistema SMP simile a quello della Figura 1.6. Illustrate con un esempio come i dati presenti nella memoria potrebbero avere due valori differenti in ognuna delle cache locali.
- 1.27 Illustrate, con l'ausilio di esempi, come si manifesta il problema della coerenza dei dati memorizzati nella cache nei seguenti ambienti di elaborazione:
- a. sistemi a processore unico;
  - b. sistemi multiprocessore;
  - c. sistemi distribuiti.
- 1.28 Descrivete un meccanismo di protezione della memoria grazie al quale sia possibile impedire a un programma la modifica della memoria di pertinenza di altri programmi.
- 1.29 Dite quale configurazione di rete sia più adatta ai seguenti ambienti:
- a. un piano di un pensionato universitario;
  - b. un campus universitario;
  - c. una regione;
  - d. una nazione.
- 1.30 Si definiscano le caratteristiche tipiche appartenenti a ciascun sistema operativo:
- a. a lotti;
  - b. interattivo;
  - c. a tempo ripartito;
  - d. in tempo reale;
  - e. di rete;
  - f. parallelo;

- g. distribuito;
- h. cluster;
- i. palmare.

- 1.31 Quali sono i vantaggi e gli svantaggi offerti dai computer palmari?
- 1.32 Identificate vantaggi e svantaggi dei sistemi operativi open-source. Discutete anche le tipologie di persone che potrebbero definire determinati aspetti come vantaggi oppure svantaggi.

## 1.15 Note bibliografiche

[Brooks 2003] fornisce una panoramica a grandi linee sull'informatica in generale. [Bovet e Cesati 2002] presentano la struttura di base del sistema operativo Linux. L'opera di [Solomon e Russinovich 2000], oltre a dare un'idea generale su Windows, è ricca di dettagli tecnici sui componenti del sistema. Russinovich e Salomon [2005] hanno aggiornato questa informazione fino a Windows Server 2003 e Windows XP. McDougall e Mauro [2007] trattano dei componenti del sistema operativo Solaris. Mac OS X è presente su <http://www.apple.com/macosx>. I componenti di Mac OS X vengono trattati in Singh [2007].

Tra quanti hanno esplorato i sistemi peer-to-peer citiamo [Parameswaran et al. 2001], [Gong 2002], [Ripeanu et al. 2002], [Agre 2003], [Balakrishnan et al. 2003] e [Loo 2003]. In [Lee 2003] è reperibile una discussione sui sistemi peer-to-peer per la condivisione dei file. Un'opera di riferimento per i cluster di elaboratori è quella di [Buyya 1999]. Sviluppi più recenti dello stesso argomento sono descritti da [Ahmed 2000]. Una rassegna di questioni relative ai sistemi distribuiti è fornita da [Tanenbaum e Van Renesse 1985].

I sistemi operativi trovano spazio in numerosi libri di testo generali, tra cui [Stallings 2000b], [Nutt 2004] e [Tanenbaum 2001].

[Hamacher et al. 2002] descrivono l'organizzazione degli elaboratori. McDougall e Laudon [2006] discutono dei processori multicore. Hennessy e Patterson [2007] trattano dei sistemi di I/O, dei canali di comunicazione e delle architetture di sistema in genere. Blaauw e Brooks [1997] descrivono dettagliatamente l'architettura di molti sistemi di elaborazione, compresi molti sistemi IBM. Stokes [2007] fornisce un'introduzione illustrata ai microprocessori e all'architettura dei calcolatori.

La memoria cache, compresa la memoria associativa, è analizzata in [Smith 1982], un articolo che offre una vasta bibliografia sull'argomento.

Alcuni approfondimenti riguardanti la tecnologia dei dischi magnetici si trovano in [Freedman 1983] e [Harker et al. 1981], mentre i dischi ottici sono trattati da [Kenville 1982], [Fujitani 1984], [O'Leary e Kitts 1985], [Gait 1988] e [Olsen e Kenley 1989]. Sui floppy disk si soffermano [Pechura e Schoeffler 1983] e anche [Sarisky 1983]. Considerazioni generali sulle tecnologie per l'archiviazione di massa dei dati sono offerte da [Chi 1982] e [Hoagland 1985].

[Kurose e Ross 2005] e [Tanenbaum 2003] offrono una trattazione generale delle reti di calcolatori. [Fortier 1989] presenta un dettagliato approfondimento del software e dell'hardware impiegato nelle reti. Kozierok [2005] discute il TCP in dettaglio. Mullender [1993] fa una panoramica dei sistemi distribuiti. [Wolf 2003] espone gli sviluppi recenti relativi ai sistemi integrati. Questioni legate ai dispositivi palmari sono illustrate da [Myers e Beigl 2003], [Di Pietro e Mancini 2003].

Una trattazione completa della storia dell'open-source e dei suoi vantaggi e sfide si può trovare in Raymond [1999]. La storia della pirateria informatica (*hacking*) è discussa da Levy [1994]. La Free Software Foundation ha pubblicato il suo manifesto all'indirizzo <http://gnu.org/philosophy/free>.

software-for-freedom.html. Istruzioni dettagliate su come compilare il kernel Linux Ubuntu si trovano su: [http://howtoforge.com/kernel\\_compilation\\_ubuntu](http://howtoforge.com/kernel_compilation_ubuntu). I componenti open-source di MAC sono disponibili su <http://developer.apple.com/open-source/index.html>.

Wikipedia ([http://en.wikipedia.org/wiki/richard\\_stallman](http://en.wikipedia.org/wiki/richard_stallman)) contiene una voce su Richard Stallman.

Il codice sorgente di Multics è disponibile su [http://web.mit.edu/multics-history/source/multics\\_internet\\_server/multics\\_sources.html](http://web.mit.edu/multics-history/source/multics_internet_server/multics_sources.html).

## Capitolo 2

# Strutture dei sistemi operativi



### OBIETTIVI

- Descrizione dei servizi messi a disposizione dal sistema operativo a utenti, processi e altri sistemi.
- Esame delle possibili strutture dei sistemi operativi.
- Installazione e adattamento dei sistemi operativi; descrizione delle operazioni da eseguire all'avvio.

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

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

## 2.1 Servizi di un sistema operativo

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

Ogni insieme di servizi offre funzionalità utili all'utente.

- ◆ **Interfaccia con l'utente.** Quasi tutti i sistemi operativi hanno un'interfaccia con l'utente (UI). Essa può assumere diverse forme. Un'interfaccia a riga di comando (CLI) è basata su stringhe che codificano i comandi, insieme a un metodo per inserirli e modificarli come ad esempio un programma apposito. Un'interfaccia a lotti, invece, prevede che comandi e relative direttive siano codificati nei file, eseguiti successivamente

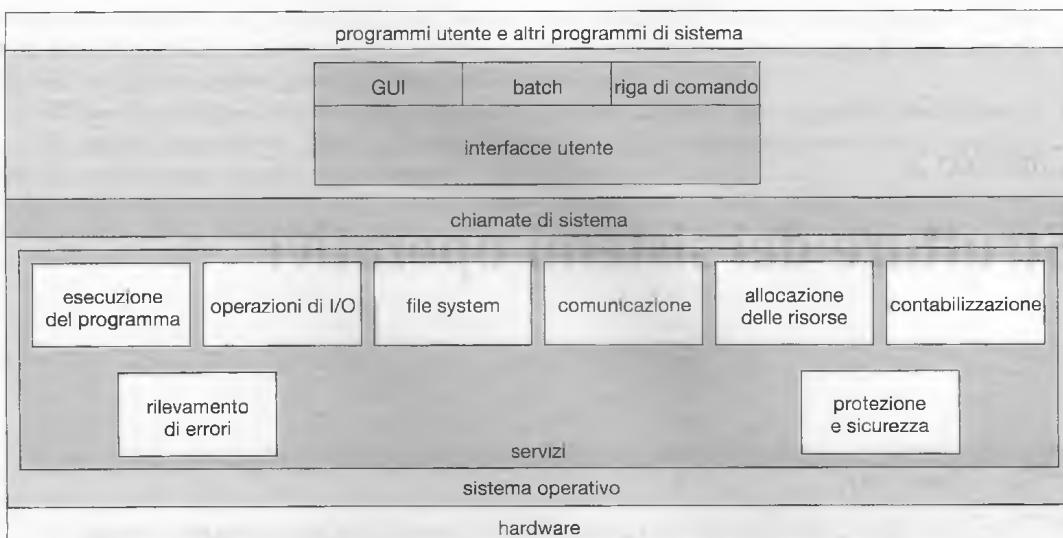


Figura 2.1 Panoramica dei servizi del sistema operativo.

a lotti. La forma senz'altro più diffusa è l'**interfaccia grafica con l'utente** (GUI), ossia un sistema grafico a finestre dotato di un dispositivo puntatore (per esempio, il mouse) per comandare operazioni di I/O e selezionare opzioni dai menu, insieme a una tastiera per inserire del testo. Certi sistemi offrono alcune o anche tutte queste soluzioni.

- ◆ **Esecuzione di un programma.** Il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anormale (indicando l'errore).
- ◆ **Operazioni di I/O.** Un programma in esecuzione può richiedere un'operazione di I/O che implica l'uso di un file o di un dispositivo di I/O. Per particolari dispositivi possono essere necessarie funzioni speciali, come il riavvolgimento di un'unità a nastro, oppure la cancellazione dello schermo di un tubo a raggi catodici (CRT). Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di I/O, quindi il sistema operativo deve offrire mezzi adeguati.
- ◆ **Gestione del file system.** Il file system riveste un interesse particolare. I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file con un certo nome, e disporre di informazioni relative al file stesso. Alcuni programmi, infine, devono poter gestire i permessi di accesso ai file sulla base della proprietà del file interessato. Molti sistemi operativi offrono all'utente la scelta di file system diversi con funzionalità e prestazioni specifiche.
- ◆ **Comunicazioni.** In molti casi un processo ha bisogno di scambiare informazioni con un altro processo. Ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una memoria condivisa o attraverso lo **scambio di messaggi**, in questo caso il sistema operativo trasferisce pacchetti d'informazioni tra i vari processi.

- ◆ **Rilevamento d'errori.** Il sistema operativo deve essere sempre capace di rilevare eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria, quali un errore di memoria o un guasto all'alimentazione elettrica; nei dispositivi di I/O, come un errore di parità in un nastro, il guasto di una connessione di rete, la mancanza di carta nella stampante; in un programma utente, come una divisione per zero, un tentativo d'accesso a una locazione di memoria illegale, un uso eccessivo del tempo di CPU. Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore. Naturalmente sistemi operativi differenti reagiscono agli errori e vi pongono riparo in modi diversi. Eventuali funzionalità di debug – cioè, degli strumenti che permettano l'analisi, per esempio, del software che ha causato un errore – aumentano di molto le possibilità dei programmati e degli utenti di usare il sistema efficientemente.

Esiste anche un'altra serie di funzioni del sistema operativo che non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso. Sistemi con più utenti possono guadagnare in efficienza condividendo le risorse del calcolatore tra i diversi utenti.

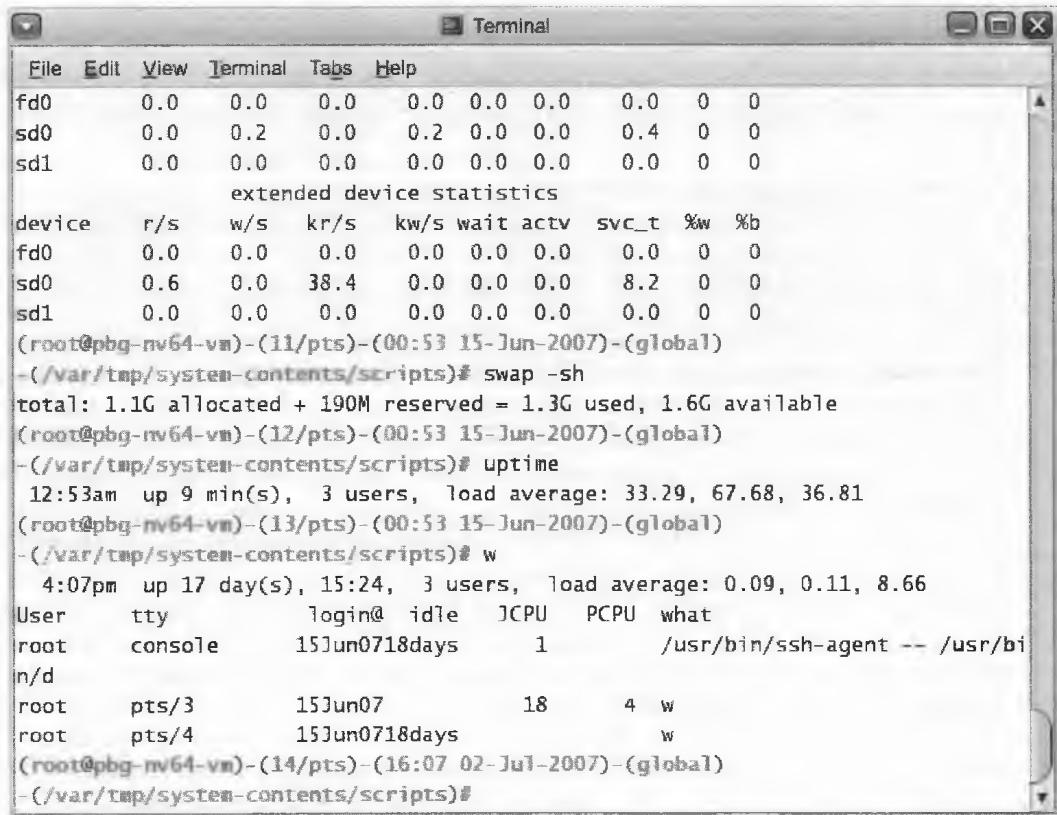
- ◆ **Assegnazione delle risorse.** Se sono in corso più sessioni di lavoro di utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di CPU, la memoria centrale e la registrazione in file, possono avere un codice di assegnazione speciale, mentre altre, come i dispositivi di I/O, possono avere un codice di richiesta e di rilascio più generale. Per esempio, per determinare come utilizzare al meglio la CPU, i sistemi operativi impiegano le procedure di scheduling della CPU, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili e di altri fattori. Esistono anche procedure per l'assegnazione di risorse a uso di un processo, quali stampanti, modem, driver di memorizzazione USB e altri dispositivi periferici.
- ◆ **Contabilizzazione dell'uso delle risorse.** È possibile registrare quali utenti usino il calcolatore, segnalando quali e quante risorse impieghino. Questo tipo di registrazione si può usare per contabilizzare l'uso delle risorse, per addebitare il costo agli utenti, oppure per redigere statistiche; queste ultime possono essere un valido strumento per i ricercatori che desiderano riconfigurare il sistema per migliorarne i servizi di calcolo.
- ◆ **Protezione e sicurezza.** I proprietari di informazioni memorizzate in un sistema di calcolo multiutente o in rete possono voler controllare l'uso di tali informazioni. Più processi non correlati e in esecuzione concorrente non devono influenzarsi o interferire con il sistema operativo. La protezione assicura che l'accesso alle risorse del sistema sia controllato. È importante anche proteggere il sistema dagli estranei. La sicurezza di un sistema comincia con l'obbligo d'identificazione da parte di ciascun utente, di solito attraverso parole d'ordine che permettono l'accesso alle risorse; si estende alla 'difesa' dei dispositivi di I/O (compresi i modem e gli adattatori di rete) dai tentativi d'accesso illegali e provvede al loro rilevamento. Se un sistema deve essere protetto e sicuro, al suo interno devono esistere precauzioni ovunque. La forza di una catena è esattamente quella del suo anello più debole.

## 2.2 Interfaccia con l'utente del sistema operativo

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

### 2.2.1 Interprete dei comandi

Talvolta l'interprete dei comandi è una funzionalità compresa nel kernel dei sistemi operativi. In altri ambienti, come Windows XP e UNIX, l'interprete dei comandi è considerato un programma speciale, che si innesca all'avvio di un processo o allorché un utente si collega per la prima volta (nel caso di sistemi interattivi). Quando i sistemi consentono la scelta tra molteplici interpreti dei comandi, questi vengono definiti **shell**. In UNIX e Linux, per esempio, l'utente può scegliere tra svariate shell differenti, come la *Bourne*, la *C*, la *Bourne-again*, la *Korn*, e così via. Sono anche disponibili shell di terze parti e shell gratuite scritte dagli utenti. Nella maggior parte dei casi, le shell forniscono funzionalità simili e la scelta di un utente è solitamente dovuta alle preferenze personali. La Figura 2.2 illustra la shell Bourne, l'interprete dei comandi utilizzato da Solaris 10.



```

Terminal
File Edit View Terminal Tabs Help
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0   0
sd0      0.0    0.2    0.0    0.2    0.0    0.0    0.4    0   0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0   0
          extended device statistics
device   r/s    w/s    kr/s   kw/s  wait  activ  svc_t %w %b
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0   0
sd0      0.6    0.0   38.4    0.0    0.0    0.0    8.2    0   0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0   0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)#
w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User     tty           login@  idle   JCPU   PCPU  what
root    console        15Jun0718days    1      /usr/bin/ssh-agent -- /usr/bi
n/d
root    pts/3          15Jun07              18      4   w
root    pts/4          15Jun0718days                  w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#

```

Figura 2.2 Shell Bourne, l'interprete dei comandi utilizzato da Solaris 10.

La funzione principale dell'interprete dei comandi consiste nel prelevare ed eseguire il successivo comando impartito dall'utente. A questo livello, si usano molti comandi per la gestione dei file, vale a dire per creazione, cancellazione, elenchi, stampe, copie, esecuzioni di file, e così via. Le shell dell' MS-DOS e di UNIX funzionano in questo modo.

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

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

```
rm file.txt
```

cerca un file chiamato `rm`, lo carica in memoria e lo esegue con il parametro `file.txt`. La funzione corrispondente al comando `rm` è interamente definita dal codice del file `rm`. In questo modo i programmatori possono aggiungere nuovi comandi al sistema, semplicemente creando nuovi file con il nome appropriato. Il programma dell'interprete dei comandi, che può quindi essere abbastanza piccolo, non necessita di alcuna modifica quando s'introducono nuovi comandi.

## 2.2.2 Interfaccia grafica con l'utente

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

Le interfacce grafiche con l'utente si affacciarono sulla scena, da principio, per effetto delle ricerche condotte a Palo Alto nei primi anni '70 dai laboratori di ricerca Xerox PARC. La prima GUI apparve sul computer Xerox Alto nel 1973. Tuttavia, una diffusione più consistente delle interfacce grafiche si ebbe con l'avvento dei computer Apple Macintosh negli anni '80. L'interfaccia utente con il sistema operativo Macintosh (Mac OS) ha subito, nel corso degli anni, diverse modifiche, la più significativa delle quali è stata l'adozione dell'interfaccia *Aqua* per il Mac OS X. La prima versione del Windows di Microsoft, cioè la 1.0, era costruita su di un'interfaccia GUI al sistema operativo MS-DOS. I vari sistemi Windows che si rifacevano a questa versione iniziale hanno apportato ritocchi cosmetici all'aspetto della GUI e una serie di miglioramenti sul piano della funzionalità, compresa l'adozione di Windows Explorer.

Nei sistemi UNIX, tradizionalmente, le interfacce a riga di comando hanno avuto un ruolo preponderante, quantunque vi sia disponibilità di alcune interfacce GUI, come il CDE (*common desktop environment*, ambiente da scrivania comune) e i sistemi X-Windows, che so-

no diffusi fra le versioni commerciali di UNIX quali Solaris e il sistema AIX di IBM. Nella creazione di interfacce, tuttavia, un impulso determinante è giunto da vari progetti **open-source** come il KDE (*K desktop environment*, ambiente da scrivania K) e la scrivania GNOME del progetto GNU. Ambedue le scrivanie, KDE e GNOME, sono compatibili con Linux e con vari sistemi UNIX, e sono regolate da licenza open-source, vale a dire che il loro codice sorgente è reso disponibile per consultazioni e per modifiche soggette a specifiche condizioni di licenza.

La scelta di un'interfaccia GUI piuttosto che di una testuale dipende in buona misura dalle preferenze personali. In linea di massima, gli utenti di UNIX optano per le interfacce a riga di comando, dato che esse offrono shell dotate di caratteristiche potenti. Molti utenti Windows, d'altro canto, sono soddisfatti dell'ambiente Windows GUI, e per questo motivo non usano quasi mai la shell dell'interfaccia MS-DOS. I sistemi operativi Macintosh, con i molti cambiamenti attraversati, costituiscono un utile caso di studio da confrontare alla situazione appena descritta per UNIX e Windows. Fino a tempi recenti, il Mac OS non disponeva di un'interfaccia a riga di comando, e vincolava l'interazione degli utenti con il sistema alla propria interfaccia GUI. Tuttavia, con l'introduzione del Mac OS X (realizzato, in parte, sfruttando il kernel UNIX), il sistema operativo contiene ora sia la nuova interfaccia grafica Aqua sia un'interfaccia testuale a riga di comando. La Figura 2.3 mostra una schermata dell'interfaccia grafica di Mac OS X.

L'interfaccia con l'utente può cambiare da sistema a sistema e persino da utente a utente all'interno dello stesso sistema; in genere è ben distinta dalla struttura portante del sistema. La progettazione di un'interfaccia utile e intuitiva per l'utente non è, pertanto, intrinseca-



**Figura 2.3** Interfaccia grafica di Mac OS X.

mente legata al sistema operativo. In questo libro vengono evidenziati i problemi correlati alla prestazione di un servizio adeguato ai programmi utenti: dal punto di vista del sistema operativo non si applicherà alcuna distinzione tra programmi utenti e programmi del sistema.

## 2.3 Chiamate di sistema

Le **chiamate di sistema** costituiscono l'interfaccia tra un processo e il sistema operativo. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in C o C++, sebbene per alcuni compiti di basso livello, come quelli che comportano un accesso diretto all'hardware, sarebbe necessario il ricorso a istruzioni in linguaggio assembly.

Prima di illustrare come le chiamate di sistema vengano rese disponibili da parte del sistema operativo, consideriamo come esempio la scrittura di un semplice programma che legga i dati da un file e li trascriva in un altro. La prima informazione di cui il programma necessita è costituita dai nomi dei due file: il file in ingresso e il file in uscita. Questi file si possono indicare in molti modi diversi, secondo la struttura del sistema operativo. Un primo metodo consiste nel richiedere i nomi dei due file all'utente del programma. In un sistema interattivo questa operazione necessita di una sequenza di chiamate di sistema, innanzitutto per scrivere un messaggio di richiesta sullo schermo e quindi per leggere dalla tastiera i caratteri che compongono i nomi dei due file. Nei sistemi basati su mouse e finestre in genere appare in una finestra un menu contenente i nomi dei file. L'utente può usare il mouse per scegliere il nome del file di origine, dopodiché è possibile aprire una finestra simile alla precedente in cui specificare il nome del file di destinazione. Come vedremo, questa sequenza richiede molte chiamate di sistema.

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

Una volta predisposti i due file, si entra in un ciclo che legge dal file in ingresso (una chiamata di sistema) e scrive nel file di destinazione (altra chiamata di sistema). Ciascuna lettura (`read`) e ogni scrittura (`write`) deve riportare informazioni di stato relative alle possibili condizioni d'errore. Nel file in ingresso il programma può rilevare che è stata raggiunta la fine del file, oppure che nella lettura si è riscontrato un errore del dispositivo, per esempio un errore di parità. Nella fase di scrittura si possono verificare vari errori, la cui natura dipende dal dispositivo impiegato. Esempi tipici sono l'esaurimento dello spazio nei dischi, mancanza della carta in una stampante, e così via.

Infine, una volta copiato tutto il file, il programma può chiuderli entrambi (altre chiamate di sistema), inviare un messaggio alla console (più chiamate di sistema) e infine termi-

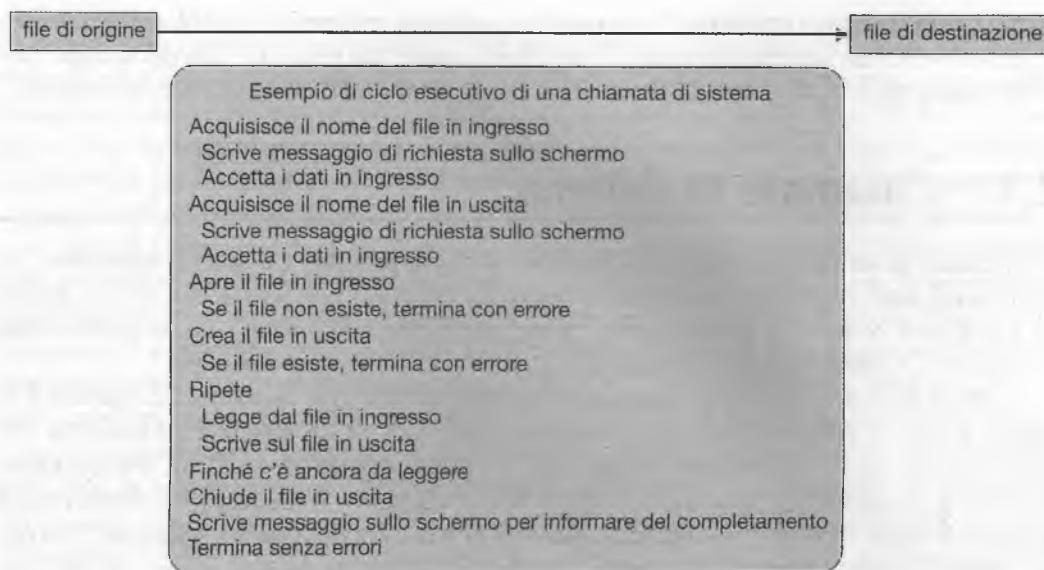


Figura 2.4 Esempio d'uso delle chiamate di sistema.

nare normalmente (ultima chiamata di sistema). Come s'è visto, anche programmi molto semplici possono fare un intenso uso del sistema operativo. Non è raro che un sistema esegua migliaia di chiamate di sistema al secondo. Il ciclo di esecuzione di una chiamata di sistema è illustrato nella Figura 2.4.

La maggior parte dei programmati, tuttavia, non si dovrà mai preoccupare di questi dettagli: infatti, gli sviluppatori usano in genere un'**interfaccia per la programmazione di applicazioni** (API, *application programming interface*). Essa specifica un insieme di funzioni a disposizione dei programmati, e dettaglia i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti. Tre delle interfacce più diffuse sono la API Win32 per i sistemi Windows, la API POSIX per i sistemi basati sullo standard POSIX (il che include essenzialmente tutte le versioni di UNIX, Linux e Mac OS X), e la API Java per la progettazione di applicazioni eseguite dalla macchina virtuale Java.

Si noti che (se non diversamente specificato) i nomi delle chiamate di sistema che ri-corrono in questo libro sono esempi generici; un dato sistema operativo adotterà nomi suoi propri per funzioni analoghe.

Dietro le quinte, le funzioni fornite da un API invocano solitamente le chiamate di sistema per conto del programmatore. La funzione Win32 `CreateProcess()`, per esempio, che serve a generare un nuovo processo, invoca in effetti `NTCreateProcess()`, una chiamata di sistema del kernel di Windows. Ci sono molte ragioni per cui è preferibile, per un programmatore, sfruttare l'intermediazione della API piuttosto che invocare direttamente le chiamate di sistema. Una di loro è legata alla portabilità delle applicazioni: a grandi linee, un programma sviluppato sulla base di una certa API girerà su qualunque sistema che la metta a disposizione, anche se le differenze architettoniche possono rendere la transizione non del tutto indolore. Inoltre, le chiamate di sistema sono spesso più dettagliate e difficili da usare dell'interfaccia API. Bisogna però dire che vi è spesso una stretta correlazione tra le funzioni di una API e le associate chiamate di sistema all'interno del kernel. In effetti, molte funzioni delle API POSIX e WIN32 sono simili alle chiamate di sistema fornite dai sistemi operativi UNIX, Linux e Windows.

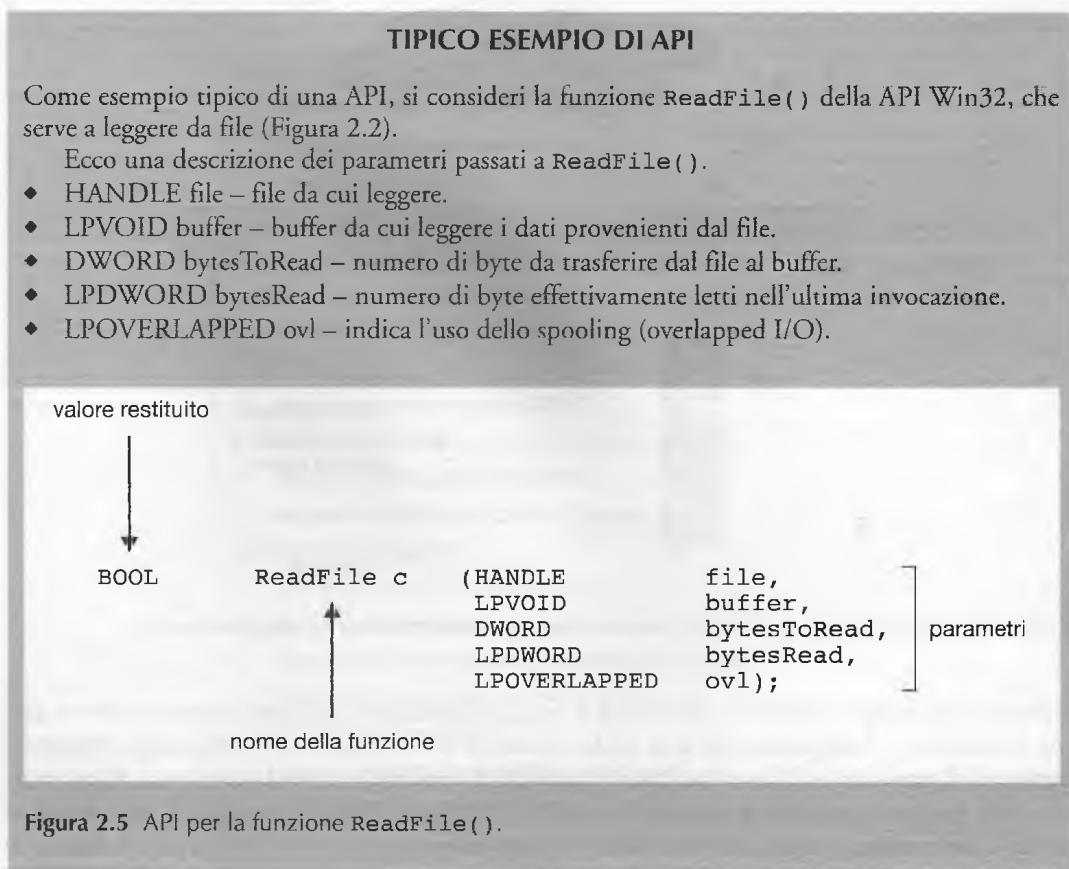
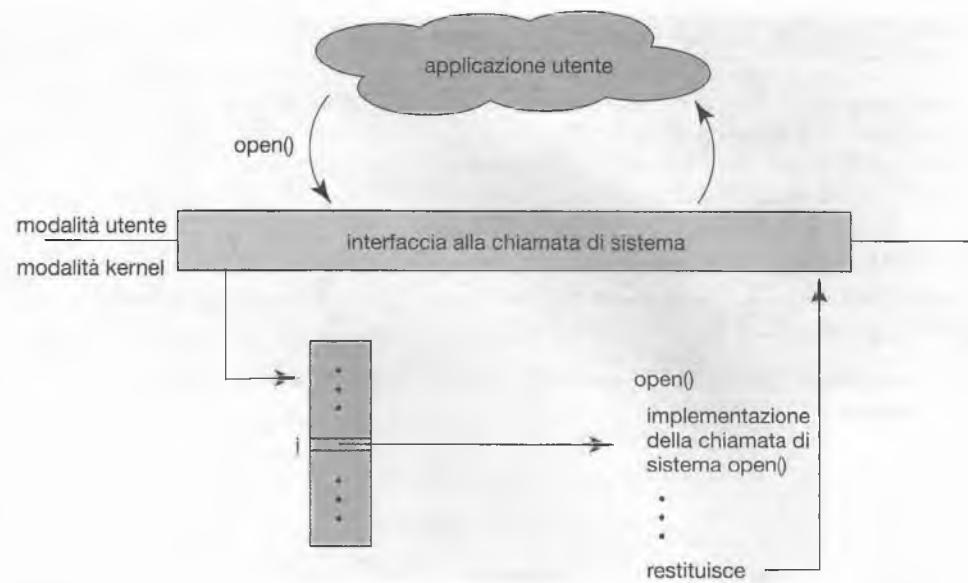


Figura 2.5 API per la funzione `ReadFile()`.

Il cosiddetto sistema di supporto all'esecuzione (*run-time support system*) di un linguaggio di programmazione, ossia l'insieme di funzioni strutturate in librerie incluse nel compilatore, fornisce nella gran parte dei casi un'interfaccia alle chiamate di sistema rese disponibili dal sistema operativo, che funge da raccordo tra il linguaggio e il sistema stesso. L'interfaccia intercetta le chiamate di sistema invocate dalla API, e richiede effettivamente la chiamata necessaria. Di solito, ogni chiamata di sistema è codificata da un numero; il compilatore mantiene una tabella delle chiamate di sistema, cui si accede usando questi numeri come indici. L'interfaccia alle chiamate di sistema invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, e passa al chiamante i valori restituiti dalla chiamata di sistema, inclusi quelli di stato.

Il chiamante non ha alcuna necessità di conoscere alcunché sull'implementazione della chiamata di sistema o del suo ciclo esecutivo: gli è sufficiente riconoscere la API e il risultato dell'esecuzione della chiamata di sistema operativo. Ne consegue che la gran parte dei dettagli relativi alle chiamate di sistema sono nascosti al programmatore dalla API, e gestiti dal sistema di supporto all'esecuzione. Le relazioni fra la API, l'interfaccia alle chiamate di sistema e il sistema operativo sono illustrati nella Figura 2.6, ove si mostra come il sistema operativo tratti l'invocazione della chiamata di sistema `open()` da parte di un'applicazione.

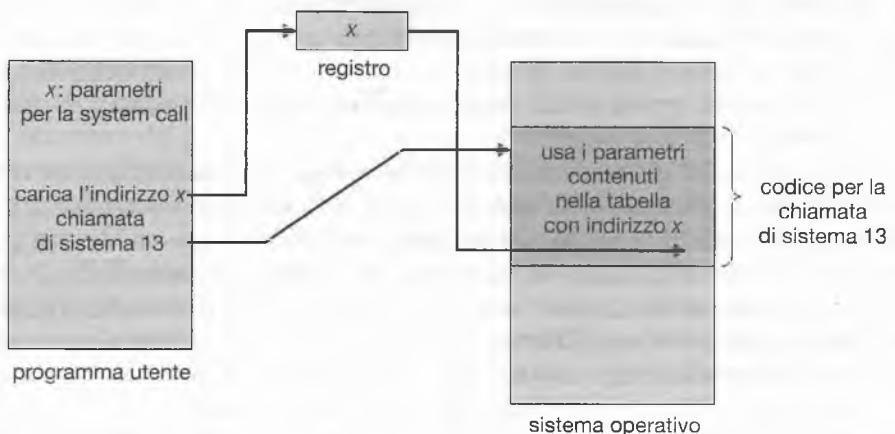
Le chiamate di sistema si presentano in modi diversi, secondo il calcolatore in uso. Spesso sono richieste maggiori informazioni oltre alla semplice identità della chiamata di sistema desiderata. Il tipo e l'entità esatti delle informazioni variano secondo lo specifico sistema operativo e la specifica chiamata di sistema. Per ottenere l'immissione di un dato, per



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

esempio, può essere necessario specificare il file o il dispositivo da usare come sorgente e anche l'indirizzo e l'ampiezza dell'area della memoria in cui depositare i dati letti. Naturalmente, il dispositivo o il file e l'ampiezza possono essere impliciti nella chiamata di sistema.

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in *registri*; si possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un *blocco* o tabella di memoria e si passa l'indirizzo del blocco, in forma di parametro, in un registro (Figura 2.7). È il metodo seguito dal sistema operativo Linux. Il programma può anche collocare (*push*) i parametri in una pila da cui sono prelevati (*pop*) dal sistema operativo. Alcuni sistemi operativi preferiscono i metodi del blocco o della pila, poiché non limitano il numero o la lunghezza dei parametri da passare.



**Figura 2.7** Passaggio di parametri in forma di tabella.

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

**Figura 2.8** Tipi di chiamate di sistema.

## 2.4 Categorie di chiamate di sistema

Le chiamate di sistema sono classificabili approssimativamente in cinque categorie principali: **controllo dei processi, gestione dei file, gestione dei dispositivi, gestione delle informazioni e comunicazioni**. Nei Paragrafi dal 2.4.1 al 2.4.6 sono illustrati brevemente i tipi di chiamate di sistema forniti da un sistema operativo. La maggior parte di queste chiamate di sistema implica o presuppone concetti e funzioni trattati in capitoli successivi. La Figura 2.8 riassume i tipi di chiamate di sistema forniti normalmente da un sistema operativo.

### ESEMPIO DI CHIAMATE DI SISTEMA DI WINDOWS E UNIX

	Windows	UNIX
Controllo dei processi	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Gestione dei file	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Gestione dei dispositivi	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Gestione delle informazioni	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Comunicazione	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protezione	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

#### 2.4.1 Controllo dei processi

Un programma in esecuzione deve potersi fermare in modo sia normale (in tal caso si parla di *end*) sia anormale (*abort*). Se si ricorre a una chiamata di sistema per terminare in modo anormale un programma in esecuzione, oppure se il programma incontra difficoltà e causa l'emissione di un segnale di eccezione, talvolta si ha la registrazione in un file di un'immagine del contenuto della memoria (*dump*) e l'emissione di un messaggio d'errore. Uno specifico programma di ricerca e correzione degli errori (*debugger*) può esaminare tali informazioni per determinare le cause del problema. Sia in condizioni normali sia anormali il sistema operativo deve trasferire il controllo all'interprete dei comandi che legge il comando successivo. In un sistema interattivo l'interprete dei comandi continua semplicemente a interpretare il comando successivo; si suppone che l'utente invii un comando idoneo per rispondere a qualsiasi errore. In un sistema a interfaccia GUI una finestra avverte l'utente dell'errore e richiede chiarimenti. In un sistema a lotti l'interprete dei comandi generalmente termina il lavoro corrente e prosegue con il successivo. Quando si presenta un errore, alcuni sistemi permettono alle schede di controllo di indicare le specifiche azioni di recupero da intraprendere. La *scheda di controllo* è un concetto proveniente dai sistemi a lotti: è in sostanza un comando per gestire l'esecuzione di un processo. Se il programma scopre un errore nei dati ricevuti e intende terminare in modo anormale, può anche definire un livello d'errore. Più grave è l'errore, più alto è il livello del parametro che lo individua. Sono quindi possibili una terminazione normale e una terminazione anormale, definendo la termina-

zione normale come errore di livello 0. L'interprete dei comandi o un programma successivo possono usare questo livello d'errore per determinare l'azione da intraprendere.

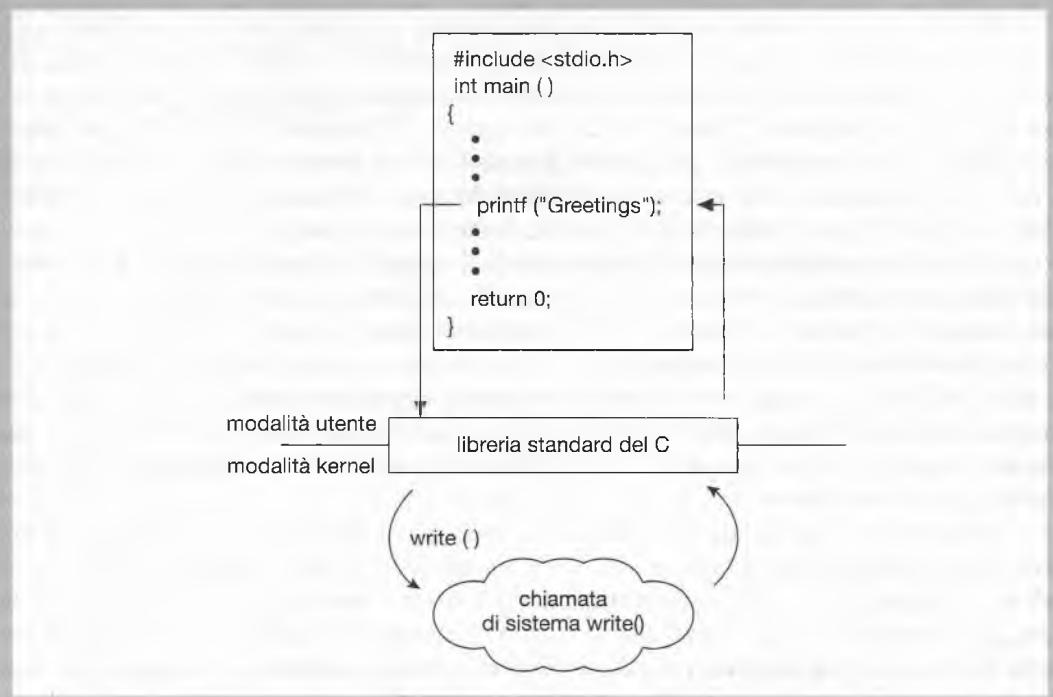
Un processo che esegue un programma può richiedere di caricare (*load*) ed eseguire (*execute*) un altro programma. In questo caso l'interprete dei comandi esegue un programma come se tale richiesta fosse stata impartita, per esempio, da un comando utente, oppure dal clic di un mouse. È interessante chiedersi dove si debba restituire il controllo una volta terminato il programma caricato. La questione è legata alle eventualità che il programma attuale sia andato perso, salvato oppure che abbia continuato l'esecuzione in modo concorrente con il nuovo programma.

Se al termine del nuovo programma il controllo rientra nel programma esistente, si deve salvare l'immagine della memoria del programma attuale, creando così effettivamente un meccanismo con cui un programma può richiamare un altro programma. Se entrambi i programmi continuano l'esecuzione in modo concorrente, si è creato un nuovo processo da sottoporre a multiprogrammazione. A questo scopo spesso si fornisce una chiamata di sistema specifica, e precisamente *create process* oppure *submit job*.

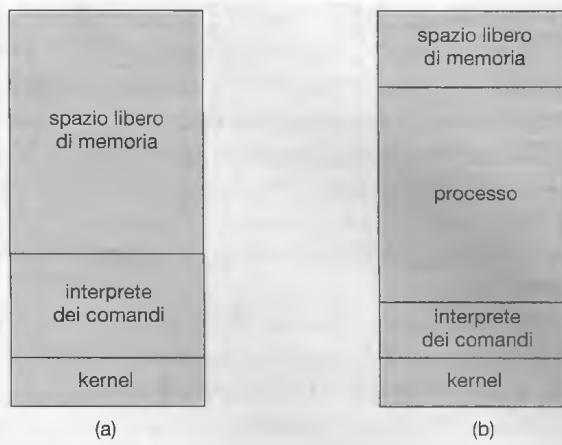
Quando si crea un nuovo processo, o anche un insieme di processi, è necessario manteenerne il controllo; ciò richiede la capacità di determinare e reimpostare gli attributi di un pro-

### ESEMPIO DI LIBRERIA STANDARD DEL LINGUAGGIO C

La libreria standard del linguaggio C fornisce una parte dell'interfaccia alle chiamate di sistema per molte versioni di UNIX e Linux. Come esempio, supponiamo che un programma C invochi la funzione *printf()*. La libreria C intercetta la funzione e invoca le necessarie chiamate di sistema: in questo caso, la chiamata *write()*. La libreria riceve il valore restituito da *write()* e lo passa al programma utente. Il meccanismo è illustrato nella Figura 2.6.



**Figura 2.9** Gestione di *write()* della libreria standard del C.



**Figura 2.10** Esecuzione nell'MS-DOS. (a) All'avviamento del sistema. (b) Durante l'esecuzione di un programma.

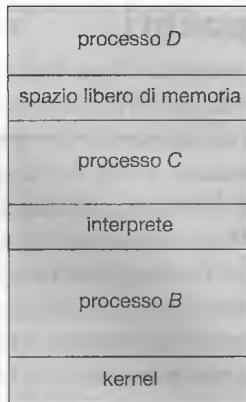
cesso, compresi la sua priorità, il suo tempo massimo d'esecuzione e così via (`get process attributes` e `set process attributes`). Inoltre, può essere necessario terminare un processo creato, se si riscontra che non è corretto o se non serve (`terminate process`).

Una volta creati, può essere necessario attendere che i processi terminino la loro esecuzione. Quest'attesa si può impostare per un certo periodo di tempo (`wait tempo`), ma è più probabile che si preferisca attendere che si verifichi un dato evento (`wait evento`). I processi devono quindi segnalare il verificarsi di quell'evento (`signal evento`). Chiamate di sistema di questo tipo, che trattano cioè il coordinamento di processi concorrenti, sono esaminate in profondità nel Capitolo 6.

Il controllo dei processi presenta così tanti aspetti e varianti: per chiarire questi concetti conviene ricorrere ad alcuni esempi. Il sistema operativo MS-DOS è un sistema che dispone di un interprete di comandi, attivato all'avviamento del calcolatore, e che esegue un solo programma alla volta (Figura 2.10.a), impiegando un metodo semplice e senza creare alcun nuovo processo; carica il programma in memoria, riscrivendo anche la maggior parte della memoria che esso stesso occupa, in modo da lasciare al programma quanta più memoria è possibile (Figura 2.10.b); quindi imposta il contatore di programma alla prima istruzione del programma da eseguire. A questo punto si esegue il programma e si possono verificare due situazioni: un errore causa un segnale di eccezione, oppure il programma esegue una chiamata di sistema per terminare la propria esecuzione. In entrambi i casi si registra il codice d'errore in memoria di sistema per un eventuale uso successivo, quindi quella piccola parte dell'interprete che non era stata sovrascritta riprende l'esecuzione e il suo primo compito consiste nel ricaricare dal disco la parte rimanente dell'interprete stesso. Eseguito questo compito, quest'ultimo mette a disposizione dell'utente, o del programma successivo, il codice d'errore registrato.

Nel FreeBSD (derivato da UNIX Berkeley), quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (in quest'ambiente chiamato *shell*) scelto dall'utente. Quest'interprete è simile a quello dell'MS-DOS nell'accettare i comandi e nell'eseguire programmi richiesti dall'utente. Tuttavia, poiché il FreeBSD è un sistema a partizione del tempo, può eseguire più programmi contemporaneamente, l'interprete dei comandi può continuare l'esecuzione mentre si esegue un altro programma (Figura 2.11).

Per avviare un nuovo processo, l'interprete dei comandi esegue la chiamata di sistema `fork()`; si carica il programma selezionato in memoria tramite la chiamata di sistema



**Figura 2.11** Esecuzione di più programmi nel sistema operativo FreeBSD.

`exec()` e infine si esegue il programma. Secondo come il comando è stato impartito, l'interprete dei comandi attende il termine del processo, oppure esegue il processo *background*. In quest'ultimo caso l'interprete dei comandi richiede immediatamente un altro comando. Se un processo è eseguito in background, non può ricevere dati direttamente dalla tastiera, giacché anche l'interprete dei comandi sta usando tale risorsa. L'eventuale operazione di I/O è dunque eseguita tramite un file o tramite un'interfaccia GUI. Nel frattempo l'utente è libero di richiedere all'interprete dei comandi l'esecuzione di altri, di controllare lo svolgimento del processo in esecuzione, di modificare la priorità di quel programma e così via. Completato il proprio compito, il processo esegue una chiamata di sistema `exit()` per terminare la propria esecuzione, riportando al processo chiamante un codice di stato 0 oppure un codice d'errore diverso da 0. Questo codice di stato (o d'errore) rimane disponibile per l'interprete dei comandi o per altri programmi. I processi sono trattati nel Capitolo 3, dove si illustra un esempio di programma che utilizza le chiamate di sistema `fork()` ed `exec()`.

## 2.4.2 Gestione dei file

Il file system è esaminato più in profondità nei Capitoli 10 e 11, tuttavia possiamo già identificare diverse chiamate di sistema riguardanti i file.

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

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

### 2.4.3 Gestione dei dispositivi

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

Le diverse risorse controllate dal sistema operativo si possono concepire come dei dispositivi, alcuni dei quali sono in effetti dispositivi fisici (per esempio nastri), mentre altre sono da considerarsi dispositivi astratti o virtuali (i file, per esempio). In presenza di utenti multipli, il sistema potrebbe prescrivere la richiesta (tramite `request`) del dispositivo, al fine di assicurarne l'uso esclusivo. Dopo l'uso, avviene il rilascio (tramite `release`). Si tratta di funzioni analoghe alle chiamate `open` e `close` per i file. Altri sistemi operativi permettono l'accesso incontrollato ai dispositivi, con il rischio che la competizione per l'accesso provochi lo stallo (Capitolo 7).

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

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

### 2.4.4 Gestione delle informazioni

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

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

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

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

### 2.4.5 Comunicazione

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

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

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

### 2.4.6 Protezione

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

Tra le chiamate di sistema che offrono meccanismi di protezione vi sono solitamente la `set permission` e la `get permission`, che permettono di modificare i permessi di accesso a risorse come file e dischi. Le chiamate di sistema `allow user` e `deny user` specificano se un particolare utente abbia il permesso di accesso a determinate risorse.

La protezione è trattata nel Capitolo 14. Il Capitolo 15 esamina la sicurezza in maniera più estesa.

## 2.5 Programmi di sistema

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

- ◆ **Gestione dei file.** Questi programmi creano, cancellano, copiano, ridenominano, stampano, elencano e in genere compiono operazioni sui file e le directory.
- ◆ **Informazioni di stato.** Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti o informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni tramite terminale, o tramite altri dispositivi per l'uscita dei dati, o, ancora, all'interno di una finestra della GUI. Alcuni sistemi comprendono anche **registri (registry)**, al fine di archiviare e poter poi consultare informazioni sulla configurazione del sistema.
- ◆ **Modifica dei file.** Diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.
- ◆ **Ambienti d'ausilio alla programmazione.** Compilatori, assemblatori, programmi per la correzione degli errori e interpreti dei comuni linguaggi di programmazione, come C, C++, Java, Visual Basic e PERL, sono spesso forniti insieme con il sistema operativo.
- ◆ **Caricamento ed esecuzione dei programmi.** Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*). Sono necessari anche i sistemi d'ausilio all'individuazione e correzione degli errori (*debugger*) per i linguaggi d'alto livello o per il linguaggio macchina.
- ◆ **Comunicazioni.** Questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi. Permettono agli utenti d'invia-

re messaggi agli schermi d'altri utenti, di consultare il Web, d'inviare messaggi di posta elettronica, di accedere a calcolatori remoti, di trasferire file da un calcolatore a un altro. Oltre ai programmi di sistema, con la maggior parte dei sistemi operativi sono forniti programmi che risolvono problemi comuni o che eseguono operazioni comuni, quali quelli di consultazione del Web, elaboratori di testi, fogli di calcolo, sistemi di basi di dati, compilatori, programmi di disegno, programmi per analisi statistiche e videogiochi.

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

## 2.6 Progettazione e realizzazione di un sistema operativo

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

### 2.6.1 Scopi della progettazione

Il primo problema che s'incontra nella progettazione di un sistema riguarda la definizione degli scopi e delle caratteristiche del sistema stesso. Al più alto livello, la progettazione del sistema è influenzata in modo decisivo dalla scelta dell'architettura fisica e del tipo di sistema: a lotti o a partizione del tempo, mono o multiutente, distribuito, per elaborazioni in tempo reale o d'uso generale.

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

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

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

Non esiste una soluzione unica al problema della definizione dei requisiti di un sistema operativo. L'ampia gamma di sistemi mostra che da requisiti diversi possono risultare le soluzioni più varie per ambienti diversi. Per esempio, i requisiti VxWorks, un sistema operativo real-time per sistemi integrati, sono assai diversi da MVS, il sistema operativo multiaccesso e multiutente per mainframe IBM.

## 2.6.2 Meccanismi e criteri

Un principio molto importante è quello che riguarda la distinzione tra **meccanismi** e **criteri** o **politiche** (*policy*). I meccanismi determinano *come* eseguire qualcosa; i criteri, invece, stabiliscono *che cosa* si debba fare. Il timer del sistema (Paragrafo 1.5.2), per esempio, è un meccanismo che assicura la protezione della CPU, ma la decisione riguardante la quantità di tempo da impostare nel timer per un utente specifico riguarda i criteri.

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

I sistemi operativi basati su microkernel (Paragrafo 2.7.3) portano alle estreme conseguenze la separazione dei meccanismi dai criteri, fornendo un insieme di funzioni fondamentali da impiegare come elementi di base; tali funzioni, quasi indipendenti dai criteri, consentono l'aggiunta di meccanismi e criteri più complessi tramite moduli del kernel creati dagli utenti o anche tramite programmi utenti. Per un esempio, si consideri l'evoluzione di UNIX. In principio, si trattava di un sistema basato sulla ripartizione del tempo. Nelle ultime versioni di Solaris, lo scheduling è controllato da tabelle caricabili: a seconda della tabella corrente, il sistema può essere a tempo ripartito, a lotti, in tempo reale, fair share, o adottare una combinazione delle strategie precedenti. Tale parametrizzazione dei meccanismi di scheduling permette di attuare cambiamenti di vasta portata ai criteri del sistema con l'esecuzione di un singolo comando di caricamento (`load-new-table`) di una nuova tabella. All'altro estremo si trovano sistemi come Windows, nei quali sia i criteri sia i meccanismi sono fissati a priori e cablati nel sistema, al fine di fornire agli utenti un'unica immagine globale. In questi casi, infatti, tutte le applicazioni hanno interfacce simili, perché l'interfaccia stessa fa parte del kernel e delle librerie del sistema. Il sistema Mac OS X è di tipo analogo.

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

## 2.6.3 Realizzazione

Una volta progettato, un sistema operativo va realizzato. Tradizionalmente i sistemi operativi si scrivevano in un linguaggio assembly, attualmente si scrivono spesso in linguaggi di alto livello come il C o il C++.

Il primo sistema scritto in un linguaggio di alto livello fu probabilmente il Master Control Program (MCP) per i calcolatori Burroughs; l'MCP fu scritto infatti in una variante del linguaggio ALGOL; il MULTICS, sviluppato al MIT, fu scritto prevalentemente in PL/I; Linux e Windows XP sono scritti per lo più in C, sebbene si trovino alcune brevi sezioni di codice assembly che riguardano i driver dei dispositivi e il salvataggio e il ripristino dei registri del sistema.

I vantaggi derivanti dall'uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, sono gli stessi che si ottengono quando il linguaggio si usa per i programmi applicativi: il codice si scrive più rapidamente, è più compatto ed è più facile da capire e mettere a punto. Inoltre il perfezionamento delle tecniche di compilazione consente di migliorare il codice generato per l'intero sistema ope-

rativo con una semplice ricompilazione. Infine, un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un'altra architettura (*porting*). L'MS-DOS, per esempio, fu scritto nel linguaggio assembly dell'Intel 8088, quindi è disponibile solo per la famiglia di CPU Intel. Il sistema operativo Linux, scritto prevalentemente in C, è invece disponibile su diversi tipi di CPU, tra cui Intel 80x86, Motorola 680x0, SPARC e MIPS RX000.

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

Come in altri sistemi, i miglioramenti principali nel rendimento sono dovuti più a strutture dati e algoritmi migliori che a un ottimo codice in linguaggio assembly. Inoltre, sebbene i sistemi operativi siano molto grandi, solo una piccola parte del codice assume un'importanza critica riguardo al rendimento: il gestore della memoria e lo scheduler della CPU sono probabilmente le procedure più critiche. Una volta scritto il sistema e verificato il suo corretto funzionamento, è possibile identificare le procedure che possono costituire colli di bottiglia e sostituirle con procedure equivalenti scritte in linguaggio assembly.

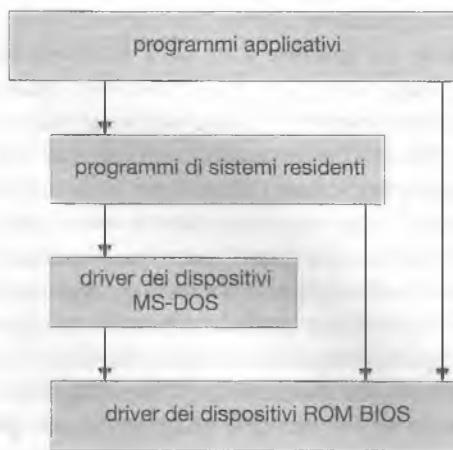
## 2.7 Struttura del sistema operativo

Affinché possa funzionare correttamente ed essere facilmente modificato, un sistema vasto e complesso come un sistema operativo moderno va progettato con estrema attenzione. Anzi-ché progettare un sistema monolitico, un orientamento diffuso prevede la sua suddivisione in piccoli componenti; ciascuno deve essere un modulo ben definito del sistema, con interfacce e funzioni definite con precisione. Nel Capitolo 1 sono stati brevemente illustrati i componenti comuni ai sistemi operativi; nel paragrafo seguente si descrive come questi componenti siano interconnessi e fusi in un kernel.

### 2.7.1 Struttura semplice

Molti sistemi commerciali non hanno una struttura ben definita; spesso sono nati come sistemi piccoli, semplici e limitati, e solo in un secondo tempo si sono accresciuti superando il loro scopo originale. Un sistema di questo tipo è l'MS-DOS, originariamente progettato e realizzato da persone che non avrebbero mai immaginato una simile diffusione. Non fu suddiviso attentamente in moduli poiché, a causa dei limiti dell'architettura su cui era eseguito, lo scopo prioritario era fornire la massima funzionalità nel minimo spazio. La Figura 2.12 riporta la sua struttura.

In MS-DOS non vi è una netta separazione fra le interfacce e i livelli di funzionalità, tanto che, per esempio, le applicazioni accedono direttamente alle routine di sistema per l'I/O, scrivendo direttamente sul video e sui dischi. Libertà di questo genere rendono MS-DOS vulnerabile agli errori e agli attacchi dei programmi utenti, fino al blocco totale del sistema. Naturalmente, le limitazioni di MS-DOS riflettono quelle dell'hardware disponibile ai tempi della sua progettazione. Il processore Intel 8088, per il quale fu scritto, non distingueva fra modalità utente e di sistema, e non offriva protezione hardware, ciò che non lasciava altra scelta ai progettisti di MS-DOS se non permettere accesso incondizionato all'hardware sottostante.

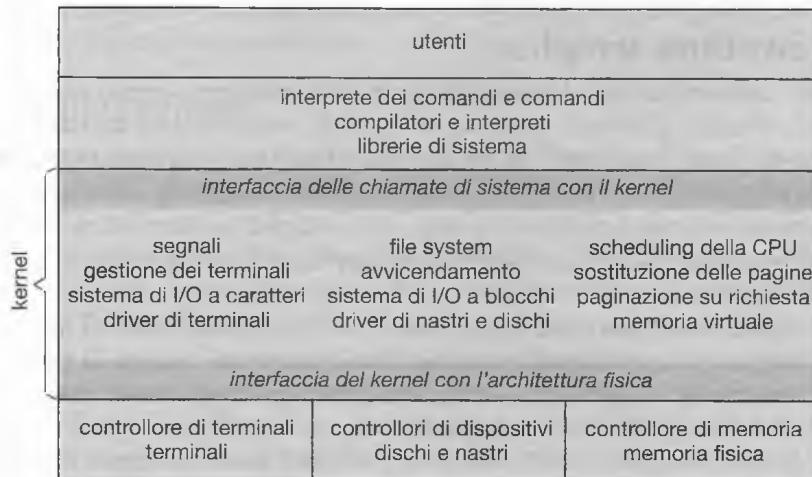


**Figura 2.12** Struttura degli strati dell'MS-DOS.

Anche il sistema UNIX originale è poco strutturato, a causa delle limitazioni dell'hardware disponibile al tempo della sua progettazione. Il sistema consiste di due parti separate, il kernel e i programmi di sistema. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi, aggiunti ed espansi nel corso dell'evoluzione di UNIX. La Figura 2.13 mostra i diversi livelli di UNIX: tutto ciò che sta al di sotto dell'interfaccia alle chiamate di sistema e al di sopra dell'hardware costituisce il kernel. Esso comprende il file system, lo scheduling della CPU, la gestione della memoria, e le altre funzionalità del sistema rese disponibili tramite chiamate di sistema. Tutto sommato, si tratta di un'enorme massa di funzionalità diverse combinate in un solo livello. È questa struttura monolitica che rendeva difficile l'implementazione e la manutenzione.

### 2.7.2 Metodo stratificato

In presenza di hardware appropriato, i sistemi operativi possono essere suddivisi in moduli più piccoli e gestibili di quanto non fosse possibile nelle prime versioni di MS-DOS e UNIX.



**Figura 2.13** Struttura del sistema UNIX.

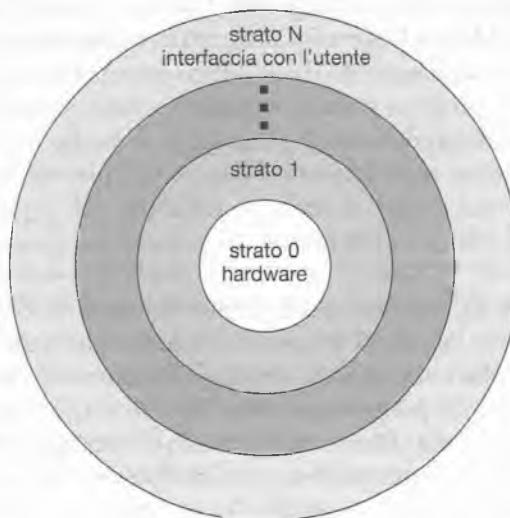
Ciò permette al sistema operativo di mantenere un controllo molto più stretto delle applicazioni che girano sulla macchina. Inoltre, gli sviluppatori del sistema godono di maggiore libertà nel modificare i meccanismi interni del sistema e nel suddividere il sistema in moduli. Secondo i dettami della metodologia di progettazione dall'alto verso il basso (*top-down approach*), le specifiche del sistema partono dall'individuazione delle sue funzionalità e delle sue caratteristiche complessive, che sono poi suddivise in componenti distinte. L'attenzione all'incapsulamento delle informazioni è anche importante, in quanto dà libertà agli sviluppatori di implementare le routine di basso livello nel modo che ritengono più opportuno, fintanto che l'interfaccia esterna alla routine rimanga invariata, e la routine stessa esegua il compito per cui è stata prevista.

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

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

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

Ogni strato si realizza impiegando unicamente le operazioni messe a disposizione dagli strati inferiori, considerando soltanto le azioni che compiono, senza entrare nel merito di



**Figura 2.14** Struttura a strati di un sistema operativo.

come queste sono realizzate. Di conseguenza ogni strato nasconde a quelli superiori l'esistenza di determinate strutture dati, operazioni ed elementi fisici.

La principale difficoltà del metodo stratificato risiede nella definizione appropriata dei diversi strati. È necessaria una progettazione accurata, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia. Il dispositivo che controlla lo spazio sul disco usato dagli algoritmi che implementano la memoria virtuale deve risiedere in uno strato che si trovi sotto le routine per la gestione della memoria, perché essa richiede l'utilizzo di quello spazio.

Altri requisiti possono non essere così ovvi. Normalmente il driver della memoria ausiliaria (*backing store*) dovrebbe trovarsi sopra lo scheduler della CPU, poiché può accadere che il driver debba attendere un'istruzione di I/O, e in questo periodo la CPU si può sottoporre a scheduling. Tuttavia, in un grande sistema, lo scheduler della CPU può avere più informazioni su tutti i processi attivi di quante se ne possano contenere in memoria; perciò è probabile che queste informazioni si debbano caricare e scaricare dalla memoria, quindi il driver della memoria ausiliaria dovrebbe trovarsi sotto lo scheduler della CPU.

Un ulteriore problema che si pone con la struttura stratificata è che essa tende a essere meno efficiente delle altre; per esempio, per eseguire un'operazione di I/O un programma utente invoca una chiamata di sistema che è intercettata dallo strato di I/O che, a sua volta, esegue una chiamata allo strato di gestione della memoria, che a sua volta richiama lo strato di scheduling della CPU e che quindi è passata all'opportuno dispositivo di I/O. In ciascuno strato i parametri sono modificabili, può rendersi necessario il passaggio di dati, e così via; ciascuno strato aggiunge un carico alla chiamata di sistema. Ne risulta una chiamata di sistema che richiede molto più tempo di una chiamata di sistema corrispondente in un sistema non stratificato.

Negli ultimi anni questi limiti hanno causato una piccola battuta d'arresto allo sviluppo della stratificazione. Attualmente si progettano sistemi basati su un numero inferiore di strati con più funzioni, che offrono la maggior parte dei vantaggi del codice modulare, evitando i difficili problemi connessi alla definizione e all'interazione degli strati.

### 2.7.3 Microkernel

A mano a mano che il sistema operativo UNIX è stato esteso, il kernel è cresciuto notevolmente, diventando sempre più difficile da gestire. Verso la metà degli anni '80 un gruppo di ricercatori della Carnegie Mellon University progettò e realizzò un sistema operativo, Mach, col kernel strutturato in moduli secondo il cosiddetto orientamento a **microkernel**. Seguendo questo orientamento si progetta il sistema operativo rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema. Ne risulta un kernel di dimensioni assai inferiori. Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione.

Lo scopo principale del microkernel è fornire funzioni di comunicazione tra i programmi client e i vari servizi, anch'essi in esecuzione nello spazio utente. La comunicazione si realizza secondo il modello a scambio di messaggi, descritto nel Paragrafo 2.4.5. Per accedere a un file, per esempio, un programma client deve interagire con il file server; ciò non avviene mai in modo diretto, ma tramite uno scambio di messaggi con il microkernel.

Uno dei vantaggi del microkernel è la facilità di estensione del sistema operativo: i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel. Poiché è ridotto all'essenziale, se il kernel deve essere modificato, i cambiamenti da fare sono

ben circoscritti, e il sistema operativo risultante è più semplice da adattare alle diverse architetture. Inoltre offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto.

L'orientamento a microkernel è stato adottato per molti sistemi operativi moderni: il sistema Tru64 UNIX (in origine Digital UNIX), per esempio, offre all'utente un'interfaccia di tipo UNIX, ma il suo kernel è il Mach (il kernel traduce le chiamate di sistema di UNIX in messaggi ai corrispondenti servizi del livello utente). Anche il kernel di Mac OS X (conosciuto con il nome di *Darwin*) è basato sul microkernel Mach.

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

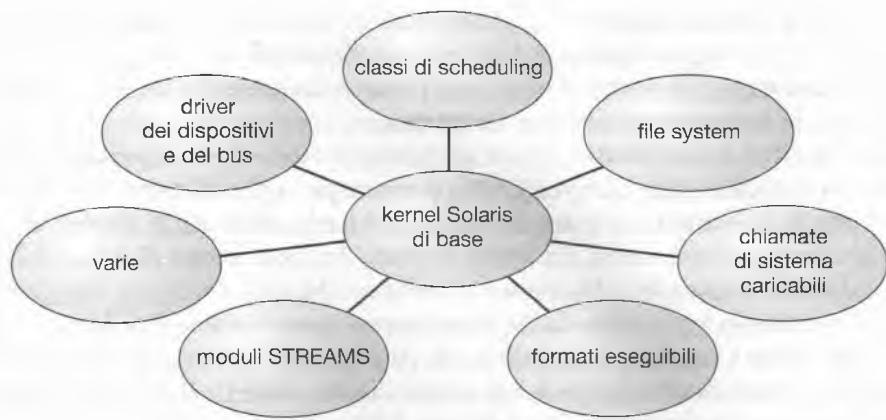
Purtroppo i microkernel possono incorrere in cali di prestazioni dovuti al sovraccarico indotto dall'esecuzione di processi utente con funzionalità di sistema. La prima versione di Windows NT, basata su un microkernel stratificato, aveva prestazioni inferiori rispetto a Windows 95. La versione 4.0 di Windows NT risolse parzialmente il problema, spostando strati dal livello utente al livello kernel, e integrandoli più strettamente. Questa tendenza fece sì che, al tempo della progettazione di Windows XP, l'architettura di NT era ormai monolitica.

## 2.7.4 Moduli

Forse il miglior approccio attualmente disponibile per la progettazione dei sistemi operativi si fonda su tecniche della programmazione orientata agli oggetti per implementare un kernel modulare. In questo contesto, il kernel è costituito da un insieme di componenti fondamentali, integrati poi da funzionalità aggiunte dinamicamente durante l'avvio o l'esecuzione. Questa strategia, che impiega moduli caricati dinamicamente, è comune nelle implementazioni moderne di UNIX, come Solaris, Linux e Mac OS X. La struttura di Solaris, per esempio, illustrata dalla Figura 2.15, si incentra su un kernel dotato dei seguenti sette tipi di moduli caricabili.

1. Classi di scheduling.
2. File system.
3. Chiamate di sistema caricabili.
4. Formati eseguibili.
5. Moduli STREAMS.
6. Varie.
7. Driver dei dispositivi e del bus.

Questa organizzazione lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente certe caratteristiche. È possibile aggiungere driver per dispositivi specifici, per esempio, o gestire file system diversi tramite moduli caricabili. Il risultato complessivo somiglia ai sistemi a strati, perché ogni parte del kernel ha interfacce ben definite e protette. È tuttavia più flessibile dei sistemi a strati tradizionali, perché ciascun modulo può invocare funzionalità di un qualunque altro modulo. Inoltre, come nei sistemi basati su microkernel, il modulo principale gestisce solo i servizi essenziali, oltre a poter caricare altri moduli e comunicare con loro. E tuttavia, rispetto ai sistemi orientati a un microkernel, l'efficienza è superiore, perché i moduli sono in grado di comunicare senza invocare le funzionalità di trasmissione dei messaggi.



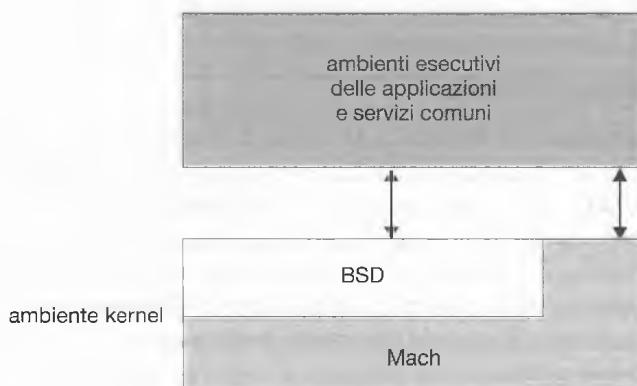
**Figura 2.15** Moduli caricabili di Solaris.

Il sistema operativo Mac OS X di Apple adotta una struttura ibrida; è organizzato in strati, uno dei quali contiene il microkernel Mach. Si veda la Figura 2.16.

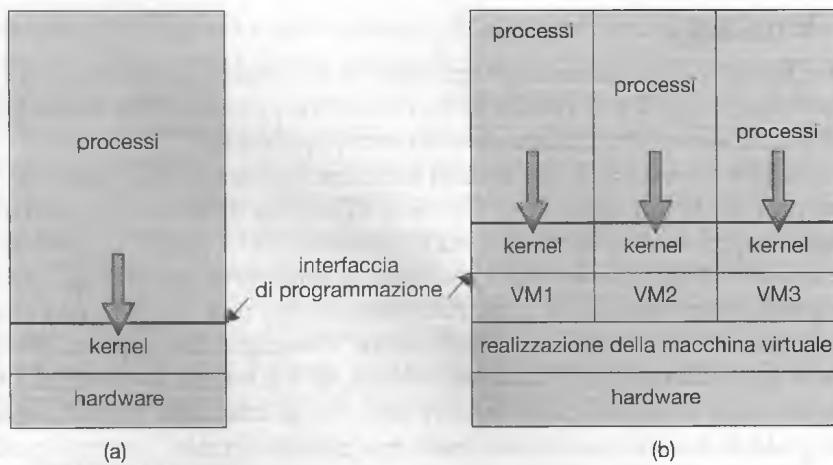
Gli strati superiori comprendono gli ambienti esecutivi delle applicazioni e un insieme di servizi che offre un'interfaccia grafica con le applicazioni. Il kernel si trova in uno strato sottostante, ed è costituito dal microkernel Mach e dal kernel BSD. Il primo cura la gestione della memoria, le chiamate di procedure remote (RPC), la comunicazione tra processi (IPC) – compreso lo scambio di messaggi – e lo scheduling dei thread. Il secondo mette a disposizione un'interfaccia BSD a riga di comando, i servizi legati al file system e alla rete, e la API POSIX, compreso Pthreads. Oltre a Mach e BSD, il kernel possiede un kit di strumenti connessi all'I/O per lo sviluppo di driver dei dispositivi e di moduli dinamicamente caricabili, detti **estensioni del kernel** nel gergo di Mac OS X. Come si vede dalla figura, le applicazioni e i servizi comuni possono accedere direttamente sia ai servizi di Mach sia a quelli di BSD.

## 2.8 Macchine virtuali

La metodologia di progettazione per strati descritta nel Paragrafo 2.7.2 porta nella sua naturale conclusione al concetto di **macchina virtuale** (*virtual machine*). L'idea alla base delle macchine virtuali è di astrarre dalle unità hardware del singolo computer (CPU, memoria,



**Figura 2.16** Struttura di Mac OS X.



**Figura 2.17** Modelli di sistema: (a) Semplice. (b) Macchina virtuale.

dispositivi periferici, e così via), progettando per ciascuna unità un ambiente esecutivo software diverso, così da dare l'impressione che ognuno di loro giri sulla propria macchina.

Tramite lo scheduling della CPU (Capitolo 5) e la memoria virtuale (Capitolo 9), il sistema operativo può dare l'impressione che ogni processo sia dotato del proprio processore e del proprio spazio di memoria (virtuale). La macchina virtuale fornisce un'interfaccia *coincidente* con il nudo hardware. Ogni processo ospite può usufruire di una copia (virtuale) del calcolatore sottostante (Figura 2.17). Solitamente il processo ospite è un sistema operativo. In questo modo una singola macchina fisica può far girare più sistemi operativi concorrentemente, ciascuno sulla sua macchina virtuale.

### 2.8.1 Storia

Le macchine virtuali hanno fatto la loro prima apparizione commerciale nel 1972 sui mainframe IBM, grazie al sistema operativo VM. Questo sistema ha subito un'evoluzione ed è ancora oggi disponibile. Molte delle idee introdotte da VM si possono ritrovare in altri sistemi e dunque vale la pena esplorare questo strumento.

IBM VM370 suddivideva un mainframe in diverse macchine virtuali, ognuna dotata del proprio sistema operativo. Una tra le difficoltà maggiori che si incontrarono con le macchine virtuali di VM riguardò i sistemi di dischi. Supponiamo che la macchina fisica contenga tre dischi, ma voglia supportare sette macchine virtuali. Chiaramente, non era possibile allocare un disco fisso a ogni macchina virtuale, perché il solo software della macchina virtuale richiedeva una notevole quantità di spazio su disco per memoria virtuale e spooling. La soluzione fu quella di fornire dischi virtuali (chiamati *minidisk* nel sistema operativo VM di IBM) fra loro identici in tutto tranne che nelle dimensioni. Il sistema implementava ciascun minidisk allocando la quantità necessaria di tracce sul disco fisso.

Una volta create queste macchine virtuali, l'utente poteva eseguire tutti i sistemi operativi e i pacchetti software disponibili sulla macchina sottostante. Nel caso di VM, l'utente eseguiva solitamente CMS, un sistema operativo interattivo a singolo utente.

## 2.8.2 Vantaggi

Sono diverse le ragioni che portano alla creazione di una macchina virtuale, ma la maggior parte è legata alla possibilità di condividere l'utilizzo concorrente dello stesso hardware in diversi ambienti di esecuzione (ovvero diversi sistemi operativi).

Un vantaggio importante è che sistema ospitante è protetto dalle macchine virtuali, e queste sono protette le une dalle altre. Un virus all'interno di un sistema operativo ospite può danneggiare quel sistema operativo, ma è improbabile che colpisca il sistema ospitante o altri sistemi ospiti. Non ci sono quindi problemi di protezione, perché ogni macchina virtuale è completamente isolata dalle altre. Allo stesso tempo non vi è però una condivisione diretta delle risorse. Per la condivisione delle risorse si seguono due approcci diversi. Il primo prevede la possibilità di condividere un volume del file system, e quindi di condividere file. Il secondo offre la possibilità di definire una rete di macchine virtuali, ognuna delle quali sia in grado di inviare informazioni sulla rete privata virtuale. La rete è modellata come una rete fisica, ma è implementata via software.

Una macchina virtuale è uno strumento perfetto per condurre ricerche sui sistemi operativi e per svilupparne di nuovi. La modifica di un sistema operativo è solitamente un compito difficile. I sistemi operativi sono programmi lunghi e complessi ed è difficile essere sicuri che un cambiamento in una parte del sistema non causi dei malfunzionamenti nascosti in altre parti. La potenza dei sistemi operativi rende le modifiche particolarmente pericolose. Essi, infatti, sono in esecuzione in modalità kernel, e un'errata modifica a un puntatore potrebbe causare un errore che cancella l'intero file system. È quindi necessario un test scrupoloso dopo ogni modifica effettuata.

I sistemi operativi gestiscono e controllano l'intera macchina. È quindi necessario, dopo eventuali modifiche e durante i test, arrestare il sistema e renderlo inutilizzabile dagli utenti. Questo periodo di tempo è solitamente chiamato **periodo di sviluppo del sistema** (*system-development time*). Il periodo di sviluppo del sistema è usualmente pianificato la notte o nel fine settimana, quando il carico del sistema è basso, proprio perché in questo periodo il sistema operativo non è utilizzabile dagli utenti.

Una macchina virtuale può eliminare molti di questi problemi. Ai programmati di sistema è messa a disposizione una macchina virtuale dedicata e lo sviluppo di sistema è effettuato sulla macchina virtuale invece che sulla macchina fisica. Di rado le normali operazioni di sistema hanno bisogno di essere disturbate a causa dello sviluppo.

Un altro vantaggio che le macchine virtuali offrono agli sviluppatori è dato dal fatto che diversi sistemi operativi possono lavorare in concorrenza sulla stessa macchina. Una workstation così configurata permette una rapida portabilità delle applicazioni su differenti piattaforme e facilita la fase di test. In maniera del tutto simile, gli ingegneri addetti alla qualità possono provare le applicazioni su piattaforme multiple senza dover comprare, né aggiornare, né mantenere un computer diverso per ogni diversa piattaforma.

Un grande vantaggio delle macchine virtuali, sfruttato nei centri di trattamento dati, è il **consolidamento** del sistema, ovvero l'esecuzione su uno stesso sistema di due o più sistemi diversi originariamente installati su macchine fisiche distinte. Questo passaggio dai sistemi fisici ai sistemi virtuali permette un'efficace ottimizzazione delle risorse, in quanto diversi sistemi scarsamente utilizzati possono essere combinati su un'unica macchina utilizzata più intensamente.

Se l'utilizzo di macchine virtuali continuasse a diffondersi, lo sviluppo di applicazioni dovrebbe subire una conseguente evoluzione. Infatti, se un sistema può facilmente aggiungere, rimuovere o spostare macchine virtuali, perché installare un'applicazione direttamente su quel sistema? Piuttosto, gli sviluppatori installerebbero le applicazioni su un sistema operati-

vo correttamente configurato e personalizzato all'interno di una macchina virtuale. Questa piattaforma virtuale diventerà il meccanismo di distribuzione dell'applicazione. Questo metodo costituirebbe una notevole miglioria per gli sviluppatori: la gestione delle applicazioni diventerebbe più semplice, sarebbe richiesta una minor messa a punto (*tuning*) e il supporto tecnico sarebbe semplificato. Anche gli amministratori di sistema vedrebbero semplificato il loro compito. L'installazione sarebbe infatti più semplice e la ricompilazione di un'applicazione su un nuovo sistema sarebbe facilitata, non richiedendo più gli usuali passi di disinstallazione e reinstallazione. Affinché sia possibile un'adozione di massa della metodologia descritta, è però necessaria la definizione di uno standard del formato delle macchine virtuali, in modo che ogni macchina virtuale possa girare su ogni piattaforma di virtualizzazione. Il progetto "Open Virtual Machine Format" è un tentativo di definire questo standard e il suo successo porterebbe a una unificazione del formato delle macchine virtuali.

### 2.8.3 Simulazione

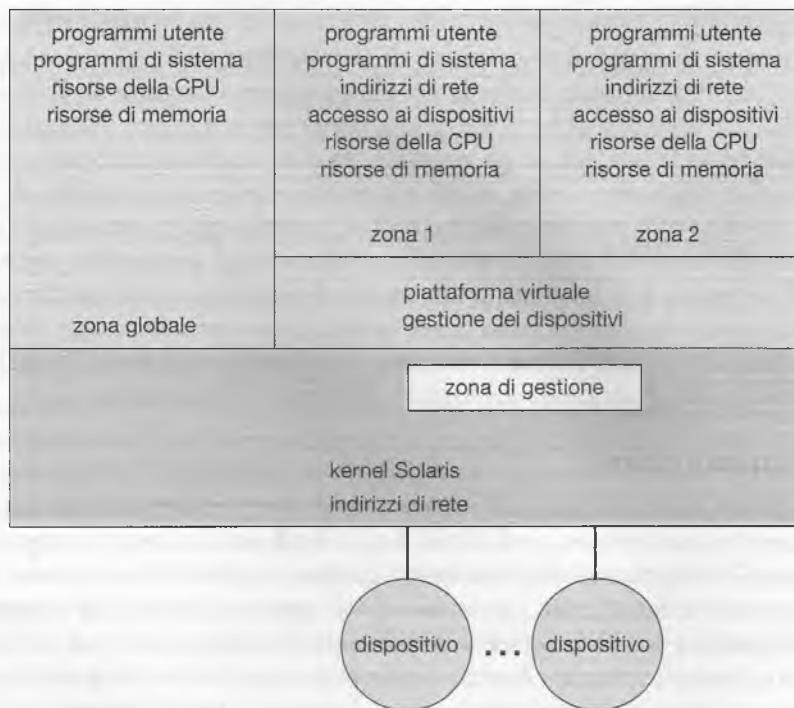
La virtualizzazione, descritta fin qui, è soltanto uno dei numerosi metodi per emulare un sistema. È il metodo più comune, perché fa in modo che il sistema operativo ospite e le applicazioni "credano" di essere in esecuzione su un hardware nativo. Visto che solo le risorse di sistema devono essere virtualizzate, i processi ospitati possono girare quasi a piena velocità.

Un altro metodo di emulazione è la **simulazione**. In questo caso si ha un sistema ospitante con una propria architettura e un sistema ospite compilato per un'architettura diversa. Supponiamo per esempio che un'azienda abbia sostituito i vecchi sistemi con sistemi più nuovi, ma che voglia continuare a utilizzare alcuni importanti programmi compilati per i vecchi sistemi. I programmi potrebbero essere mandati in esecuzione su un emulatore in grado di tradurre le istruzioni del vecchio sistema in istruzioni per il nuovo sistema. L'emulazione permette quindi di incrementare la vita dei programmi e inoltre aiuta a studiare vecchie architetture di sistema. Il grosso limite sono le prestazioni. Le istruzioni emulate vengono eseguite infatti molto più lentamente rispetto alle istruzioni native. Succede dunque che, anche se si dispone di una macchina 10 volte più potente rispetto alla vecchia, il programma in esecuzione sulla nuova macchina sarà ancora più lento di quanto lo era sull'hardware per il quale era stato progettato. Un'ulteriore difficoltà si incontra nella creazione di un emulatore che funzioni correttamente, perché, in sostanza, progettare un emulatore significa riscrivere un processore via software.

### 2.8.4 Paravirtualizzazione

Un'altra variazione sul tema è la **paravirtualizzazione**. Piuttosto che provare a offrire al sistema ospite una piattaforma identica a quella da esso desiderata, con la paravirtualizzazione si cerca di rendere disponibile al sistema ospite un sistema simile, ma non identico, alle sue preferenze. L'ospite deve allora essere modificato prima di essere eseguito su un hardware paravirtualizzato. Il lavoro addizionale richiesto da queste modifiche offre in cambio un guadagno in termini di uso più efficiente delle risorse e snellezza del livello paravirtuale.

Solaris 10 include dei **contenitori** (*containers*), anche detti **zone**, che creano un livello virtuale tra il sistema operativo e le applicazioni. In questo sistema è installato un unico kernel e l'hardware non è virtualizzato. A essere virtualizzato è il sistema operativo con i suoi dispositivi, in modo da dare ai processi un contenitore in cui vi sia l'impressione di essere l'unico processo in esecuzione sul sistema. Possono essere creati uno o più contenitori, ognuno con le proprie applicazioni, il proprio indirizzo di rete, le proprie porte, i propri account utente, e così via. Le risorse del processore possono ripartite tra i contenitori e i pro-



**Figura 2.18** Solaris 10 con due contenitori.

cessi di sistema. La Figura 2.18 mostra un sistema Solaris 10 con due contenitori e lo spazio utente standard, chiamato “globale”.

### 2.8.5 Realizzazione

Benché utile, il concetto di macchina virtuale è difficile da realizzare; è difficile ottenere un *esatto* duplicato della macchina sottostante. Si ricordi che la macchina sottostante ha due modalità di funzionamento, utente e di sistema. I programmi che realizzano il sistema di macchine virtuali si possono eseguire nella modalità di sistema, poiché costituiscono il sistema operativo, mentre ciascuna macchina virtuale può funzionare solo nella modalità utente. Quindi, proprio come la macchina fisica, anche quella virtuale deve avere due modalità; di conseguenza si devono avere modalità utente virtuale e modalità di sistema virtuale, entrambe operanti in modalità utente fisico. Le azioni che causano un trasferimento dalla modalità utente a quella di sistema su una macchina reale, come una chiamata di sistema o un tentativo d'esecuzione di un'istruzione privilegiata, in una macchina virtuale devono causare un analogo trasferimento dalla modalità utente virtuale a quella di sistema virtuale.

Generalmente questo trasferimento è eseguibile in modo abbastanza semplice. Se, per esempio, un programma in esecuzione su una macchina virtuale in modalità utente virtuale esegue una chiamata di sistema, si passa alla modalità di sistema della macchina virtuale all'interno della macchina reale. Quando il sistema della macchina virtuale acquisisce il controllo, può modificare il contenuto dei registri e il contatore di programma della macchina virtuale, in modo da simulare l'effetto della chiamata di sistema; quindi può riavviare la macchina virtuale, che si trova in modalità di sistema virtuale.

La differenza più rilevante consiste, naturalmente, nel tempo. Mentre l'I/O reale potrebbe richiedere 100 millisecondi, quello virtuale potrebbe comportare un tempo inferiore (poiché le operazioni di I/O avvengono su file contenuti in dischi, che simulano i dispositivi periferici di I/O), o superiore, poiché è interpretato. Inoltre la condivisione della CPU rallenta ulteriormente in modo imprevedibile le macchine virtuali. In un caso limite, per offrire una vera macchina virtuale, può essere necessario simulare tutte le istruzioni. Il sistema VM funziona su calcolatori IBM, poiché le normali istruzioni per le macchine virtuali sono eseguibili direttamente dalla macchina reale. Soltanto le istruzioni privilegiate, richieste soprattutto per le operazioni di I/O, devono essere simulate e quindi eseguite più lentamente.

Senza qualche supporto hardware la virtualizzazione non sarebbe possibile. Più supporto hardware fornirà un sistema, più le macchine virtuali saranno stabili, performanti e ricche di funzionalità. Le più diffuse CPU general-purpose offrono un supporto hardware dedicato alla virtualizzazione. La tecnologia di virtualizzazione AMD, presente su diversi processori AMD, ne è un esempio. Tale tecnologia definisce due nuove modalità per le operazioni: modalità host (ospitante) e modalità guest (ospitato). Il software per la macchina virtuale può abilitare la modalità host, definire le caratteristiche di ogni macchina virtuale e quindi cambiare la modalità in guest e passare il controllo del sistema al processo ospite in esecuzione sulla macchina virtuale. In modalità guest il sistema operativo virtualizzato crede di essere in esecuzione su hardware nativo e può vedere alcuni dei dispositivi (quelli inclusi nella definizione delle caratteristiche dell'ospite fatta da parte dell'ospitante). Se il processo ospite prova ad accedere a una risorsa il controllo passa all'ospitante, che renderà possibile l'interazione.

## 2.8.6 Esempi

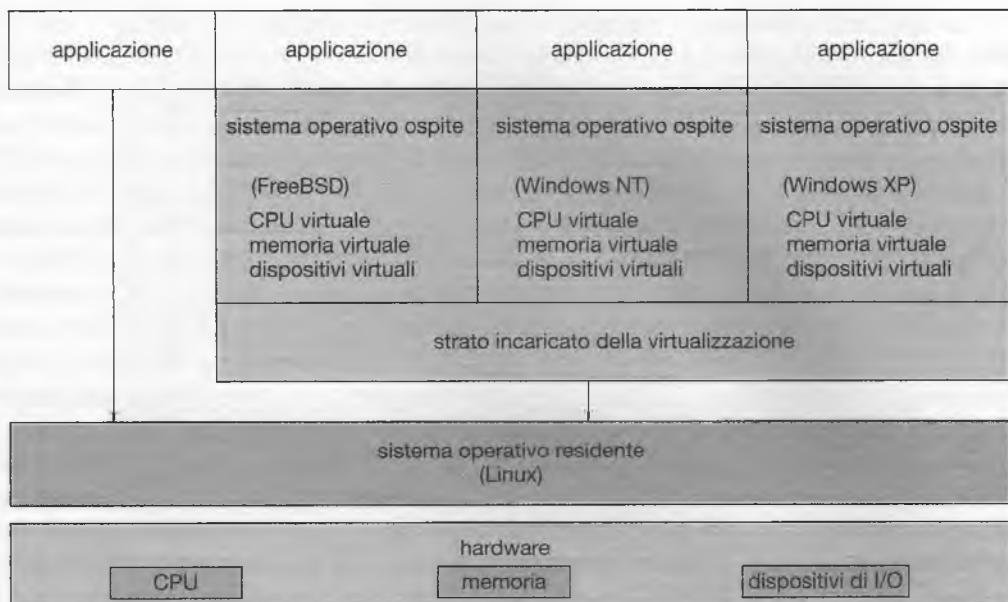
Nonostante i vantaggi offerti dalle macchine virtuali, dopo il loro primo sviluppo esse sono state oggetto di poche attenzioni per diversi anni. Ciononostante, oggi le macchine virtuali sono diventate di moda come strumento per risolvere problemi di compatibilità di sistemi. In questo paragrafo analizziamo due popolari macchine virtuali: VMware workstation e la Java virtual machine. Come vedremo, queste macchine virtuali possono essere solitamente eseguite su ognuno dei sistemi operativi descritti in precedenza. Le metodologie di disegno dei sistemi operativi (livelli semplici, microkernel, moduli e macchine virtuali) non sono quindi mutuamente esclusive.

### 2.8.6.1 VMware

La maggior parte delle tecniche di virtualizzazione discusse in questo paragrafo richiede il supporto del kernel alla virtualizzazione. Un altro metodo richiede che gli strumenti di virtualizzazione siano scritti per essere eseguiti in modalità utente, come applicazione eseguita dal sistema operativo. Le macchine virtuali in esecuzione tramite questi strumenti credono di essere in esecuzione su un hardware dedicato, ma in effetti sono eseguite all'interno di un'applicazione a livello utente.

VMware è una nota applicazione commerciale che converte l'apparato fisico Intel 80x86 in macchine virtuali distinte. VMware funge da applicazione in un sistema operativo ospitante, quale Windows o Linux, che può così installare alcuni sistemi operativi ospiti a titolo di macchine virtuali indipendenti.

Si può osservare l'architettura di un sistema siffatto nella Figura 2.19. In questo esempio, Linux è il sistema operativo residente, mentre FreeBSD, Windows NT e Windows XP sono i sistemi ospiti. Lo strato incaricato della virtualizzazione è il fulcro di VMware, poiché grazie a esso i dispositivi fisici sono trasformati in macchine virtuali a sé stanti che fungono



**Figura 2.19** Architettura VMware.

da sistemi operativi ospiti. Ciascuna macchina virtuale, oltre a possedere una CPU virtuale, può contare su elementi virtuali propri, quali la memoria, i drive del disco, le interfacce di rete e via di seguito.

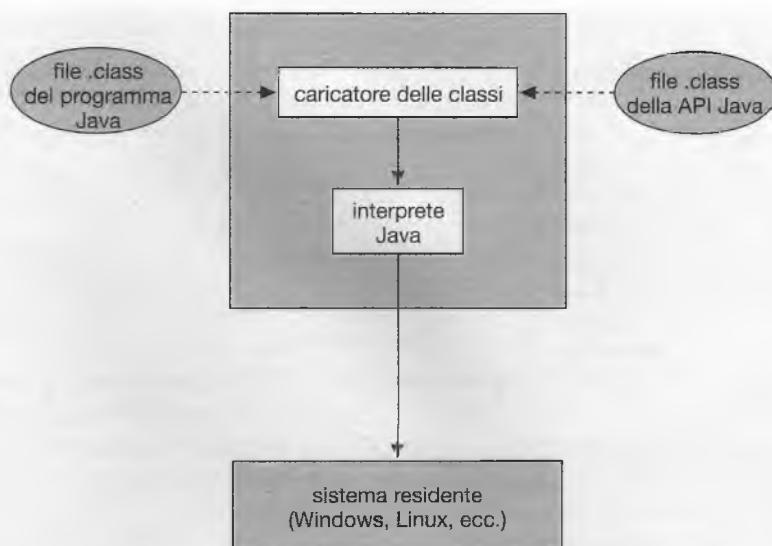
Il disco fisico di cui l'ospite dispone è in realtà semplicemente un file all'interno del file system del sistema operativo ospitante. Per creare un'istanza identica all'ospite è sufficiente copiare il file. Copiare il file in una nuova posizione protegge l'istanza dell'ospite da possibili danneggiamenti alla posizione originale. Spostando il file in una nuova posizione viene spostato il sistema ospite. Questi scenari mostrano come la virtualizzazione può effettivamente aumentare l'efficienza nell'amministrazione di un sistema e nell'uso delle risorse di sistema.

### 2.8.6.2 Macchina virtuale Java

Il linguaggio di programmazione Java, introdotto dalla Sun Microsystems alla fine del 1995, è un linguaggio orientato agli oggetti molto diffuso e fornisce, oltre a una descrizione analitica della sintassi del linguaggio e a una vasta libreria API, anche la definizione della **macchina virtuale Java** (*Java virtual machine*, JVM).

Gli oggetti si specificano con il costrutto **class** e un programma consiste di una o più classi. Per ognuna di queste, il compilatore produce un file (.class) contenente il cosiddetto *bytecode*; si tratta di codice nel linguaggio di macchina della JVM, indipendente dall'architettura soggiacente, che viene per l'appunto eseguito dalla JVM.

La JVM è un calcolatore astratto che consiste di un **caricatore delle classi** e di un interprete del linguaggio che esegue il bytecode (Figura 2.20). Il caricatore delle classi carica i file .class, sia del programma scritto in Java sia dalla libreria API, affinché l'interprete possa eseguirli. Dopo che una classe è stata caricata, il verificatore delle classi controlla la correttezza sintattica del codice bytecode, che il codice non produca accessi oltre i limiti della



**Figura 2.20** Macchina virtuale Java.

pila e che non esegua operazioni aritmetiche sui puntatori, che potrebbero generare accessi illegali alla memoria. Se il controllo ha un esito positivo, la classe viene eseguita dall'interprete. La JVM gestisce la memoria in modo automatico procedendo alla sua “ripulitura” (*garbage collection*) che consiste nel recupero delle aree della memoria assegnate a oggetti non più in uso per restituirla al sistema. Al fine di incrementare le prestazioni dei programmi eseguiti dalla macchina virtuale una notevole attività di ricerca è focalizzata allo studio degli algoritmi di ripulitura della memoria.

La JVM può essere implementata come software ospitato da un sistema operativo residente, per esempio Windows, Linux o Mac OS X, oppure all'interno di un browser web. In alternativa, può essere cablata in un circuito integrato espressamente progettato per l'esecuzione di programmi Java. Nel primo caso, l'interprete Java *interpreta* le istruzioni bytecode una alla volta. Una soluzione più efficiente consiste nell'uso di un **compilatore istantaneo** o **just-in-time (JIT)**. Alla prima invocazione di un metodo Java, il bytecode relativo è tradotto in linguaggio macchina comprensibile dalla macchina fisica ospitante. Il codice macchina relativo è poi salvato appropriatamente, in modo da essere direttamente riutilizzabile a una successiva invocazione del metodo Java, evitando la lenta interpretazione delle istruzioni bytecode. Una soluzione potenzialmente ancora più veloce è cablare la JVM in un circuito integrato, come detto poc'anzi, che esegua le istruzioni bytecode come codice macchina primitivo, eliminando del tutto la necessità di interpreti e compilatori.

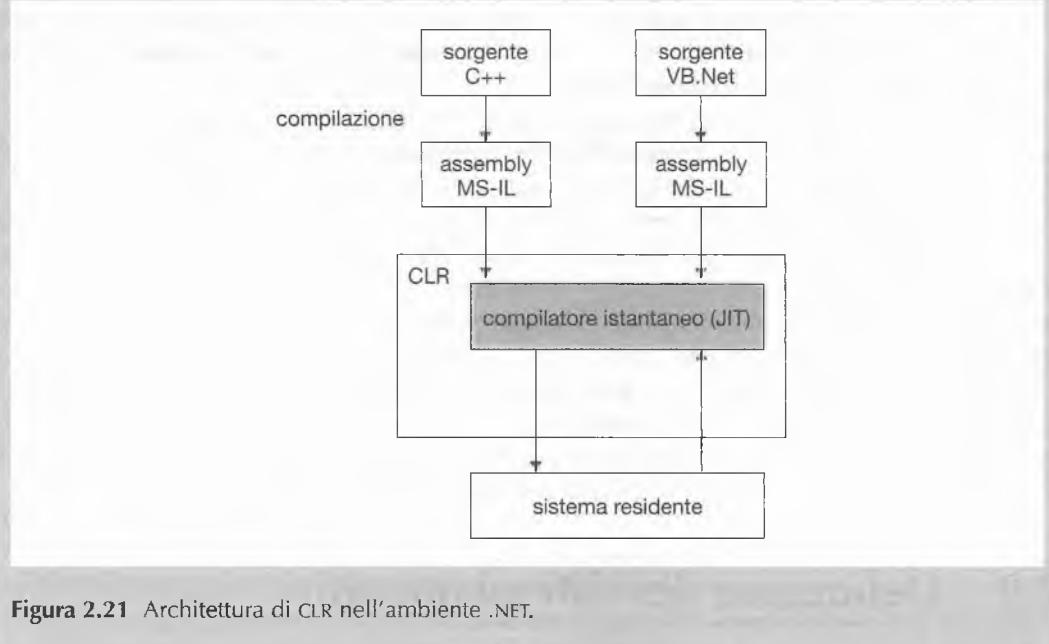
## 2.9 Debugging dei sistemi operativi

Il **debugging** può essere genericamente definito come l'attività di individuare e risolvere errori nel sistema, i cosiddetti *bachi* (*bugs*). Questa operazione viene effettuata sia sull'hardware sia sul software. I problemi che condizionano le prestazioni sono considerati bachi, quindi il debugging può comprendere anche una **regolazione delle prestazioni** (*performance tuning*), che ha lo scopo di migliorare le prestazioni eliminando i **colli di bottiglia** (*bottlenecks*).

## L'AMBIENTE .NET

L'ambiente .NET può essere definito come la sinergia tra un gruppo di risorse (per esempio, le librerie delle classi) e un ambiente di esecuzione che, combinati insieme, formano una piattaforma idonea allo sviluppo di programmi. La piattaforma permette di realizzare programmi concepiti espressamente per l'ambiente .NET anziché per altre specifiche architetture. Un programma scritto per .NET non deve preoccuparsi dei dettagli legati all'hardware o al sistema operativo che lo eseguirà. Infatti, ogni sistema che implementi .NET, sarà in grado di eseguire con successo il programma. Ciò è reso possibile dall'ambiente di esecuzione, che ricava i dettagli necessari e fornisce una macchina virtuale che opera come intermediario tra il programma in esecuzione e l'architettura sottostante.

Il Linguaggio Comune Runtime (CLR) è ciò che dà vita alla cornice .NET. Il CLR è l'implementazione della macchina virtuale .NET, che permette l'esecuzione di programmi scritti nei linguaggi compatibili con l'ambiente .NET. I programmi scritti con linguaggi come il C# (pronunciato, in inglese, "C-sharp") e il VB.NET sono compilati in un linguaggio intermedio indipendente dall'architettura, denominato Linguaggio Intermedio Microsoft (MS-IL). Questi file compilati, detti assemblati, comprendono le istruzioni MS-IL e i metadati. Hanno l'estensione .EXE oppure .DLL. Al momento dell'esecuzione di un programma, il CLR carica i file assemblati in ciò che è noto come **dominio dell'applicazione**. Mentre il programma in esecuzione richeiede le istruzioni, il CLR converte, tramite compilazione istantanea, le istruzioni MS-IL nei file assemblati nel codice nativo utilizzato dall'architettura sottostante. Una volta che le istruzioni siano state convertite in codice nativo, sono appropriatamente salvate e continueranno a essere eseguite, all'occorrenza, dalla CPU, in forma di codice macchina. L'architettura del CLR per la cornice .NET è rappresentata nella Figura 2.21.



**Figura 2.21** Architettura di CLR nell'ambiente .NET.

*neck)* che hanno luogo nei processi di sistema. In questo paragrafo tratteremo del debugging del kernel, degli errori di processo e dei problemi di prestazioni; il tema del debugging dell'hardware esula invece dagli scopi di questo testo.

### 2.9.1 Analisi dei guasti

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

Se il debugging di processi a livello utente è una sfida, a livello del kernel del sistema operativo esso è un'attività ancora più difficile a causa della dimensione e della complessità del kernel, del suo controllo dell'hardware e della mancanza di strumenti per eseguire il debugging a livello utente. Un guasto nel kernel viene chiamato **crash**. Come accade per i guasti dei processi, anche in questo caso dell'informazione riguardante l'errore viene salvata in un file di log, mentre lo stato della memoria viene salvato in un'immagine del contenuto della memoria al momento del crash (**crash dump**).

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

### 2.9.2 Regolazione delle prestazioni

Per identificare eventuali colli di bottiglia dobbiamo essere in grado di monitorare le prestazioni del sistema. A tale scopo deve essere presente del codice che esegua misurazioni sul comportamento del sistema e mostri i risultati. In molti casi il sistema operativo produce degli elenchi che tracciano il comportamento del sistema; tutti gli eventi di rilievo sono descritti indicandone l'ora e i parametri importanti e sono scritti in un file. Successivamente, un programma di analisi può esaminare il file di log al fine di determinare le prestazioni del sistema e di identificarne ostacoli e inefficienze. Le stesse tracce possono essere utilizzate come input per la simulazione di una miglioria al sistema operativo e inoltre possono contribuire alla scoperta di errori nel comportamento dello stesso.

#### LA LEGGE DI KERNIGHAN

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

Un altro approccio alla regolazione delle prestazioni consiste nell'includere nel sistema strumenti interattivi che permettano a utenti e amministratori di interrogare lo status dei vari componenti del sistema per individuare i colli di bottiglia. Il comando `top` in UNIX mostra le risorse di sistema impiegate, nonché un elenco ordinato dei principali processi che utilizzano le risorse. Altri strumenti mostrano lo stato del disco I/O, la memoria allocata e il traffico di rete. Gli autori di questi strumenti dotati di un'unica finalità provano a indovinare le necessità dell'utente che analizza il sistema e offrono queste informazioni.

Rendere i sistemi operativi esistenti più facili da comprendere e semplificare il debugging e la regolazione delle prestazioni sono un'area attiva della ricerca e dell'implementazione dei sistemi. Il ciclo che comincia con il tracciare i problemi riscontrati nel sistema e prosegue con l'analisi delle tracce sta per essere interrotto da una nuova generazione di strumenti adibiti all'analisi delle prestazioni del kernel. Questi strumenti non hanno un unico scopo né sono destinati esclusivamente a sezioni del codice scritte appositamente per produrre informazioni sul debugging. Solaris 10 DTrace è un'utilità per il tracciamento dinamico che costituisce un esempio rilevante di tali strumenti.

### 2.9.3 DTrace

DTrace è un'utilità che aggiunge dinamicamente delle sonde al sistema operativo, sia nei processi utente sia nel kernel. Queste sonde possono essere interrogate attraverso il linguaggio di programmazione D per determinare una quantità stupefacente di informazioni sul kernel, sullo status del sistema e sulle attività di processo. Ad esempio, nella Figura 2.22 si segue il comportamento di un'applicazione mentre esegue una chiamata di sistema (`ioctl`)

```
# ./all.d `pgrep xclock` XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued          U
  0 -> _XEventsQueued         U
  0 -> _X11TransBytesReadable U
  0 <- _X11TransBytesReadable U
  0 -> _X11TransSocketBytesReadable U
  0 <- _X11TransSocketBytesReadable U
  0 -> ioctl                  U
  0 -> ioctl                  K
  0 -> getf                  K
  0 -> set_active_fd          K
  0 <- set_active_fd          K
  0 <- getf                  K
  0 -> get_udatamodel        K
  0 <- get_udatamodel        K
...
  0 -> releasef              K
  0 -> clear_active_fd       K
  0 <- clear_active_fd       K
  0 -> cv_broadcast           K
  0 <- cv_broadcast           K
  0 <- releasef              K
  0 <- ioctl                 K
  0 <- ioctl                 U
  0 <- _XEventsQueued         U
  0 <- XEventsQueued          U
```

**Figura 2.22** DTrace, su Solaris 10, segue una chiamata di sistema all'interno del kernel.

e si mostrano le chiamate di funzioni all'interno del kernel che vengono eseguite per completare la chiamata di sistema. Le linee che terminano con "U" sono eseguite in modalità utente, quelle che terminano con "K" in modalità kernel. È pressoché impossibile eseguire il debugging dell'interazione tra livello utente e codice kernel senza avere a disposizione un insieme di strumenti che capiscano entrambi i tipi di codice e possano tenere traccia di questa interazione con strumenti appropriati.

Affinché un tale gruppo di strumenti risulti veramente utile deve essere in grado di fare il debugging di tutte le aree del sistema, incluse quelle scritte originariamente senza prendere in considerazione il debugging, e deve poterlo fare senza condizionare l'affidabilità del sistema. Questo strumento deve avere inoltre un impatto minimo sulle prestazioni: idealmente, non dovrebbe avere alcun impatto mentre non è in funzione e un impatto minimo durante l'utilizzo. L'utilità DTrace soddisfa questi requisiti offrendo uno strumento di debugging dinamico, sicuro e a basso impatto.

Il framework e gli strumenti DTrace furono disponibili a partire dal Solaris 10. Fino a quel momento, il debugging del kernel era una sorta di oggetto misterioso; lo si eseguiva tramite codice e strumenti arcaici e non sistematici. Ad esempio, i processori sono dotati di una funzionalità di breakpoint che ferma l'esecuzione e permette al debugger di esaminare lo stato del sistema. L'esecuzione può poi continuare fino al breakpoint o alla terminazione successivi. Questo metodo non può essere utilizzato in un kernel multiutente senza condizionare negativamente tutti gli utenti del sistema. La **profilazione** o **profiling**, che saggia periodicamente il puntatore alle istruzioni per verificare qual è il codice in esecuzione, mostra le tendenze statistiche, ma non le attività individuali. Può essere incluso nel kernel un codice destinato a produrre dati specifici in specifiche circostanze, ma quel codice rallenta il kernel e tende a non essere incluso in quella parte del kernel dove si è verificato il problema che ha richiesto il debugging.

Al contrario, DTrace funziona su sistemi di produzione (sistemi che stanno mettendo in funzione applicazioni importanti o critiche) e non è dannoso al sistema. Pur rallentando le attività quando è in esecuzione, dopo l'esecuzione riporta il sistema allo stato precedente il debugging. Si tratta inoltre di uno strumento che agisce ampiamente e in profondità, in quanto può eseguire il debugging di tutto ciò che sta accadendo nel sistema (a livello utente e a livello del kernel, comprese le interazioni tra i due livelli) e può scavare profondamente nel codice, mostrando le singole istruzioni del processore e le attività del kernel.

DTrace è composto da un compilatore, un framework, diversi **provider di sonde** (*providers of probes*), scritti all'interno di quel framework, e **consumer delle sonde**. I provider di DTrace creano le sonde, delle quali viene tenuta traccia in apposite strutture del kernel. Le sonde vengono immagazzinate in una tabella hash, dove sono suddivise per nome e indicizzate secondo identificatori univoci di sonde. Quando una sonda viene abilitata, una porzione di codice nell'area da sondare è riscritta per eseguire la chiamata `dtrace_probe` (*probe identifier*) per poi proseguire con il normale flusso del codice. Provider differenti danno origine a differenti tipi di sonde. Ad esempio, una sonda che esegue una chiamata di sistema del kernel lavora in modo differente da una sonda in un processo utente, che è a sua volta diversa da una sonda I/O.

DTrace fornisce un compilatore che genera un byte code eseguito nel kernel; il compilatore stesso garantisce la sicurezza del codice che ha creato. Ad esempio, non sono permessi cicli e vengono messe solo specifiche modifiche allo stato del kernel ed esclusivamente su richiesta. Solamente gli utenti di DTrace che godono di privilegi (ovvero gli amministratori, o *utenti root*) possono usare DTrace, dal momento che questo può recuperare dati privati (e modificarli se richiesto). Il codice generato funziona nel kernel e attiva le sonde, oltre ad attivare i consumer in modalità utente e a permettere la comunicazione tra i due.

Un consumer di DTrace è un codice interessato a una sonda e ai suoi risultati. Esso richiede al provider di creare una o più sonde. Quando una sonda si attiva, produce dei dati che sono gestiti dal kernel. Nel momento dell'attivazione all'interno del kernel vengono eseguite delle azioni di **abilitazione dei blocchi di controllo** (*enabling control blocks*, ECB). Una sonda può provocare l'esecuzione di diversi ECB se più consumer sono interessati a essa. Ogni ECB contiene un predicato ("if statement", ovvero istruzione if) che può o escludere l'ECB in questione, oppure eseguire la lista di azioni nell'ECB. L'azione più frequente è quella di catturare alcuni gruppi di dati, come il valore di una variabile a quel punto dell'esecuzione della sonda. Raccogliendo tali dati, può essere costruita un'immagine completa dell'azione dell'utente o del kernel. Inoltre, l'attivazione delle sonde dall'area utente e dal kernel può mostrare come un'azione a livello utente abbia causato reazioni a livello kernel. Questi dati hanno un valore inestimabile per il monitoraggio delle prestazioni e l'ottimizzazione del codice.

Una volta che il consumer delle sonde abbia terminato le sue operazioni vengono rimossi i rispettivi ECB. Nel caso in cui non ci siano ECB che utilizzano una sonda viene rimossa anche la sonda. Il codice viene quindi riscritto per rimuovere la chiamata `dtrace_probe` e ripristinare il codice originario. In questo modo il sistema è esattamente lo stesso di prima della creazione della sonda e dopo la sua distruzione è proprio come se l'attività di quella sonda non fosse mai esistita.

Per evitare danni al sistema, DTrace si preoccupa di fare in modo che le sonde non utilizzino troppa memoria né troppa capacità del processore. I buffer utilizzati per conservare i risultati delle analisi (le attività della sonda) vengono monitorati per verificare che non eccedano la dimensione massima consentita. Anche il tempo che il processore impiega a eseguire l'analisi è monitorato. Se si superano i limiti, il consumer e le sonde dannose vengono eliminate. Per evitare conflitti e perdite di dati, si allocano buffer dedicati per ogni processore.

Un esempio di codice D e del suo output ne illustra alcuni vantaggi. Il programma seguente mostra il codice DTrace che attiva sonde dello scheduler e registra il tempo di CPU utilizzato dai processi aventi ID utente 101 mentre le sonde sono attive (ossia mentre il programma è in esecuzione):

```

sched:::on-cpu
uid == 101
{
    self->ts = timestamp;
}

sched:::off-cpu
self->ts
{
    @time[execname] = sum(timestamp - self->ts);
    self->ts = 0 ;
}

```

La Figura 2.23 mostra il risultato del programma, cioè quali sono i processi e per quanto tempo (in nanosecondi) occupano il processore.

Poiché DTrace fa parte del sistema operativo open-source Solaris 10, lo si può incorporare in altri sistemi operativi qualora non vi siano conflitti fra le licenze. Ad esempio,

```
# dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
'C
      gnome-setting-d          142354
      gnome-vfs-deamon         158243
      dsdm                      189804
      wnck-applet                200030
      gnome-panel                 277864
      clock-applet                374916
      mapping-deamon              385475
      xscreensaver                  514177
      metacity                     539281
      Xorg                         2579646
      gnome-terminal                5007269
      mixer_applet2                7388447
      java                        10769137
```

**Figura 2.23** Risultato del codice D.

DTrace è già stato aggiunto a Mac OS X 10.5 e a FreeBSD. È probabile che esso si diffonderà ulteriormente in futuro per merito delle sue capacità uniche. Anche altri sistemi operativi, in special modo i derivati da Linux, si stanno attrezzando per incorporare funzionalità di tracciamento del kernel. Altri sistemi stanno cominciando a includere strumenti di monitoraggio delle prestazioni e di tracciamento promossi dalla ricerca in vari istituti, come nel caso del progetto Paradyn.

## 2.10 Generazione di sistemi operativi

Un sistema operativo si può progettare, codificare e realizzare specificamente per una singola macchina; tuttavia, è più diffusa la pratica di progettare sistemi operativi da impiegare in macchine di una stessa classe con configurazioni diverse. Il sistema si deve quindi configurare o generare per ciascuna situazione specifica, un processo talvolta noto come **generazione di sistemi** (SYSGEN).

I sistemi operativi sono normalmente distribuiti per mezzo di dischi o CD-ROM. Per generare un sistema è necessario usare un programma speciale che può leggere da un file o richiedere all'operatore le informazioni riguardanti la configurazione specifica del sistema o anche esplorare il sistema di calcolo per determinarne i componenti. A questo scopo sono necessarie le seguenti informazioni.

- ◆ La CPU che si deve impiegare e le opzioni installate (serie di istruzioni estese, aritmetica in virgola mobile, e così via). Nel caso di sistemi multiprocessore occorre anche descrivere ciascuna di loro.
- ◆ Come verrà formattato il disco d'avvio? In quante sezioni o “partizioni” verrà suddiviso e che cosa ci sarà in ciascuna partizione?
- ◆ La quantità di memoria disponibile. Alcuni sistemi determinano autonomamente questi valori, accedendo a tutte le locazioni della memoria, fino alla generazione di un errore di indirizzo illegale. Questa procedura definisce l'indirizzo legale finale e quindi la quantità di memoria disponibile.

- ◆ I dispositivi disponibili. Il sistema deve conoscere gli indirizzi di ogni dispositivo e sapere come farvi riferimento (numero di dispositivo), deve conoscere il numero del segnale d'interruzione del dispositivo, il tipo e il modello del dispositivo e tutte le sue caratteristiche specifiche.
- ◆ Le opzioni del sistema operativo richieste o i valori dei parametri che è necessario usare. Queste informazioni possono contenere il numero delle aree di memoria da usare per le operazioni di I/O e la loro dimensione, l'algoritmo di scheduling della CPU richiesto, il numero massimo di processi da sostenere, e così via.

Una volta ottenute, queste informazioni si possono impiegare in modi diversi. In un caso limite, un amministratore di sistema le potrebbe usare per modificare una copia del codice sorgente del sistema operativo, che si dovrebbe poi ricompilare. Dichiarazioni di dati, inizializzazioni e costanti, insieme con una compilazione condizionale, produrrebbero una versione in codice di macchina del sistema operativo specifica per il sistema desiderato.

Per ottenere una versione meno specifica ma comunque adatta al sistema desiderato, la descrizione del sistema può determinare la creazione di tabelle e la selezione di moduli da una libreria precompilata, che si collegano per formare il sistema operativo richiesto. La selezione permette alla libreria di contenere i driver di tutti i dispositivi di I/O previsti, sebbene solo quelli effettivamente necessari siano inclusi nel sistema operativo richiesto. Poiché non si ricompila il sistema, la sua generazione risulta più rapida, ma il sistema ottenuto può essere inutilmente generale.

Un altro caso limite è rappresentato dalla costruzione di un sistema completamente controllato mediante tabelle. In questo caso tutto il codice è sempre parte del sistema e la selezione si effettua al momento dell'esecuzione del sistema stesso, anziché nella fase della compilazione o del collegamento. La generazione del sistema implica semplicemente la creazione di tabelle idonee a descrivere il sistema stesso. Le differenze più rilevanti tra questi metodi riguardano la dimensione e la generalità del sistema ottenuto, oltre alla facilità di apportarvi modifiche in seguito a cambiamenti della sua struttura fisica. Si consideri, per esempio, il costo delle modifiche da apportare al sistema per consentirgli la gestione di un nuovo terminale grafico o di un'altra unità a disco. I costi vanno ovviamente bilanciati con la frequenza delle modifiche.

## 2.11 Avvio del sistema

Dopo che un sistema operativo è stato scritto, bisogna predisporlo all'uso da parte dei dispositivi fisici. Ma come fa l'apparato fisico dell'elaboratore a sapere dove si trova il kernel e a caricarlo? La procedura d'avviamento di un calcolatore attraverso il caricamento del kernel è nota come **avviamento** (*booting*) del sistema: nella maggior parte dei sistemi di calcolo c'è un piccolo segmento di codice, noto come **programma d'avvio** (*bootstrap program*) o **caricatore d'avvio** (*bootstrap loader*), che individua il kernel, lo carica in memoria e ne avvia l'esecuzione. Alcuni sistemi, come i PC, eseguono tale compito in due fasi: un caricatore d'avvio molto semplice preleva dal disco un più complesso programma d'avvio, che a sua volta carica il kernel.

Quando una CPU sta per entrare in funzione – per esempio, quando l'elaboratore viene acceso o riavviato – il registro delle istruzioni è caricato con una locazione di memoria predefinita, da cui ha inizio l'esecuzione. Il programma di avvio inizia da questa locazione.

Esso è contenuto in una **memoria a sola lettura** (**read-only memory**, ROM), poiché non si conosce lo stato della RAM all'avvio del sistema; inoltre, la ROM presenta il vantaggio di non dover essere inizializzata e di essere immune ai virus.

Il programma di avvio può effettuare operazioni di vario genere. Una di queste, solitamente, sottopone a diagnosi la macchina per ottenere informazioni sul suo stato. Se la diagnostica dà esito positivo, il programma è in grado di proseguire con le altre fasi di avvio. Esso può anche inizializzare il sistema in ogni suo elemento, dai registri della CPU ai controllori dei dispositivi, fino ai contenuti della memoria centrale. Presto o tardi, comunque, farà partire il sistema operativo.

Alcuni sistemi, come i telefoni cellulari, i PDA e le console per videogiochi, memorizzano l'intero sistema operativo nella ROM. La scelta di custodire nella ROM il sistema operativo si addice a sistemi di piccole dimensioni, con dispositivi fisici modesti e un funzionamento tutt'altro che sofisticato. Questa soluzione comporta un problema, cioè la necessità di modificare i circuiti ROM al fine di poter modificare il codice del programma di avvio. Alcuni sistemi ovviano a questo inconveniente utilizzando la **memoria a sola lettura programmabile e cancellabile** (EPROM), che è appunto a sola lettura, ma può diventare riscrivibile qualora riceva un comando apposito. Tutte le forme di ROM sono anche dette **firmware**, in considerazione delle loro caratteristiche, che sono un ibrido tra hardware e software. Un problema sollevato dal firmware, in genere, concerne l'esecuzione del codice, più lenta di quanto avvenga con la RAM. Taluni sistemi memorizzano il sistema operativo nel firmware e lo copiano nella RAM per eseguirlo rapidamente. Ancora, una pecca del firmware è il suo essere relativamente costoso, una circostanza per cui, di solito, è disponibile in piccole quantità.

Per sistemi operativi di grandi dimensioni (tra cui Windows, Mac OS X, UNIX e quasi tutti quelli a carattere generale) o per sistemi che cambiano di frequente, il caricatore di avvio è memorizzato nel firmware e il sistema operativo risiede su disco. In questo caso, il programma di avvio applica gli strumenti di diagnosi e utilizza una parte di codice per la lettura di un blocco singolo che occupa una locazione fissa del disco (per esempio, il blocco zero); quindi, lo trasferisce in memoria per eseguire il codice da quel **blocco di avvio** (*boot block*). Il programma custodito dal blocco di avvio è a volte abbastanza complesso per caricare in memoria l'intero sistema operativo e dare avvio alla sua esecuzione. Più spesso, è un programma semplice (deve risiedere in un singolo blocco del disco) che conosce unicamente la lunghezza e l'indirizzo sul disco del codice residuo di cui è composto l'intero programma d'avvio. Tutto il codice di avviamento destinato al disco, e il sistema operativo stesso, può essere sostituito facilmente, scrivendone nuove versioni sul disco. Un disco che contenga una partizione di avvio è chiamato **disco di avvio** (*boot disk*) o **disco di sistema**; si veda il Paragrafo 12.5.1 sull'argomento.

Una volta caricato il programma di avvio completo, esso può addentrarsi nel file system per localizzare il kernel, così da caricarlo in memoria e dare inizio alla sua esecuzione. È solo a questo punto che il sistema può essere considerato in funzione (**running**).

## 2.12 Sommario

I sistemi operativi offrono diversi servizi: al livello più basso, le chiamate di sistema permettono al programma in esecuzione di fare richieste direttamente al sistema operativo; a un livello superiore, l'interprete dei comandi (in alcuni ambiti noto come *shell*) mette a disposizione un meccanismo che consente a un utente di impartire una richiesta senza scrivere un

programma. I comandi possono provenire da file, in un'esecuzione a lotti (*batch*) oppure direttamente da una tastiera, in modo interattivo o a partizione del tempo. I programmi di sistema offrono agli utenti i servizi più comuni.

I tipi di richieste variano secondo il livello delle stesse richieste. Il livello cui appartengono le chiamate di sistema deve offrire le funzioni di base, come quelle di controllo dei processi e gestione di file e dispositivi. Le richieste di livello superiore, soddisfatte dall'interprete dei comandi o dai programmi di sistema, sono tradotte in una sequenza di chiamate di sistema. I servizi di sistema si possono classificare in diverse categorie: controllo dei programmi, richieste di stato e richieste di I/O. Gli errori dei programmi si possono considerare richieste di servizio implicite.

Una volta definiti i servizi del sistema è possibile passare allo sviluppo della struttura del sistema operativo. Per registrare le informazioni che definiscono lo stato del calcolatore e lo stato dei processi del sistema occorrono diverse tabelle.

La progettazione di un nuovo sistema operativo è un compito molto difficile. Gli scopi del sistema si devono definire chiaramente prima di iniziare la progettazione; costituiscono la base da cui partire per poter scegliere tra le varie strategie e i vari algoritmi necessari.

Poiché un sistema operativo è di grandi dimensioni, la modularità è un altro fattore importante. La progettazione di un sistema come una sequenza di strati o l'uso di un microkernel sono considerati buone tecniche. Il concetto di macchina virtuale tiene in grande considerazione il metodo basato sulla stratificazione e tratta il kernel del sistema operativo come se facesse parte della macchina fisica. Su questa macchina virtuale si possono caricare persino altri sistemi operativi.

In tutto il ciclo di progettazione del sistema operativo, occorre prestare attenzione alla distinzione tra la scelta dei criteri e i dettagli dei meccanismi adottati. In questo modo si ottiene la massima flessibilità, che all'occorrenza consente di modificare più facilmente i criteri adottati.

Ormai i sistemi operativi sono quasi tutti scritti in un linguaggio per lo sviluppo di sistemi o in un linguaggio di alto livello; questa caratteristica facilita realizzazione, manutenzione e adattabilità a sistemi diversi.

Il processo di debugging e i guasti nel kernel possono essere studiati grazie all'utilizzo di debugger e di altri strumenti in grado di analizzare un'immagine dello stato della memoria. Strumenti quali DTrace analizzano i sistemi di produzione per trovare colli di bottiglia e capire altri comportamenti del sistema.

All'avvio di un calcolatore, la CPU deve eseguire il programma d'avvio residente nel firmware. Se l'intero sistema operativo risiede nel firmware, all'accensione l'intero sistema è eseguibile direttamente; altrimenti, il ciclo di avvio della macchina procede per fasi progressive, a ognuna delle quali si caricano in memoria, dal firmware e dal disco, porzioni sempre più potenti del sistema operativo, fino a eseguire l'intero sistema stesso.

## Esercizi pratici

- 2.1 Qual è lo scopo delle chiamate di sistema?
- 2.2 Quali sono le cinque attività principali di un sistema operativo dal punto di vista della gestione dei processi?
- 2.3 Quali sono le tre attività principali di un sistema operativo dal punto di vista della gestione della memoria?
- 2.4 Quali sono le tre attività principali di un sistema operativo dal punto di vista della memoria secondaria?

- 
- 2.5 Qual è lo scopo dell'interprete dei comandi? Perché è solitamente separato dal kernel?
  - 2.6 Quali chiamate di sistema devono essere eseguite dall'interprete dei comandi, o shell, per avviare un nuovo processo?
  - 2.7 Qual è lo scopo dei programmi di sistema?
  - 2.8 Qual è il vantaggio principale dell'approccio a strati (layer) all'architettura di sistema? Quali sono invece i suoi svantaggi?
  - 2.9 Elencate cinque servizi forniti da un sistema operativo e spiegate la convenienza per l'utente di ciascuno. In quali casi sarebbe impossibile per i programmi a livello utente offrire questi servizi? Argomentate la vostra risposta.
  - 2.10 Perché alcuni sistemi memorizzano il sistema operativo nel firmware mentre altri lo memorizzano su disco?
  - 2.11 Come potrebbe essere progettato un sistema perché offra la possibilità di scegliere quale sistema operativo avviare? Che cosa dovrebbe fare in questo caso il bootstrap?

## Esercizi

- 2.12 I servizi e le funzioni offerti da un sistema operativo possono essere divisi in due categorie. Procedete a una loro breve descrizione, analizzandone le differenze.
- 2.13 Descrivete tre metodi generali per passare parametri al sistema operativo.
- 2.14 Descrivete come si possa ottenere un profilo statistico del tempo consumato da un programma per eseguire le differenti parti del proprio codice. Argomentate l'importanza di simili profili statistici.
- 2.15 Quali sono le cinque attività principali di un sistema operativo relative alla gestione dei file?
- 2.16 Quali sono i vantaggi e gli svantaggi di usare la medesima interfaccia alle chiamate di sistema sia per i file sia per i dispositivi?
- 2.17 Sarebbe possibile per l'utente sviluppare un nuovo interprete dei comandi utilizzando le chiamate di sistema offerte dal sistema operativo?
- 2.18 Quali sono i due modelli della comunicazione tra processi? Quali i loro punti di forza e di debolezza?
- 2.19 Perché è auspicabile separare i meccanismi dai criteri o politiche?
- 2.20 Talvolta è difficile realizzare un'architettura a strati se due componenti del sistema operativo dipendono l'uno dall'altro. Identificate una situazione in cui non risulti immediatamente evidente come stratificare due componenti del sistema e nel contempo mantenere strettamente connesse le rispettive funzionalità.
- 2.21 Quale vantaggio si riscontra nell'architettura orientata al microkernel? In che modo interagiscono i programmi utenti e i servizi del sistema in tale architettura? Quali sono gli svantaggi?
- 2.22 Per quali versi la strategia del kernel modulare è simile alla strategia stratificata? Per quali aspetti la prima si differenzia dalla seconda?

- 2.23 Dite qual è il vantaggio principale che ottengono i progettisti di un sistema operativo che impiega un'architettura a macchine virtuali, e qual è il vantaggio principale per gli utenti.
- 2.24 Spiegate perché un compilatore istantaneo (*just-in-time*) è utile per l'esecuzione dei programmi scritti in Java.
- 2.25 Che tipo di relazione sussiste tra un sistema operativo ospite e un sistema operativo residente in un ambiente quale VMware? Quali fattori devono essere valutati nella scelta del sistema operativo residente?
- 2.26 Il sistema operativo sperimentale Synthesis ha un assemblatore incorporato nel kernel. Per ottimizzare le prestazioni delle chiamate di sistema, il kernel assembla le procedure nello spazio del kernel, al fine di ridurre al minimo il percorso che le chiamate di sistema devono seguire attraverso il kernel. Tale metodo è in antitesi al metodo stratificato che, pur rendendo più semplice la costruzione di un sistema operativo, determina un prolungamento del percorso delle chiamate di sistema attraverso il kernel. Valutate i pro e i contro di tale metodo nella progettazione di un kernel e nell'ottimizzazione delle prestazioni di un sistema.

## Problemi di programmazione

- 2.27 Nel Paragrafo 2.3 si è descritto un programma che copia i contenuti di un file di origine in un file di destinazione. Questo programma esordisce con la richiesta, indirizzata all'utente, dei nomi dei file di origine e di destinazione. Si scriva un tale programma usando Windows 32 o mediante la API del POSIX. Prestate particolare attenzione alla gestione degli errori, assicurandovi che il file di origine esista. Fatto ciò, eseguite il programma insieme a un'applicazione per la tracciatura delle chiamate di sistema, se si dispone di un sistema operativo con tale funzionalità. In ambiente Linux è disponibile l'applicazione `ptrace`, mentre i sistemi Solaris ricorrono ai comandi `truss` o `dtrace`. Una funzionalità simile è fornita, per il Mac OS X, dall'istruzione `ktrace`. Dato che i sistemi Windows non offrono queste caratteristiche, è necessario eseguire il tracciamento della versione Win32 del programma utilizzando un debugger.

## Progetti di programmazione

### 2.28 Introduzione di una chiamata di sistema nel kernel di Linux

In questo progetto si studierà l'interfaccia alle chiamate di sistema fornita da Linux, analizzando come i programmi utenti, attraverso questa interfaccia, possano comunicare con il kernel del sistema operativo. Il compito del lettore è incorporare una nuova chiamata di sistema all'interno del kernel, aumentando così le funzionalità del sistema operativo.

#### Parte 1: Per cominciare

Una chiamata di procedura in modalità utente si implementa passando gli argomenti alla funzione chiamata o tramite la pila (*stack*) oppure tramite registri, salvando lo stato corrente e il valore nel registro contatore di programma (*program counter*) e saltando all'inizio del codice che corrisponde alla procedura chiamata. Il processo continua a mantenere i privilegi di cui godeva in precedenza.

Le chiamate di sistema appaiono come chiamate di procedura rivolte a programmi utenti, ma finiscono per modificare i privilegi, determinando un diverso contesto di esecuzione. Se Linux gira su Intel 386, l'attuazione di una chiamata di sistema consiste nel memorizzare il suo numero identificativo nel registro EAX, collocare gli argomenti della chiamata in altri registri, e nell'eseguire un'istruzione di eccezione (che è l'istruzione 0x80 dell'assembly INT). Una volta sollevata l'eccezione, il numero della chiamata di sistema è usato come indice in una tabella di puntatori al codice, per ottenere l'indirizzo iniziale del codice che implementa il gestore della chiamata di sistema. Il processo, allora, salta a questo indirizzo, e i suoi privilegi subiscono una mutazione, dalla modalità utente alla modalità di sistema. In seguito all'ampliamento dei privilegi, il processo può ora eseguire il codice del kernel, che potrebbe contenere istruzioni privilegiate altrimenti non eseguibili. Il codice del kernel può dunque attuare i servizi richiesti come, per esempio, l'interazione con i dispositivi di I/O e la gestione dei processi, e può svolgere altre attività di questo genere che, nella modalità utente, sarebbero precluse.

I numeri delle chiamate di sistema relativi alle ultime versioni del kernel Linux sono riportati in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (Per esempio `__NR_close`, che corrisponde alla chiamata di sistema `close()` per la chiusura di un descrittore di file, ha il numero 6.) L'elenco dei puntatori ai gestori delle chiamate di sistema è memorizzato, in genere, nel file `/usr/src/linux-2.x/arch/i386/kernel/entry.S` sotto l'intestazione `ENTRY(sys_call_table)`. Si noti come `sys_close` sia memorizzato nella posizione numero 6 della tabella per essere coerente con il numero della chiamata di sistema definito nel file `unistd.h` (La parola chiave `.long` denota che l'elemento occuperà lo stesso numero di byte di un dato di tipo `long`.)

## Parte 2: Costruzione di un nuovo kernel

Prima di poter aggiungere una chiamata di sistema al kernel, il lettore dovrà acquisire dimensione con la traduzione del codice sorgente in codice binario, e con la procedura di riavvio della macchina con il nuovo kernel. Queste attività prevedono le seguenti operazioni, alcune delle quali dipendono dalla specifica distribuzione di Linux.

- ◆ Procurarsi il sorgente del kernel per la distribuzione di Linux in questione. Se il pacchetto con il codice sorgente è stato già installato sulla macchina del lettore, i file corrispondenti potrebbero essere disponibili in `/usr/src/linux` o in `/usr/src/linux-2.x` (dove il suffisso identifica la versione del kernel). Qualora il pacchetto non sia stato precedentemente installato, può essere scaricato dal fornitore della vostra versione di Linux o da: <http://www.kernel.org>.
- ◆ Apprendere come si configura, compila e installa il codice binario del kernel. A causa delle differenze tra le varie versioni, queste operazioni non sono generalizzabili, ma alcuni tipici comandi per costruire il kernel (dalla directory in cui si trova il codice sorgente) sono:
  - ◊ `make xconfig`
  - ◊ `make dep`
  - ◊ `make bzImage`
- ◆ Aggiungere un nuovo elemento alla serie di kernel avviabili presenti nel sistema. In genere il sistema operativo Linux si avvale di istruzioni quali `lilo` e `grub` per mantenere una lista di kernel avviabili, da cui l'utente può selezionare durante la fase di avvio della macchina. Se il vostro sistema prevede `lilo`, aggiungete un elemento a `lilo.conf`, quale:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

dove `/boot/bzImage.mykernel` è l'immagine del kernel, mentre `mykernel` è il nome simbolico associato al nuovo kernel, da selezionare durante la fase di avviamento. In questo modo si avrà la possibilità di avviare il nuovo kernel o, in alternativa, il vecchio kernel non sottoposto a modifiche, nel caso in cui il nuovo non funzioni come dovrebbe.

### Parte 3: Estensione del sorgente del kernel

A questo punto il lettore potrà sperimentare l'inserimento di un nuovo file nel gruppo dei file sorgente utilizzati per la compilazione del kernel. Il codice sorgente è spesso memorizzato nella directory `/usr/src/linux-2.x/kernel`, sebbene questa collocazione potrebbe cambiare da una versione all'altra di Linux. Per aggiungere una chiamata di sistema vi sono due opzioni. Una è di annettere la chiamata a un file sorgente preesistente di questa directory. L'altra consiste nella creazione di un file inedito nella directory, modificando `/usr/src/linux-2.x/kernel/Makefile` di modo che includa questo nuovo file nel processo di compilazione. Il vantaggio della prima soluzione è che, agendo su un file esistente, e dunque già incluso nel processo di compilazione, non è necessario modificare `Makefile`.

### Parte 4: Introduzione di una chiamata di sistema al kernel

Dopo aver preso confidenza con le varie operazioni propedeutiche alla costruzione e all'avvio dei kernel in ambiente Linux, il lettore potrà ora dedicarsi all'introduzione di una nuova chiamata di sistema nel kernel di Linux. In questo progetto la chiamata di sistema avrà funzionalità contenute; si limiterà semplicemente a passare dalla modalità utente a quella di sistema, stampare un messaggio che viene conservato nell'apposito spazio previsto dal kernel, e ritornare quindi alla modalità utente. Chiameremo questa chiamata di sistema `hello-world`. Pur avendo effetti limitati, essa illustra la dinamica delle chiamate di sistema, facendo luce sull'interazione tra i programmi utenti e il kernel.

- Create un nuovo file denominato `helloworld.c` per definire la chiamata di sistema. Includete i file di intestazione `linux/linkage.h` e `linux/kernel.h`. Si aggiunga a questo file il codice seguente:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
    printk(KERN_EMERG "hello world!");
    return 1;
}
```

Otterrete come risultato una chiamata di sistema dal nome `sys_helloworld()`. Se decidete di aggregare questa chiamata di sistema a un file già esistente nella directory con il codice sorgente, sarà sufficiente aggiungere la funzione `sys_helloworld()` al file selezionato.

to. La presenza di `asmlinkage`, risalente all'epoca in cui Linux adoperava non solo il C++ ma anche il C, segnala che il codice è scritto in C. La funzione `printf()` serve per stampare messaggi su un file di log gestito dal kernel, e pertanto può unicamente essere invocata dal kernel stesso. I messaggi del kernel specificati nel parametro di `printf()` sono registrati nel file di log `/var/log/kernel/warnings`. Il prototipo della funzione `printf()` è definito in `/usr/include/linux/kernel.h`.

- ◆ Definite un nuovo numero della chiamata di sistema per `__NR_helloworld` in `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Un programma utente può servirsi di tale numero per identificare la nuova chiamata inserita. Accertatevi, inoltre, di aumentare di uno il valore di `__NR_syscalls`, anch'esso custodito nello stesso file, e volto a tenere traccia del numero di chiamate di sistema presenti nel kernel.
- ◆ Introducete l'elemento `.long sys_helloworld.c` nella `sys_call_table` definita nel file `/usr/src/linux-2.x/arch/i386/kernel/entry.S`. Come già chiarito, il numero della chiamata di sistema funge da indice in una tabella per individuare la posizione del codice gestore della chiamata di sistema invocata.
- ◆ Aggiungete il vostro file `helloworld.c` al `Makefile` (qualora si sia creato un nuovo file per la chiamata di sistema). Salvate una copia dell'immagine binaria del vecchio kernel (come precauzione per eventuali problemi con il nuovo). Procedete ora alla costruzione del nuovo kernel; rinominatelo per distinguerlo dal kernel originario e aggiungete una posizione ai file di configurazione del caricatore (per esempio, a `lilo.conf`). Dopo aver completato questi passaggi, si potrà finalmente avviare o il kernel originario oppure il nuovo, che ospita la nuova chiamata di sistema.

## Parte 5: Uso della chiamata di sistema da un programma utente

All'avvio della macchina con il nuovo kernel, la nuova chiamata di sistema sarà pronta all'uso; ora si tratta semplicemente di invocarla da un programma utente. La libreria standard del C prevede normalmente un'interfaccia alle chiamate di sistema Linux. Poiché la nuova funzione non è collegata alla libreria standard del C, per invocare la chiamata di sistema si dovrà ricorrere a un intervento manuale.

Come notavamo prima, invocare una chiamata di sistema equivale a memorizzare il valore appropriato in un registro hardware e sollevare un'eccezione. Queste operazioni di basso livello, tuttavia, non si prestano a essere compiute con la sintassi del linguaggio C, richiedendo invece istruzioni assembly. Fortunatamente Linux dispone di macro per istanziare funzioni che racchiudono le istruzioni assembly appropriate. Per esempio, il seguente programma in C utilizza la macro `_syscall0()` per invocare la nuova chiamata di sistema:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>
_syscall0(int, helloworld);
main()
{
    helloworld();
}
```

- ◆ La macro `_syscall0` accetta due argomenti. Il primo specifica il tipo del valore restituito dalla chiamata di sistema; il secondo è il nome della chiamata di sistema. Il nome viene utilizzato per identificare il numero della chiamata di sistema, memorizzato nel registro prima che sia sollevata l'eccezione.

Se la chiamata di sistema richiede più argomenti, si potrà impiegare una macro differente (come `_syscall11`, il cui suffisso indica il numero di argomenti) per istanziare il codice assembly necessario a effettuare la chiamata di sistema.

- ◆ Compilate ed eseguite il programma con il kernel di nuova creazione. Il messaggio “`hello, world!`” nel file di log `/var/log/kernel/warnings` del kernel indicherà l'avvenuta esecuzione della chiamata di sistema.

Come possibile prosegua del progetto, si rifletta su come ampliare le funzionalità della nuova chiamata di sistema. Come si potrebbe passare un valore intero o una costante carattere alla chiamata di sistema per farlo successivamente stampare nel file di log del kernel? Quali implicazioni comporterebbe il passaggio di puntatori ai dati memorizzati nello spazio degli indirizzi del programma utente rispetto al passaggio di un semplice valore intero dal programma utente al kernel usando i registri hardware?

## 2.13 Note bibliografiche

L'orientamento stratificato alla progettazione dei sistemi operativi è stato sostenuto da [Dijkstra 1968]. [Brinch-Hansen 1970] è stato uno dei primi sostenitori della costruzione di un sistema operativo intorno a un nucleo (il kernel), sulla base del quale sviluppare sistemi più completi.

La strumentazione del sistema e la tracciatura dinamica sono descritti in [Tarnches e Miller 1999]. DTrace è trattato in [Cantrill et al. 2004]. [Cheung e Loong 1995] affrontano la questione della strutturazione di un sistema operativo, dal microkernel ai sistemi estendibili.

L'MS-DOS, Versione 3.1, è descritto in [Microsoft 1986]. Windows NT e Windows 2000 sono esaminati in [Solomon 1998], nonché in [Solomon e Russinovich 2000]. Il Berkeley UNIX (BSD) è descritto in [McKusick et al. 1996]. [Bovet e Cesati 2002] dissertano ampiamente sul kernel di Linux. Diversi sistemi UNIX – tra cui Mach – sono esaminati in dettaglio da [Vahalia 1996]. Il Mac OS X è presentato all'indirizzo <http://www.apple.com/macosx>. Il sistema sperimentale Synthesis è presentato in [Massalin e Pu 1989]. Solaris è dettagliatamente descritto da [Mauro e McDougall 2001].

Il primo sistema operativo a offrire una macchina virtuale è stato il CP/67 su un IBM 360/67. Il sistema operativo IBM VM/370 disponibile in commercio deriva dal CP/67. Dettagli sul sistema operativo Mach, basato su microkernel, sono reperibili in [Young et al. 1987]. [Kaashoek et al. 1997] riportano in merito ai sistemi operativi basati sul cosiddetto esokernel, la cui architettura tiene separate le questioni di protezione dalla gestione ordinaria, tollerando così il controllo indiscriminato delle risorse da parte di programmi inaffidabili.

Le specifiche del linguaggio Java e della relativa macchina virtuale sono presentate in [Gosling et al. 1996] e in [Lindholm e Yellin 1999], rispettivamente. Il funzionamento interno della JVM è descritto in [Venners 1998]. [Golm et al. 2002] si soffermano sul sistema operativo JX; [Back et al. 2000] evidenziano numerose tematiche di rilievo per la progettazione dei sistemi operativi Java. Ulteriori informazioni sul linguaggio Java sono disponibili all'indirizzo <http://www.javasoft.com>. Un approfondimento sull'implementazione di VMware è reperibile in [Sugerman et al. 2001]. All'indirizzo <http://www.vmware.com/appliances/learn.ovf.html> si possono trovare informazioni sul progetto Open Virtual Machine Format.

## Seconda parte

# Gestione dei processi

Un *processo* si può pensare come un programma in esecuzione. Per svolgere il proprio compito, un processo richiede determinate risorse, come tempo d'elaborazione della CPU, memoria, file e dispositivi di I/O. Queste risorse si assegnano al processo al momento della sua creazione o durante l'esecuzione.

Il processo è l'unità di lavoro nella maggior parte dei sistemi. Un sistema tipico è formato da processi del sistema operativo, che eseguono il codice di sistema, e da processi utenti, che eseguono il codice utente. Tutti questi processi sono eseguibili concorrentemente.

Benché tradizionalmente un processo contenga un solo thread di controllo dell'esecuzione, la maggior parte dei sistemi operativi moderni attualmente gestisce processi con più thread.

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei processi e dei thread di sistema e utenti: creazione e cancellazione, scheduling, offerta dei meccanismi di sincronizzazione e comunicazione, gestione delle situazioni di stallo.

## Capitolo 3

# Processi



### OBIETTIVI

- Introduzione del concetto di processo – un programma in esecuzione – che forma la base della computazione.
- Spiegazione delle diverse caratteristiche legate ai processi, comprese quelle relative a scheduling, creazione e terminazione, e comunicazione tra processi.
- Descrizione della comunicazione nei sistemi client-server.

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

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

## 3.1 Concetto di processo

Una questione che sorge dall'analisi dei sistemi operativi è la definizione delle attività della CPU. Un **sistema a lotti** (*batch*) esegue lavori (*job*), mentre un sistema a partizione del tempo esegue **programmi utenti** o **task**. Persino in un sistema monoutente, come Microsoft Windows, un utente può far eseguire diversi programmi contemporaneamente: un elaboratore di testi, un programma di consultazione del Web, un programma per la posta elettronica,

ca. Anche se l'utente esegue un solo programma alla volta, il sistema operativo deve svolgere le proprie attività interne, per esempio la gestione della memoria. Queste attività sono simili per molti aspetti, perciò sono denominate *processi*.

In questo testo i termini *lavoro* e *processo* sono usati in modo quasi intercambiabile. Sebbene si preferisca il termine *processo*, occorre ricordare che la maggior parte della terminologia e della teoria dei sistemi operativi si è sviluppata in un periodo in cui l'attività principale dei sistemi operativi riguardava la gestione dei lavori d'elaborazione in sistemi a lotti. Sarebbe fuorviante evitare di usare termini comunemente accettati che contengono la parola *lavoro* o *job*, per esempio *job scheduling*, solo perché il termine *processo* ha ormai soppiantato il termine *lavoro*.

### 3.1.1 Processo

Informalmente, un *processo* è un programma in esecuzione. È qualcosa di più del codice di un programma, talvolta noto anche come **sezione di testo**: comprende l'attività corrente, rappresentata dal valore del **contatore di programma** e dal contenuto dei registri della CPU; normalmente comprende anche la propria **pila (stack)**, contenente a sua volta i dati temporanei, come i parametri di un metodo, gli indirizzi di rientro e le variabili locali, e una **sezione di dati** contenente le variabili globali. Un processo può includere uno **heap**, ossia della memoria dinamicamente allocata durante l'esecuzione del processo. La struttura di un processo in memoria è illustrata nella Figura 3.1.

Sottolineiamo che un programma di per sé non è un processo; un programma è un'entità *passiva*, come il contenuto di un file memorizzato in un disco, mentre un processo è un'entità *attiva*, con un contatore di programma che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo allorquando il file eseguibile che lo contiene è caricato in memoria. Due tecniche comuni per ottenere questo effetto sono il doppio clic sull'icona del file seguibile e la digitazione del nome del file eseguibile nella riga di comando. (Esempi tipici di nomi di file eseguibili: `prog.exe` oppure `a.out`.)

Sebbene due processi siano associabili allo stesso programma, sono tuttavia da considerare due sequenze d'esecuzione distinte. Alcuni utenti possono, per esempio, far eseguire diverse istanze dello stesso programma di posta elettronica, così come un utente può invocare più istanze dello stesso programma di consultazione del Web. Ciascuna di loro è un diverso processo, e benché le sezioni di testo siano equivalenti, quelle dei dati, dello heap e

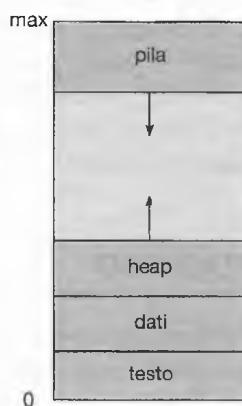


Figura 3.1 Processo in memoria.

della pila sono diverse. È inoltre usuale che durante la propria esecuzione un processo generi altri processi. Questo argomento è trattato nel Paragrafo 3.4.

### 3.1.2 Stato del processo

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

- ◆ **Nuovo.** Si crea il processo.
- ◆ **Esecuzione.** Un'unità d'elaborazione esegue le istruzioni del relativo programma.
- ◆ **Attesa.** Il processo attende che si verifichi qualche evento (come il completamento di un'operazione di I/O o la ricezione di un segnale).
- ◆ **Pronto.** Il processo attende di essere assegnato a un'unità d'elaborazione.
- ◆ **Terminato.** Il processo ha terminato l'esecuzione.

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

### 3.1.3 Blocco di controllo dei processi

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

- ◆ **Stato del processo.** Lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.
- ◆ **Contatore di programma.** Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.



**Figura 3.2** Diagramma di transizione degli stati di un processo.

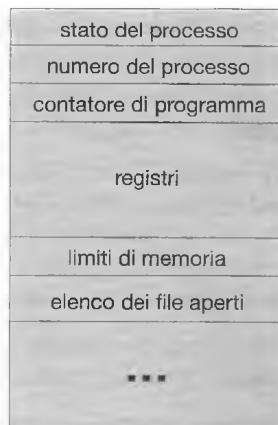


Figura 3.3 Blocco di controllo di un processo (PCB).

- **Registri di CPU.** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri d'indice, puntatori alla cima delle strutture a pila (*stack pointer*), registri d'uso generale e registri contenenti informazioni relative ai codici di condizione. Quando si verifica un'interruzione della CPU, si devono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo (Figura 3.4).
- **Informazioni sullo scheduling di CPU.** Queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling (nel Capitolo 5 si descrive lo scheduling dei processi).

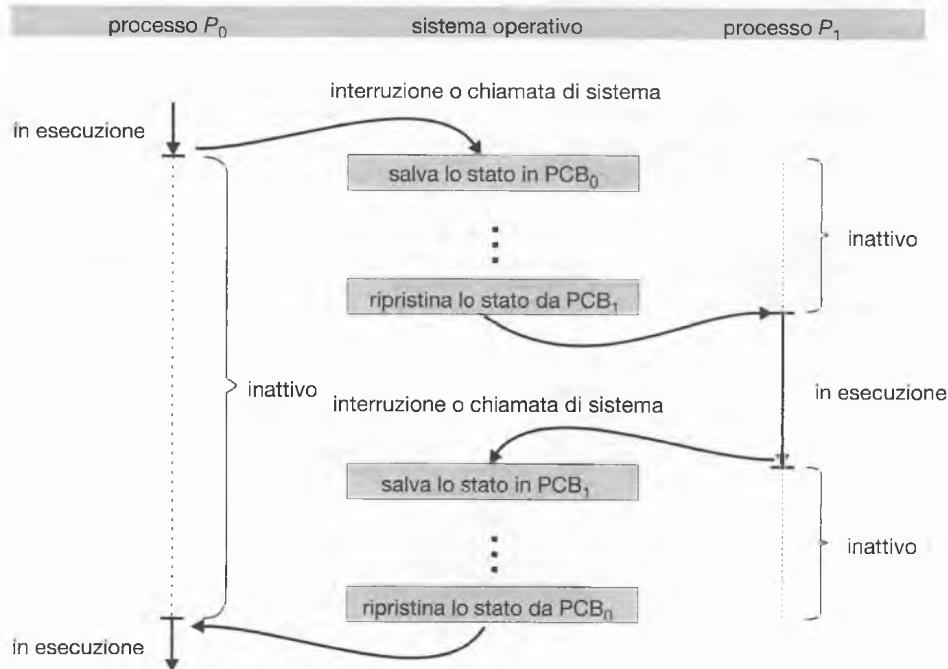


Figura 3.4 La CPU può essere commutata tra i processi.

- ◆ **Informazioni sulla gestione della memoria.** Queste informazioni si possono esprimere attraverso i valori dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo (Capitolo 8).
- ◆ **Informazioni di contabilizzazione delle risorse.** Queste informazioni comprendono il tempo d'uso della CPU e il tempo reale d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.
- ◆ **Informazioni sullo stato dell'I/O.** Queste informazioni comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti, e così via.

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

### 3.1.4 Thread

Il modello dei processi illustrato fin qui sottintende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione, detto **thread**. Se un processo sta, per esempio, eseguendo un programma di elaborazione di testi, l'esecuzione avviene seguendo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta. Tramite lo stesso processo, per esempio, un utente non può contemporaneamente inserire caratteri e verificare la correttezza ortografica di quel che sta scrivendo. In molti sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più percorsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta. Il Capitolo 4 è dedicato ai processi multithread.

## 3.2 Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della CPU. L'obiettivo della partizione del tempo è di commutare l'uso della CPU tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo **scheduler** dei processi seleziona un processo da eseguire dall'insieme di quelli disponibili. Nei sistemi monoprocesso non vi sarà mai più di un processo in esecuzione: gli altri dovranno attendere finché la CPU sia nuovamente disponibile.

### 3.2.1 Code di scheduling

Ogni processo è inserito in una **coda di processi**, composta da tutti i processi del sistema. I processi presenti in memoria centrale, che sono pronti e nell'attesa d'essere eseguiti, si trovano in una lista detta **coda dei processi pronti** (*ready queue*). Questa coda generalmente si memorizza come una lista concatenata: un'intestazione della coda dei processi pronti contiene i puntatori al primo e all'ultimo PCB dell'elenco, e ciascun PCB è esteso con un campo puntatore che indica il successivo processo contenuto nella coda dei processi pronti.

Il sistema operativo ha anche altre code. Quando si assegna la CPU a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta di I/O. Una richiesta di I/O può essere diretta a un dispositivo condiviso, come un disco. Poiché nel sistema esistono molti processi, il disco può essere occupato con una richiesta di I/O di qualche altro processo, quindi il processo deve attendere che il disco sia disponibile.

## RAPPRESENTAZIONE DEI PROCESSI DI LINUX

Il blocco di controllo dei processi nel sistema operativo Linux è rappresentato dalla struttura C `task_struct`, contenente una descrizione completa del processo, compreso il suo stato, informazioni sullo scheduling e sulla gestione della memoria, la lista dei file aperti, e puntatori al processo padre e agli eventuali figli. (Il *padre* o *genitore* di un processo è il processo che lo ha creato; i *figli* sono i processi generati.) Alcuni dei campi sono i seguenti:

```
pid_t pid; /* identificatore del processo */
long state; /* stato del processo */
unsigned int time_slice /* informazioni per lo scheduling */
struct files_struct *files; /* lista dei file aperti */
struct mm_struct *mm; /* spazio degli indirizzi del processo */
```

Per esempio, lo stato del processo è rappresentato dal campo `long state`. In Linux l'insieme dei processi è rappresentato da una lista doppia (in cui, cioè, ogni elemento punta al predecesore e al successore) di `task_struct`, e il kernel mantiene un puntatore di nome `current` al processo attualmente in esecuzione. Si veda la Figura 3.5.

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

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

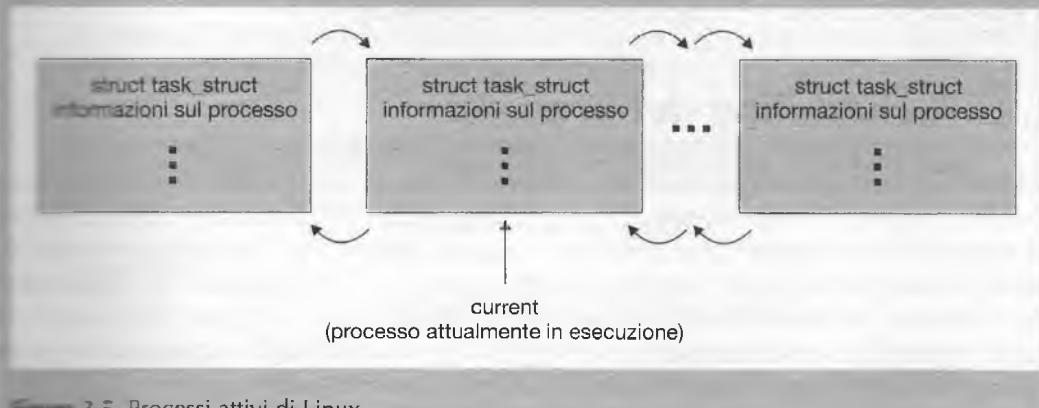


Figura 3.5 Processi attivi di Linux.

**Coda** dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama **coda del dispositivo**; ogni dispositivo ha la propria coda (Figura 3.6).

Una comune rappresentazione utile alla descrizione dello scheduling dei processi è dato da un **diagramma di accodamento** come quello illustrato nella Figura 3.7. Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la coda dei processi pronti e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

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

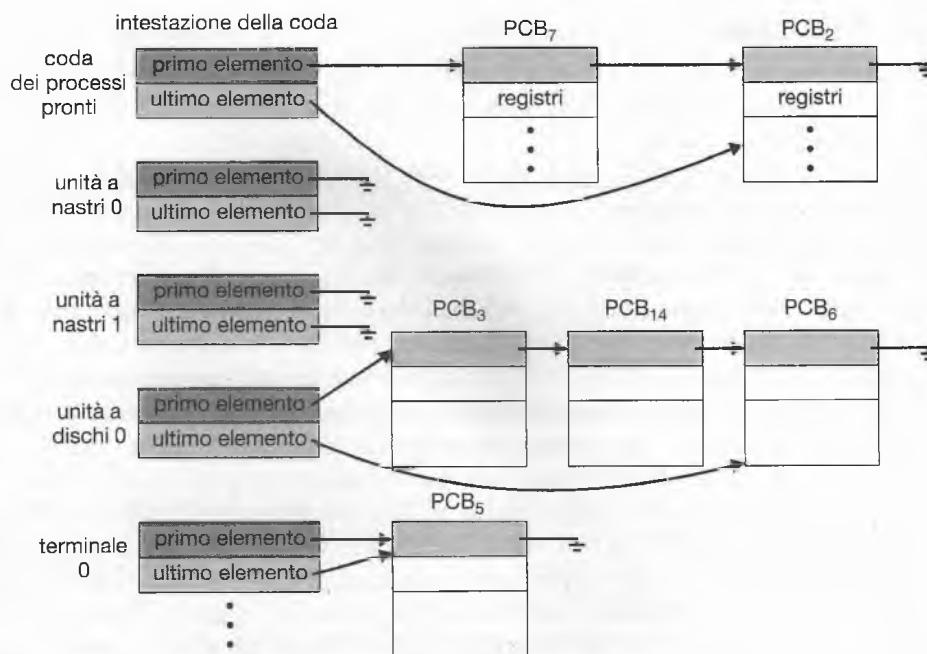


Figura 3.6 Coda dei processi pronti e diverse code di dispositivi I/O.

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

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio, il processo passa dallo stato d'attesa allo stato pronto ed è nuovamente inserito nella coda dei processi pronti. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene allontanato da tutte le code, rimosso il suo PCB e revocate le varie risorse.

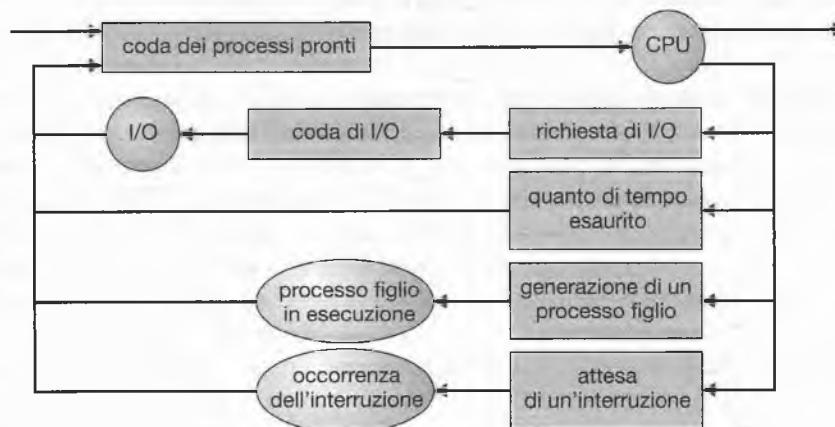


Figura 3.7 Diagramma di accodamento per lo scheduling dei processi.

### 3.2.2 Scheduler

Nel corso della sua esistenza, un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler**.

Spesso, in un sistema a lotti, accade che si sottopongano più processi di quanti se ne possano eseguire immediatamente. Questi lavori si trasferiscono in dispositivi di memoria secondaria, generalmente dischi, dove si tengono fino al momento dell'esecuzione (*spooling*). Lo **scheduler a lungo termine** (*job scheduler*), sceglie i lavori da questo insieme e li carica in memoria affinché siano eseguiti. Lo **scheduler a breve termine**, o **scheduler di CPU**, fa la selezione tra i lavori pronti per l'esecuzione e assegna la CPU a uno di loro.

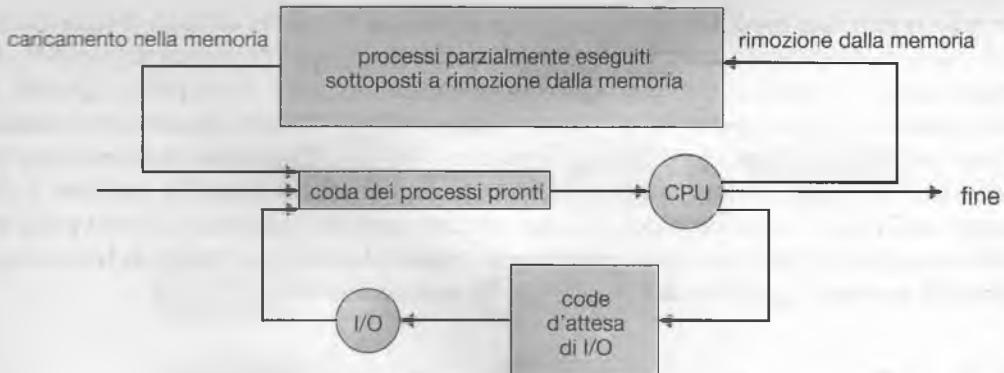
Questi due scheduler si differenziano principalmente per la frequenza con la quale sono eseguiti. Lo scheduler a breve termine seleziona frequentemente un nuovo processo per la CPU. Il processo può essere in esecuzione solo per pochi millisecondi prima di passare ad attendere una richiesta di I/O. Poiché spesso si esegue almeno una volta ogni 100 millisecondi, lo scheduler a breve termine deve essere molto rapido: se impiegasse 10 millisecondi per decidere quale processo eseguire nei 100 millisecondi successivi, si userebbe, o per meglio dire si sprecherebbe, il  $10/(100 + 10) = 9$  per cento del tempo di CPU per il solo scheduling.

Lo scheduler a lungo termine, invece, si esegue con una frequenza molto inferiore; diversi minuti possono trascorrere tra la creazione di un nuovo processo e il successivo. Lo scheduler a lungo termine controlla il **grado di multiprogrammazione**, cioè il numero di processi presenti in memoria. Se è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità media con cui i processi abbandonano il sistema; quindi lo scheduler a lungo termine si può richiamare solo quando un processo abbandona il sistema. A causa del maggior intervallo che intercorre tra le esecuzioni, lo scheduler a lungo termine dispone di più tempo per scegliere un processo per l'esecuzione.

È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di I/O, o come avente una prevalenza d'elaborazione. Un **processo con prevalenza di I/O** (*I/O bound*) impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O. Un **processo con prevalenza d'elaborazione** (*CPU bound*), viceversa, richiede poche operazioni di I/O e impiega la maggior parte del proprio tempo nelle elaborazioni. È fondamentale che lo scheduler a lungo termine selezioni una buona **combinazione di processi** con prevalenza di I/O e con prevalenza d'elaborazione. Se tutti i processi fossero con prevalenza di I/O, la coda dei processi pronti sarebbe quasi sempre vuota e lo scheduler a breve termine resterebbe pressoché inattivo. Se tutti i processi fossero con prevalenza d'elaborazione, la coda d'attesa per l'I/O sarebbe quasi sempre vuota, i dispositivi non sarebbero utilizzati, e il sistema verrebbe nuovamente sbilanciato. Le prestazioni migliori sono date da una combinazione equilibrata di processi con prevalenza di I/O e processi con prevalenza d'elaborazione.

Esistono sistemi in cui lo scheduler a lungo termine può essere assente o minimo. Per esempio, i sistemi a partizione del tempo, come UNIX e Microsoft Windows, sono spesso privi di scheduler a lungo termine, e si limitano a caricare in memoria tutti i nuovi processi, gestiti dallo scheduler a breve termine. La stabilità di questi sistemi dipende dai limiti fisici degli stessi, come un numero limitato di terminali disponibili, oppure dall'autoregolamentazione insita nella natura degli utenti umani: quando ci si accorge che il rendimento della macchina scende a livelli inaccettabili si possono semplicemente chiudere alcune attività o la sessione di lavoro.

In alcuni sistemi operativi come quelli a partizione del tempo, si può introdurre un livello di scheduling intermedio. Questo **scheduler a medio termine** è rappresentato schema-



**Figura 3.8** Aggiunta di scheduling a medio termine al diagramma di accodamento.

ticamente nella Figura 3.8. L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria (e dalla contesa attiva per la CPU), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta.

Questo schema si chiama **avvicendamento dei processi in memoria** – o, più in breve, **avvicendamento (swapping)**. Il processo viene rimosso e successivamente caricato in memoria dallo scheduler a medio termine. L'avvicendamento dei processi in memoria può servire a migliorare la combinazione di processi, oppure a liberare una parte della memoria se un cambiamento dei requisiti di memoria ha impegnato eccessivamente la memoria disponibile. L'avvicendamento dei processi in memoria è illustrato nel Capitolo 8.

### 3.2.3 Cambio di contesto

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

Il passaggio della CPU a un nuovo processo implica la registrazione dello stato del processo vecchio e il caricamento dello stato precedentemente registrato del nuovo processo. Questa procedura è nota col nome di **cambio di contesto** (*context switch*). Nell'evenienza di un cambio di contesto, il sistema salva nel suo PCB il contesto del processo uscente, e carica il contesto del processo subentrante, salvato in precedenza. Il cambio di contesto comporta un calo delle prestazioni, perché il sistema esegue solo operazioni volte alla corretta gestione dei processi, e non alla computazione. Il tempo necessario varia da sistema a sistema, dipendendo dalla velocità della memoria, dal numero di registri da copiare, e dall'esistenza di istruzioni macchina appropriate (per esempio, una singola istruzione per caricare o trasferire in memoria tutti i registri). In genere si tratta di qualche millisecondo.

La durata del cambio di contesto dipende molto dall'architettura; per esempio, alcune CPU (come la Sun UltrasPARC) offrono più gruppi di registri, quindi il cambio di contesto

prevede la semplice modifica di un puntatore al gruppo di registri corrente. Naturalmente, se il numero dei processi attivi è maggiore di quello dei gruppi di registri disponibili, il sistema rimedia copiando i dati dei registri nella e dalla memoria, come prima. Quindi, più complesso è il sistema operativo, più lavoro si deve svolgere durante un cambio di contesto. Come vedremo nel Capitolo 8, l'uso di tecniche complesse di gestione della memoria può richiedere lo spostamento di ulteriori dati a ogni cambio di contesto. Per esempio, si deve preservare lo spazio d'indirizzi del processo corrente mentre si prepara lo spazio per il processo successivo. Il modo in cui si preserva tale spazio e la relativa quantità di lavoro dipendono dal metodo di gestione della memoria del sistema operativo.

## 3.3 Operazioni sui processi

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

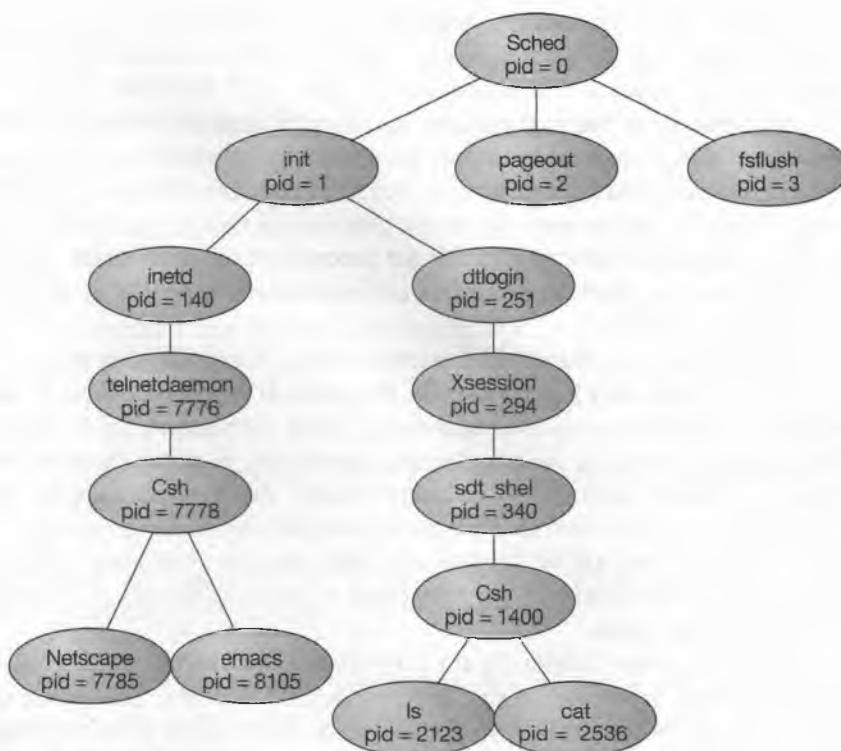
### 3.3.1 Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita chiamata di sistema (`create_process`). Il processo creante si chiama **processo genitore**, mentre il nuovo processo si chiama **processo figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero di processi**.

La maggior parte dei sistemi operativi, compresi UNIX e la famiglia Windows, identifica un processo per mezzo di un numero univoco, detto **identificatore del processo** o **pid** (*process identifier*). Si tratta solitamente di un intero. La Figura 3.9 mostra un albero dei processi del sistema operativo Solaris, con il nome e il pid di ogni processo. In questo sistema, il processo alla radice dell'albero è `sched`, il cui pid è 0. `sched` ha svariati figli, fra cui `pageout` e `fsflush`. Questi processi sono responsabili della gestione della memoria e dei file system. Il processo `sched` genera anche il processo `init`, che funge da genitore di tutti i processi utenti. La figura mostra due figli di `init`, `inetd` e `dtlogin`. Il primo gestisce i servizi di rete come `telnet` e `ftp`; il secondo rappresenta lo schermata di accesso al sistema. Quando un utente acceda al sistema, `dtlogin` crea una sessione X-windows (`xsession`), che a sua volta genera il processo `sdt_shel`, al di sotto del quale risiede la shell dell'utente – per esempio, `csh`, ossia C-shell. È da tale interfaccia che l'utente invoca vari processi figli, come per esempio i comandi `ls` e `cat`. Dalla figura si può anche notare un processo `csh` con pid pari a 7788, che rappresenta un utente che abbia avuto accesso al sistema tramite `telnet`. Questo utente ha avviato il browser Netscape (pid 7785) e l'editor `emacs` (pid 8105).

Nei sistemi UNIX si può ottenere l'elenco dei processi tramite il comando `ps`. Digitando `ps -el` si otterranno informazioni complete su tutti i processi attualmente attivi nel sistema; è facile costruire un albero come quello nella figura identificando ricorsivamente i processi genitore fino a giungere a `init`.

In generale, per eseguire il proprio compito, un processo necessita di alcune risorse (tempo d'elaborazione, memoria, file, dispositivi di I/O). Quando un processo crea un sottoprocesso, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo



**Figura 3.9** Esempio di albero dei processi di Solaris.

genitore. Il processo genitore può avere la necessità di spartire le proprie risorse tra i suoi processi figli, oppure può essere in grado di condividerne alcune, come la memoria o i file, tra più processi figli. Limitando le risorse di un processo figlio a un sottoinsieme di risorse del processo genitore, si può evitare che un processo sovraccarichi il sistema creando troppi sottoprocessi.

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

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

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

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

1. il processo figlio è un duplicato del processo genitore;
2. nel processo figlio si carica un nuovo programma.

Nel sistema operativo UNIX, per esempio, ogni processo è identificato dal proprio identificatore di processo, un intero unico. Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare senza difficoltà con il proprio processo figlio. Entrambi i processi (genitore e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema `fork()`, con una differenza: la chiamata di sistema `fork()` riporta il valore zero nel nuovo processo (il figlio), ma riporta l'identificatore del processo figlio (il PID diverso da zero) nel processo genitore. Tramite il valore riportato del PID, i due processi possono procedere nell'esecuzione "sapendo" qual è il processo padre e qual è il processo figlio.

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

Il programma scritto in C della Figura 3.10 illustra le chiamate di sistema UNIX descritte precedentemente. Si ottengono due processi distinti ciascuno dei quali è un'istanza d'esecuzione dello stesso programma. Il valore assegnato alla variabile `pid` è zero nel processo figlio, e un numero intero maggiore di zero nel processo genitore. Usando la chiamata di sistema `execvp()` – una versione della chiamata di sistema `exec()` – il processo figlio sovrappone il proprio spazio d'indirizzi con il comando `/bin/ls` di UNIX (che si usa per ottenere l'elenco del contenuto di una directory). Impiegando la chiamata di sistema `wait()`, il processo genitore attende che il processo figlio termini. Quando ciò accade, il processo genitore chiude la propria fase d'attesa dovuta alla chiamata di sistema `wait()` e termina usando la chiamata di sistema `exit()`. Questo passaggio è illustrato nella Figura 3.11.

Come altro esempio, consideriamo la creazione dei processi in Windows. Nella API Win32 la funzione `CreateProcess()` è simile alla `fork()` di UNIX, poiché serve a generare un figlio da un processo genitore. Mentre però `fork()` passa in eredità al figlio lo spazio degli indirizzi del genitore, `CreateProcess()` richiede il caricamento di un programma specificato nello spazio degli indirizzi del processo figlio al momento della sua creazione. Inoltre, mentre `fork()` non richiede parametri, `CreateProcess()` se ne aspetta non meno di dieci. Il programma nella Figura 3.12, scritto in C, illustra la funzione `CreateProcess()`; essa genera un processo figlio che carica l'applicazione `mspaint.exe`. In questo esempio, i dieci parametri passati a `CreateProcess()` hanno in molti casi il loro valore di default.

I lettori interessati alla creazione e gestione dei processi nella API Win32 possono consultare i riferimenti bibliografici alla fine del capitolo.

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

```

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

int main()
{
    pid_t pid;

    /* genera un nuovo processo */
    pid = fork();

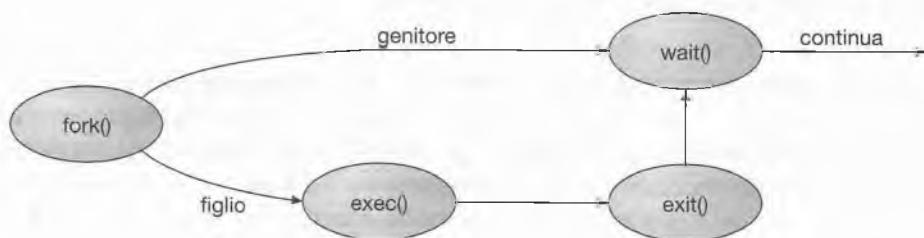
    if (pid < 0) { /* errore */
        fprintf(stderr, "generazione del nuovo processo fallita");
        return 1;
    }
    else if (pid == 0) { /* processo figlio */
        execlp("/bin/ls", "ls", NULL);
    }
    else {/* processo genitore */
        /* il genitore attende il completamento del figlio */
        wait(NULL);
        printf("il processo figlio ha terminato");
    }

    return 0;
}

```

**Figura 3.10** Creazione di un processo separato utilizzando la chiamata di sistema `fork()`.

I primi due parametri passati a `CreateProcess()` sono il nome dell'applicazione e i parametri della riga di comando. Se il nome dell'applicazione è `waitForSingleObject()` (come nell'esempio della figura), è il parametro della riga di comando a specificare quale applicazione caricare. Nell'esempio, si carica l'applicazione `mspaint.exe` di Microsoft Windows. Al di là dei primi due parametri, l'esempio usa i parametri di default per far ereditare gli handle del processo e del thread, e per specificare l'assenza di flag di creazione. Il figlio adotta inoltre il blocco ambiente del genitore e la sua directory iniziale. Infine, si passano i due



**Figura 3.11** Generazione di un processo per mezzo della chiamata di sistema `fork()`.

```

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

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

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

// genera processo figlio
if (!CreateProcess(NULL, // usa riga di comando
"C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", // riga di comando
NULL, // non eredita l'handle del processo
NULL, // non eredita l'handle del thread
FALSE, // disattiva l'ereditarieta' degli handle
0, // nessun flag di creazione
NULL, // usa il blocco ambiente del genitore
NULL, // usa la directory esistente del genitore
&si,
&pi))
{
    fprintf(stderr, "generazione del nuovo processo fallita");
    return -1
}
// il genitore attende il completamento del figlio
WaitForSingleObject(pi.hProcess, INFINITE);
printf("il processo figlio ha terminato");

// rilascia gli handle
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}

```

**Figura 3.12** Creazione di un nuovo processo con la API Win32.

puntatori alle strutture **STARTUPINFO** e **PROCESS\_INFORMATION** create all'inizio. Nella precedente Figura 3.10, il processo genitore attende la terminazione del figlio invocando la chiamata di sistema **wait()**. L'equivalente per Win32 è **WaitForSingleObject()**, a cui si passa l'handle del processo figlio – **pi.hProcess** – del cui completamento si è in attesa. A terminazione del figlio avvenuta, il controllo ritorna al processo genitore, al punto immediatamente successivo alla chiamata **WaitForSingleObject()**.

### 3.3.2 Terminazione di un processo

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

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un'opportuna chiamata di sistema (per esempio `TerminateProcess()` in Win32). Generalmente solo il genitore del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un genitore deve conoscere le identità dei propri figli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

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

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

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

Per illustrare un esempio di esecuzione e terminazione di un processo, si consideri che nel sistema operativo UNIX un processo può terminare per mezzo della chiamata di sistema `exit()`, e il suo processo genitore può attendere l'evento per mezzo della chiamata di sistema `wait()`. Quest'ultima riporta l'identificatore di un processo figlio che ha terminato l'esecuzione, sicché il genitore può stabilire quale tra i suoi processi figli l'abbia terminata. Se un processo genitore termina, tutti i suoi processi figli sono affidati al processo `init()`, che assume il ruolo di nuovo genitore, cui i processi figli possono riportare i risultati delle proprie attività.

## 3.4 Comunicazione tra processi

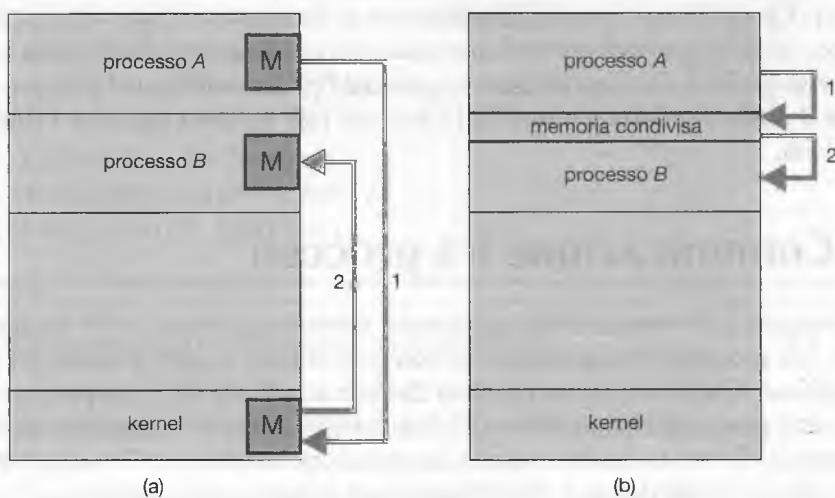
I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è **indipendente** se non può influire su altri processi del sistema o subirne l'influsso. Chiaramente, un processo che non condivide dati (temporanei o permanenti) con altri processi è indipendente.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

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

- ◆ **Condivisione d'informazioni.** Poiché più utenti possono essere interessati alle stesse informazioni (per esempio un file condiviso), è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse.
- ◆ **Accelerazione del calcolo.** Alcune attività d'elaborazione sono realizzabili più rapidamente se si suddividono in sottoattività eseguibili in parallelo. Un'accelerazione di questo tipo è ottenibile solo se il calcolatore dispone di più elementi capaci di attività d'elaborazione (come più CPU o canali di I/O).
- ◆ **Modularità.** Può essere necessaria la costruzione di un sistema modulare, che suddivide le funzioni di sistema in processi o thread distinti (si veda il Capitolo 2).
- ◆ **Convenienza.** Anche un solo utente può avere la necessità di compiere più attività contemporaneamente; per esempio, può eseguire in parallelo le operazioni di scrittura, stampa e compilazione.

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi** (*IPC, interprocess communication*). I modelli fondamentali della comunicazione tra processi sono due: (1) a **memoria condivisa** e (2) a **scambio di messaggi**. Nel modello a memoria condivisa, si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona. Nel secondo tipo di modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti. I due modelli sono messi a confronto nella Figura 3.13.

Nei sistemi operativi sono diffusi entrambi i modelli; a volte coesistono in un unico sistema. Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti. La memoria condivisa massimizza l'efficienza della comunicazione, ed è più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono gestiti alla stregua di ordinari accessi in memoria che non richiedono l'assistenza del kernel. Nel seguito di questo paragrafo esploriamo i due modelli IPC in maggiore dettaglio.



**Figura 3.13** Modelli di comunicazione tra processi basati su (a) scambio di messaggi, e (b) condivisione della memoria.

### 3.4.1 Sistemi a memoria condivisa

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

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

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

Sono utilizzabili due tipi di buffer. Quello **illimitato** non pone limiti alla dimensione del buffer. Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrne. Il problema del produttore e del consumatore con **buffer limitato** presuppone l'esistenza di una dimensione fissa del buffer in questione. In questo caso, il consumatore deve attendere che il buffer sia vuoto; viceversa, il produttore deve attendere che il buffer sia pieno.

Consideriamo più attentamente come il buffer limitato sia utilizzabile per la condivisione di memoria tra processi. Le variabili seguenti risiedono in una zona di memoria condivisa sia dal produttore sia dal consumatore.

```
#define DIM_BUFFER 10

typedef struct {

    . . .

} elemento;

elemento buffer[DIM_BUFFER];
int inserisci = 0;
int preleva = 0;
```

Il buffer condiviso è realizzato come un array circolare con due puntatori logici: **inserisci** e **preleva**. La variabile **inserisci** indica la successiva posizione libera nel buffer; **preleva** indica la prima posizione piena nel buffer. Il buffer è vuoto se **inserisci == preleva**; è pieno se  $((inserisci + 1) \% \text{DIM\_BUFFER}) == \text{preleva}$ .

```

elemento appena_Prodotto;

while (true) {
    /* produce un elemento in appena_Prodotto */
    while (((inserisci + 1) % DIM_BUFFER) == preleva)
        ; /* non fa niente */
    buffer[inserisci] = appena_Prodotto;
    inserisci = (inserisci + 1) % DIM_BUFFER;
}

```

**Figura 3.14** Processo produttore.

```

elemento da_Consumare;

while (true) {
    while (inserisci == preleva)
        ; /* non fa niente */

    da_Consumare = buffer[preleva];
    preleva = (preleva + 1) % DIM_BUFFER;
    /* consuma l'elemento in da_Consumare */
}

```

**Figura 3.15** Processo consumatore.

Di seguito si riporta il codice per i processi produttore e consumatore (illustrato, rispettivamente, nella Figura 3.14 e 3.15). Il processo produttore ha una variabile locale `appena_Prodotto` contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile locale `da_Consumare` in cui si memorizza l'elemento da consumare.

Questo metodo ammette un massimo di `DIM_BUFFER`-1 oggetti contemporaneamente presenti nel buffer. Proponiamo come esercizio per il lettore la stesura di un algoritmo che permetta la presenza contemporanea di `DIM_BUFFER` oggetti. Nel Paragrafo 3.5.1 si illustrerà la API POSIX per la memoria condivisa.

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

### 3.4.2 Sistemi a scambio di messaggi

Nel Paragrafo 3.4.1 si è parlato di come può avvenire la comunicazione tra processi cooperanti in un ambiente a memoria condivisa. Per essere applicato, lo schema proposto richiede che tali processi condividano l'accesso a una zona di memoria e che il codice per la realizzazione e la gestione della memoria condivisa sia scritto esplicitamente dal programmatore che crea l'applicazione. Un altro modo in cui il sistema operativo può ottenere i medesimi risultati consiste nel fornire ai processi appositi strumenti per lo scambio di messaggi.

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

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

Se i processi *P* e *Q* vogliono comunicare, devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un **canale di comunicazione** (*communication link*), realizzabile in molti modi. In questo paragrafo non si tratta della realizzazione fisica del canale (come la memoria condivisa, i bus o le reti, illustrati nel Capitolo 16) ma della sua realizzazione logica. Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni **send()** e **receive()**:

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

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

### 3.4.2.1 Nominazione

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

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

- ◆ **send(P, messaggio)**, invia messaggio al processo *P*;
- ◆ **receive(Q, messaggio)**, riceve, in messaggio, un messaggio dal processo *Q*.

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

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

Esiste esattamente un canale tra ciascuna coppia di processi.

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

- ◆ **send(P, messaggio)**, invia messaggio al processo *P*;
- ◆ **receive(id, messaggio)**, riceve, in messaggio, un messaggio da qualsiasi processo; nella variabile *id* si riporta il nome del processo con cui è avvenuta la comunicazione.

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

Con la comunicazione indiretta, i messaggi s'inviano a delle porte ( dette anche *mail-box*), che li ricevono. Una porta si può considerare in modo astratto come un oggetto in cui i processi possono introdurre e prelevare messaggi, ed è identificata in modo unico. Per l'identificazione di una porta le code di messaggi POSIX usano per esempio un valore intero. In questo schema un processo può comunicare con altri processi tramite un certo numero di porte. Due processi possono comunicare solo se condividono una porta. Le primitive `send()` e `receive()` si definiscono come segue:

- ◆ `send(A, messaggio)`, invia messaggio alla porta A;
- ◆ `receive(A, messaggio)`, riceve, in messaggio, un messaggio dalla porta A.

In questo schema un canale di comunicazione ha le seguenti caratteristiche:

- ◆ tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa porta;
- ◆ un canale può essere associato a più di due processi;
- ◆ tra ogni coppia di processi comunicanti possono esserci più canali diversi, ciascuno corrispondente a una porta.

A questo punto, si supponga che i processi  $P_1$ ,  $P_2$  e  $P_3$  condividano la porta  $A$ . Il processo  $P_1$  invia un messaggio ad  $A$ , mentre sia  $P_2$  sia  $P_3$  eseguono una `receive()` da  $A$ . Sorge il problema di sapere quale processo riceverà il messaggio. La soluzione dipende dallo schema prescelto:

- ◆ si può fare in modo che un canale sia associato al massimo a due processi;
- ◆ si può consentire l'esecuzione di un'operazione `receive()` a un solo processo alla volta;
- ◆ si può consentire al sistema di decidere arbitrariamente quale processo riceverà il messaggio (il messaggio sarà ricevuto da  $P_2$  o da  $P_3$ , ma non da entrambi). Il sistema può anche definire un algoritmo per selezionare quale processo riceverà il messaggio (specificando cioè uno schema, detto *round robin*, secondo il quale i processi ricevono i messaggi a turno) e può comunicare l'identità del ricevente al trasmittente.

Una porta può appartenere al processo o al sistema. Se appartiene a un processo, cioè fa parte del suo spazio d'indirizzi, occorre distinguere ulteriormente tra il proprietario, che può soltanto ricevere messaggi tramite la porta, e l'utente, che può solo inviare messaggi alla porta. Poiché ogni porta ha un unico proprietario, non può sorgere confusione su chi debba ricevere un messaggio inviato a una determinata porta. Quando un processo che possiede una porta termina, questa scompare, e qualsiasi processo che invii un messaggio alla porta di un processo già terminato ne deve essere informato.

D'altra parte, una porta posseduta dal sistema operativo è indipendente e non è legata ad alcun processo particolare. Il sistema operativo offre un meccanismo che permette a un processo le seguenti operazioni:

- ◆ creare una nuova porta;
- ◆ inviare e ricevere messaggi tramite la porta;
- ◆ rimuovere una porta.

Il processo che crea una nuova porta è il proprietario predefinito della porta, inizialmente è l'unico processo che può ricevere messaggi attraverso questa porta. Tuttavia, il diritto di proprietà e il diritto di ricezione si possono passare ad altri processi per mezzo di idonee chiamate di sistema. Naturalmente questa disposizione potrebbe dar luogo alla creazione di più riceventi per ciascuna porta.

### 3.4.2.2 Sincronizzazione

La comunicazione tra processi avviene attraverso chiamate delle primitive `send()` e `receive()`. Ci sono diverse possibilità nella definizione di ciascuna primitiva. Lo scambio di messaggi può essere **sincrono** (o **bloccante**) oppure **asincrono** (o **non bloccante**).

- ◆ **Invio sincrono.** Il processo che invia il messaggio si blocca nell'attesa che il processo ricevente, o la porta, riceva il messaggio.
- ◆ **Invio asincrono.** Il processo invia il messaggio e riprende la propria esecuzione.
- ◆ **Ricezione sincrona.** Il ricevente si blocca nell'attesa dell'arrivo di un messaggio.
- ◆ **Ricezione asincrona.** Il ricevente riceve un messaggio valido oppure un valore nullo.

È possibile anche avere diverse combinazioni di `send()` e `receive()` tra quelle illustrate sopra. Se le primitive `send()` e `receive()` sono entrambe bloccanti si parla di **rendezvous** tra mittente e ricevente. È banale dare soluzione al problema del produttore e del consumatore tramite le primitive bloccanti `send()` e `receive()`. Il produttore, infatti, si limita a invocare `send()` e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore richiama `receive()`, bloccandosi fino all'arrivo di un messaggio.

Come si avrà modo di notare durante la lettura del testo, i concetti di sincronia e asincronia compaiono spesso negli algoritmi per l'I/O dei sistemi operativi.

### 3.4.2.3 Code di messaggi

Se la comunicazione è diretta o indiretta, i messaggi scambiati tra processi comunicanti risiedono in code temporanee. Fondamentalmente esistono tre modi per realizzare queste code.

- ◆ **Capacità zero.** La coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio.
- ◆ **Capacità limitata.** La coda ha lunghezza finita  $n$ , quindi al suo interno possono risiedere al massimo  $n$  messaggi. Se la coda non è piena, quando s'invia un nuovo messaggio, quest'ultimo è posto in fondo alla coda; il messaggio viene copiato oppure si tiene un puntatore a quel messaggio. Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale ha tuttavia una capacità limitata; se è pieno, il trasmittente deve fermarsi nell'attesa che ci sia spazio disponibile nella coda.
- ◆ **Capacità illimitata.** La coda ha una lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

Il caso con capacità zero è talvolta chiamato sistema a scambio di messaggi senza memorizzazione transitoria (*no buffering*); gli altri due, sistemi con memorizzazione transitoria automatica (*automatic buffering*).

## 3.5 Esempi di sistemi per la IPC

In questo paragrafo analizzeremo tre sistemi per la comunicazione fra processi: la API POSIX basata sulla memoria condivisa, lo scambio di messaggi nel sistema operativo Mach, e la interessante mistura di queste due soluzioni adottata da Windows XP.

### 3.5.1 Un esempio: memoria condivisa secondo POSIX

Lo standard POSIX prevede svariati meccanismi per la IPC, compresa la condivisione della memoria e lo scambio di messaggi. Qui illustreremo la API POSIX per la condivisione della memoria.

Per prima cosa, il processo deve allocare un segmento condiviso di memoria tramite la chiamata di sistema `shmget()`; l'acronimo deriva da *shared memory get*, ossia “acquisisci memoria condivisa”. Per esempio, nella riga

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

il primo parametro specifica la chiave o l'identificatore del segmento condiviso. Se esso vale `IPC_PRIVATE` il sistema alloca un nuovo segmento di memoria condivisa. Il secondo parametro è la dimensione in byte del segmento. Il terzo parametro stabilisce il modo d'accesso al segmento – in lettura, in scrittura, o in entrambi i modi. Il valore `S_IRUSR | S_IWUSR` indica che il proprietario del segmento può accedervi sia in lettura sia in scrittura. Se la chiamata a `shmget()` va a buon fine, il valore restituito è un intero che identifica il segmento di memoria allocato: i processi che vorranno condividere questo segmento, dovranno richiederlo per mezzo del suo identificatore.

I processi che vogliono accedere a un segmento di memoria condivisa devono annetterlo al loro spazio degli indirizzi tramite una chiamata a `shmat()`, acronimo di *shared memory attach* – “annetti memoria condivisa”. Anch'essa richiede tre parametri, di cui il primo è l'identificatore intero del segmento da annettere, e il secondo è un puntatore che indica in che punto della memoria va annesso il segmento condiviso; se tale puntatore vale `NULL`, la scelta del punto sarà demandata al sistema operativo. Il terzo parametro è un selettor di modo d'accesso. Quando esso vale zero, il segmento è annesso con modalità sola lettura, mentre se il selettor è impostato (cioè, vale  $> 0$ ), l'accesso è consentito sia in lettura sia in scrittura. Se l'invocazione

```
shared_memory = (char *) shmat(id, NULL, 0);
```

ha successo, `shmath()` restituisce un puntatore alla prima locazione di memoria ove è stato annesso il segmento condiviso.

Usando questo puntatore, il processo può dunque accedere alla memoria annessa nello stesso modo in cui accederebbe ordinariamente alla memoria. Nell'esempio, `shmat()` restituisce un puntatore a una stringa di caratteri; si può quindi scrivere nell'area condivisa in questo modo:

```
sprintf(shared_memory, "Scrittura in memoria condivisa");
```

Gli altri processi che condividono il segmento rileveranno la modifica in memoria.

Di solito, la procedura seguita dai processi per condividere memoria già allocata è quella di annettere un segmento preesistente, e poi accedervi (eventualmente modificandolo). Quando un dato processo non ha più bisogno di condividere un segmento di memoria, lo elimina dal suo spazio degli indirizzi tramite

```
shmdt(shared_memory);
```

dove `shared_memory` è il puntatore al segmento di memoria da eliminare. Infine, un segmento condiviso può essere eliminato dall'intero sistema tramite `shmctl()`, cui si passa l'identificatore del segmento insieme al selettor `IPC_RMID`.

Il programma illustrato nella Figura 3.16 mostra la API POSIX che abbiamo preso in esame. Esso alloca un segmento condiviso di 4096 byte, e scrive il messaggio `Ciao` al suo interno. Dopo aver stampato il contenuto della memoria condivisa, il programma lo elimina prima dal proprio spazio degli indirizzi e poi dall'intero sistema. Altri esercizi sulla API POSIX per la condivisione della memoria si trovano alla fine del capitolo.

### 3.5.2 Un esempio: Mach

Come esempio di sistema operativo basato sullo scambio di messaggi, consideriamo il sistema operativo Mach, sviluppato alla Carnegie Mellon University. Abbiamo già presentato questo sistema operativo nel Capitolo 2, come parte di Mac OS X. Il suo kernel consente la creazione e la soppressione di più *task*, simili ai processi, ma che hanno più thread di controllo. La maggior parte delle comunicazioni – compresa una gran parte delle chiamate di sistema e tutte le informazioni tra task – si compie per mezzo di *messaggi*. I messaggi s'inviano e si ricevono attraverso *porte*.

Anche le chiamate di sistema s'invocano per mezzo di messaggi. Al momento della creazione di ogni task si creano due porte speciali: la porta *Kernel* e la porta *Notify*. Il kernel usa la porta *Kernel* per comunicare con il task e notifica l'occorrenza di un evento alla porta *Notify*. Per il trasferimento dei messaggi sono necessarie solo tre chiamate di sistema: `msg_send()`, che invia un messaggio a una porta; `msg_receive()`, per ricevere un messaggio; `msg_rpc()`, per le chiamate di procedure remote (*remote procedure call*, RPC), che invia un messaggio e ne attende esattamente uno di risposta per il trasmittente (in questo modo la RPC riproduce l'usuale chiamata di procedura, ma può operare tra sistemi diversi, da cui il termine *remota*).

La chiamata di sistema `port_allocate()` crea una nuova porta e assegna lo spazio per la sua coda di messaggi. La dimensione massima predefinita di tale coda è di otto messaggi. Il task che crea la porta è il proprietario della stessa, e può accedervi per la ricezione dei messaggi. Solo un task alla volta può possedere una porta o ricevere da una porta ma, all'occorrenza, questi diritti si possono trasmettere anche ad altri task.

La porta ha inizialmente una coda di messaggi vuota e i messaggi si copiano nella porta nell'ordine in cui sono ricevuti: tutti i messaggi hanno la stessa priorità. Mach garantisce che più messaggi in arrivo dallo stesso trasmittente siano accodati nell'ordine d'arrivo (*first-in, first-out*, FIFO), ma non garantisce un ordinamento assoluto. Per esempio, i messaggi inviati da due trasmittenti possono essere accodati in un ordine qualsiasi.

Gli stessi messaggi sono composti da un'intestazione di lunghezza fissa, seguita da una porzione di dati di lunghezza variabile. L'intestazione contiene la lunghezza del messaggio e due nomi di porte, uno dei quali è il nome della porta cui s'invia il messaggio. Normalmente il thread trasmittente attende una risposta; il nome della porta del trasmittente è passato al task ricevente, che lo impiega come un "indirizzo del mittente".

La parte variabile del messaggio è composta da una lista di dati tipizzati. Ogni elemento della lista ha un tipo, una dimensione e un valore. Il tipo degli oggetti specificati nel messaggio è importante, poiché oggetti definiti dal sistema operativo, come diritti di proprietà o di ricezione, stati del task e segmenti di memoria, si possono inviare all'interno dei messaggi.

Anche le operazioni di trasmissione e ricezione sono piuttosto flessibili. Per esempio, quando s'invia un messaggio a una porta che non è già piena, lo si copia al suo interno e il

```

#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    /* identificatore del segmento condiviso */
    int segment_id;
    /* puntatore al segmento condiviso */
    char* shared_memory;
    /* dimensione (in byte) del segmento condiviso */
    const int size = 4096;

    /* alloca il segmento condiviso */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* annette il segmento condiviso */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* scrive un messaggio nel segmento condiviso */
    sprintf(shared_memory, "Ciao");

    /* stampa la stringa contenuta nel segmento condiviso */
    printf("*%s\n", shared_memory);

    /* elimina il segmento condiviso dal proprio spazio indirizzi */
    shmdt(shared_memory);

    /* elimina il segmento condiviso dal sistema */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}

```

**Figura 3.16** Programma C che illustra la API POSIX per la condivisione della memoria.

thread trasmittente prosegue la sua esecuzione. Se la porta è piena, il thread trasmittente ha le seguenti possibilità.

1. Attendere indefinitamente che nella porta ci sia spazio.
2. Attendere al massimo  $n$  millisecondi.
3. Non attendere, ma rientrare immediatamente.
4. Memorizzare temporaneamente il messaggio. Un messaggio si può consegnare al sistema operativo anche se la porta che dovrebbe riceverlo è piena. Quando il messaggio può effettivamente essere messo nella porta, s'invia un avviso al trasmittente; per una porta piena può restare in sospeso un solo messaggio di questo tipo, in qualunque momento per un dato thread trasmittente.

L'ultima possibilità si usa per i task che svolgono servizi (*server task*), come i driver delle stampanti. Dopo aver portato a termine una richiesta, questi task possono aver bisogno d'inviare un'unica risposta al task che aveva richiesto il servizio, ma devono anche proseguire con altre richieste di servizi, anche se la porta di risposta per un client è piena.

Nell'operazione `receive()` occorre specificare da quale porta o insieme di porte debba provenire il messaggio. Un **insieme di porte** (*mailbox set*), dichiarato dal task, si tratta come se fosse un'unica porta. I thread di un task possono ricevere solo da un insieme di porte (o da una porta) per cui tale task ha il diritto di ricezione. Una chiamata di sistema `port_status()` restituisce il numero dei messaggi in una data porta. L'operazione di ricezione tenta di ricevere da (1) una qualsiasi tra le porte di un insieme; o (2) da una porta specifica (nominata). Se non è atteso alcun messaggio, il thread ricevente può attendere un massimo di  $n$  millisecondi o non attendere affatto.

Il sistema Mach è stato progettato per i sistemi distribuiti (descritti nei Capitoli dal 16 al 18), ma è adatto anche a sistemi con singola CPU. I problemi più gravi dei sistemi a scambio di messaggi sono generalmente dovuti alle scarse prestazioni dovute alla doppia operazione di copiatura dei messaggi dal trasmittente alla porta e quindi dalla porta al ricevente. Il sistema di messaggi di Mach cerca di evitare doppie operazioni di copiatura impiegando tecniche di gestione della memoria virtuale (Capitolo 9). Fondamentalmente Mach associa lo spazio d'indirizzi contenente il messaggio del trasmittente allo spazio d'indirizzi del ricevente. Il messaggio stesso non è mai effettivamente copiato, quindi le prestazioni del sistema migliorano notevolmente anche se solo nel caso di messaggi all'interno dello stesso sistema.

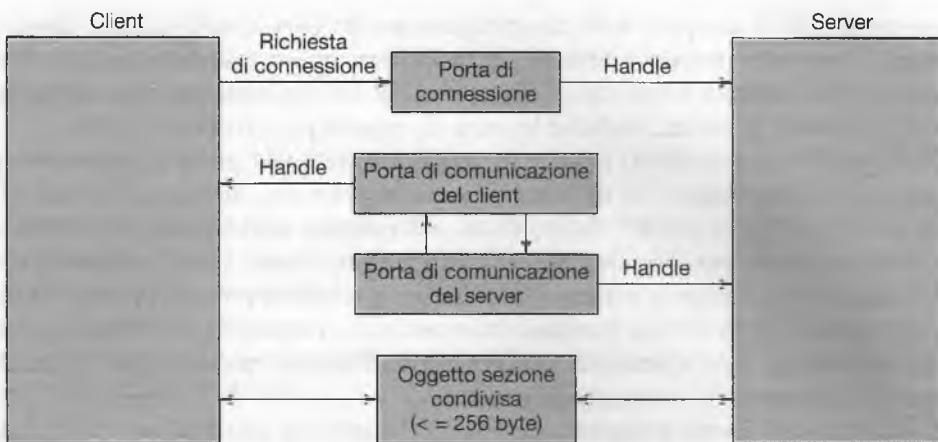
### 3.5.3 Un esempio: Windows XP

Il sistema operativo Windows XP è un esempio di moderno progetto che impiega la modularità per aumentare la funzionalità e diminuire il tempo necessario alla realizzazione di nuove caratteristiche. Windows XP gestisce più ambienti operativi o *sottosistemi*, con cui i programmi applicativi comunicano attraverso un meccanismo a scambio di messaggi; tali programmi si possono considerare *client* del *server* costituito dal sottosistema.

La funzione di scambio di messaggi di Windows XP è detta **chiamata di procedura locale** (*local procedure call*, LPC) e si usa per la comunicazione tra due processi presenti nello stesso calcolatore. È simile al meccanismo standard della chiamata di procedura remota, ma è ottimizzata per questo sistema. Come in Mach, nel sistema Windows XP s'impiega un oggetto porta per stabilire e mantenere una connessione tra due processi. Ogni client che invoca un sottosistema necessita di un canale di comunicazione che è fornito da un oggetto porta e che non è mai ereditato. Windows XP usa due tipi di porte: le porte di connessione e le porte di comunicazione. Sono essenzialmente identiche ma ricevono nomi diversi secondo il modo in cui si usano. Le porte di connessione sono *oggetti con nome* visibili a tutti i processi e forniscono alle applicazioni un modo per stabilire un canale di comunicazione (Capitolo 22). Tale comunicazione funziona come segue.

- ◆ Il client apre un handle per l'oggetto porta di connessione del sottosistema.
- ◆ Il client invia una richiesta di connessione.
- ◆ Il server crea due porte di comunicazione private e restituisce l'handle per una di loro al client.
- ◆ Il client e il server impiegano l'handle corrispondente di porta per inviare messaggi o *callback* ("chiamate di ritorno") e ascoltare le risposte.

Il sistema Windows XP si serve di tre tipi di tecniche di scambio di messaggi su una porta specificata dal client quando stabilisce il canale. La più semplice si usa per brevi messaggi (fi-



**Figura 3.17** Chiamate di procedura locale in Windows XP.

no a 256 byte) e consiste nell'impiegare la coda dei messaggi della porta come una memoria intermedia per copiare il messaggio da un processo all'altro.

Se il client deve inviare un messaggio più lungo, il trasferimento avviene tramite un **oggetto sezione** (che costituisce una regione di memoria condivisa); istituito il canale, il client stabilisce se deve inviare messaggi lunghi, in tal caso richiede la creazione di un oggetto sezione. Allo stesso modo, se il server stabilisce che le risposte sono lunghe, provvede alla creazione di un oggetto sezione. Per usare gli oggetti sezione s'invia un breve messaggio contenente un puntatore e le informazioni sulle sue dimensioni. Questo metodo è un po' più complicato del primo, ma evita la copiatura dei dati. In entrambi i casi, se né il client né il server possono rispondere immediatamente alla richiesta, si può impiegare un meccanismo di *callback*; tale meccanismo consente di gestire i messaggi in modo asincrono. La struttura delle chiamate di procedura locali in Windows XP è illustrata nella Figura 3.17.

È importante notare che il meccanismo LPC di Windows XP non è parte della API Win32: quindi, non è accessibile ai programmati di applicazioni. Le applicazioni Win32 devono comunque usare chiamate di procedura remota; nel caso in cui la chiamata si riferisca a un processo residente sulla stessa macchina del chiamante, il sistema la implementa tramite una chiamata locale. Il meccanismo LPC è anche usato per implementare qualche altra funzione della API Win32.

## 3.6 Comunicazione nei sistemi client-server

Nel Paragrafo 3.4 ci siamo soffermati su come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione tra sistemi client/server (Paragrafo 1.12.2). Consideriamo qui altre tre strategie: socket, chiamate di procedura remote (RPC) e invocazione di metodi remoti (RMI).

### 3.6.1 Socket

Una *socket* è definita come l'estremità di un canale di comunicazione. Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo IP concatenato a un numero di porta. In generale, le

socket impiegano un'architettura client-server; il server attende le richieste dei client, stando in ascolto a una porta specificata; quando il server riceve una richiesta, se accetta la connessione proveniente dalla socket del client, si stabilisce la comunicazione. I server che svolgono servizi specifici (come telnet, ftp, e http) stanno in ascolto a porte note (i server telnet alla porta 23, i server ftp alla porta 21, e i server Web o http alla porta 80). Tutte le porte al di sotto del valore 1024 sono considerate note e si usano per realizzare servizi standard.

Quando un processo client richiede una connessione, il calcolatore che lo esegue assegna una porta specifica, che consiste di un numero arbitrario maggiore di 1024. Si supponga per esempio che un processo client presente nel calcolatore X con indirizzo IP 146.86.5.20 voglia stabilire una connessione con un server Web (in ascolto alla porta 80) all'indirizzo 161.25.19.8; il calcolatore X potrebbe assegnare al client, per esempio, la porta 1625. La connessione sarebbe composta di una coppia di socket: (146.86.5.20:1625) nel calcolatore X e (161.25.19.8:80) nel server Web. La Figura 3.18 mostra questa situazione. La consegna dei pacchetti al processo giusto avviene secondo il numero della porta di destinazione.

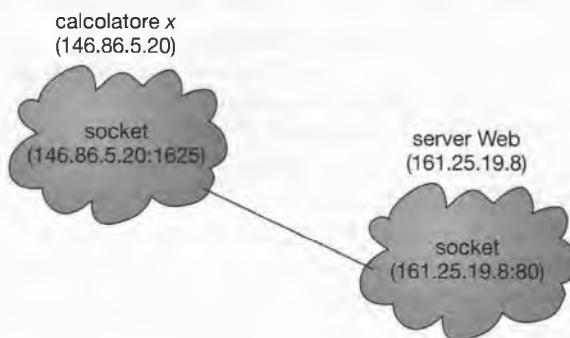
Tutte le connessioni devono essere uniche; quindi, se un altro processo, nel calcolatore X, vuole stabilire un'altra connessione con lo stesso server Web, riceve un numero di porta maggiore di 1024 e diverso da 1625. Ciò assicura che ciascuna connessione sia identificata da una distinta coppia di socket.

Sebbene la maggior parte degli esempi di programmazione di questo testo sia scritta in C, le socket sono illustrate usando il linguaggio Java, poiché offre un'interfaccia alle socket più semplice e dispone di una ricca libreria di strumenti di rete. Il lettore interessato alla programmazione con le socket in C o C++ può consultare le note bibliografiche alla fine del capitolo.

Il linguaggio Java prevede tre tipi differenti di socket: quelle **orientate alla connessione** (TCP) sono realizzate con la classe `Socket`; quelle **prive di connessione** (UDP) usano la classe `DatagramSocket`; il terzo tipo di socket è basato sulla classe `MulticastSocket`; si tratta di una sottoclasse della classe `DatagramSocket` che permette l'invio simultaneo dei dati a diversi destinatari (*multicast*).

Si consideri un esempio in cui un server usa socket TCP orientate alla connessione per fornire l'ora e la data correnti ai client. Il server si pone in ascolto alla porta 6013 – questo intero è arbitrario, purché maggiore di 1024. Quando giunge una richiesta di connessione, il server restituisce la data e l'ora al client.

Il codice del server è mostrato nella Figura 3.19. Il server crea una `ServerSocket` che ascolta alla porta 6013. Il server si pone in ascolto tramite la chiamata bloccante `accept()`, e rimane in attesa fino all'arrivo della richiesta di un client. A quel punto, `accept()` restituisce il numero della socket che il server può usare per comunicare con il client.



**Figura 3.18** Comunicazione tramite socket.

```

import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);

            // si pone in ascolto di richieste di connessione
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream(), true);

                // scrive la Data sulla socket
                pout.println(new java.util.Date().toString());

                // chiude la socket e ritorna in ascolto
                // di nuove richieste
                client.close();
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

**Figura 3.19** Server che comunica ai client la data corrente.

Ecco alcuni dettagli relativi all'uso da parte del server della socket connessa al client. Il server crea da principio un oggetto di classe `PrintWriter` che gli permette di scrivere sulla socket tramite i metodi `print()` e `println()`. Esso manda quindi la data e l'ora al client scrivendo sulla socket tramite `println()`; a questo punto, chiude la socket di comunicazione con il client e torna in attesa di nuove richieste.

Un client comunica con il server creando una socket e collegandosi per suo tramite alla porta su cui il server è in ascolto. L'implementazione è mostrata nella Figura 3.20. Il client crea una `Socket` e richiede una connessione al server alla porta 6013 dell'indirizzo IP 127.0.0.1. Stabilita la connessione, il client può leggere dalla socket tramite le ordinarie istruzioni di I/O. Dopo aver ricevuto la data dal server, il client chiude la socket e termina. L'indirizzo IP 127.0.0.1, noto come **loopback**, è peculiare: è usato da una macchina per riferirsi a se stessa. Tramite questo stratagemma, un client e un server residenti sulla stessa macchina sono in grado di comunicare tramite il protocollo TCP/IP. Nell'esempio, si può sostituire l'indirizzo loopback con l'indirizzo IP di una qualunque macchina che ospiti il server della Figura 3.19. È anche possibile usare un nome simbolico, come www.westminstercollege.edu.

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            //si collega alla porta su cui ascolta il server
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            // legge la data dalla socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            // chiude la connessione tramite socket
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

**Figura 3.20** Client che si collega al server nella Figura 3.19.

La comunicazione tramite socket è diffusa ed efficiente, ma è considerata una forma di comunicazione fra sistemi distribuiti a basso livello. Infatti, le socket permettono unicamente la trasmissione di un flusso non strutturato di byte: è responsabilità del client e del server interpretare e organizzare i dati in forme più complesse. Nei due paragrafi successivi sono illustrati due metodi di comunicazione di più alto livello, le chiamate di procedure remote (RPC) e le invocazioni di metodi remoti (RMI).

### 3.6.2 Chiamate di procedure remote

Uno tra i più diffusi tipi di servizio remoto è il paradigma della RPC, presentato brevemente nel Paragrafo 3.5.2. La RPC è stata progettata per astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete. Per molti aspetti è simile al meccanismo IPC descritto nel Paragrafo 3.4, ed è generalmente costruita su un sistema di questo tipo. Poiché in un sistema distribuito si eseguono i processi su sistemi distinti, per offrire un servizio remoto occorre impiegare uno schema di comunicazione basato sullo scambio di messaggi. Contrariamente a quel che accade nella funzione IPC, i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone di RPC, in ascolto a una porta del sistema remoto, e contengono un identificatore della

funzione da eseguire e i parametri da inviare a tale funzione. Nel sistema remoto si esegue questa funzione e s'invia ogni risultato al richiedente in un messaggio distinto.

La *porta* è semplicemente un numero inserito all'inizio dei pacchetti di messaggi. Mentre un singolo sistema ha normalmente un solo indirizzo di rete, all'interno dell'indirizzo può avere molte porte che servono a distinguere i numerosi servizi di rete che può fornire. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente; per esempio, per permettere ad altri di ottenere un elenco dei suoi attuali utenti, un sistema deve possedere un demone in ascolto a una porta, per esempio la porta 3027, che realizzzi una siffatta RPC. Qualsiasi sistema remoto può ottenere l'informazione richiesta, vale a dire l'elenco degli utenti, inviando un messaggio RPC alla porta 3027 del server; i dati si ricevono in un messaggio di risposta.

La semantica delle RPC permette a un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Il sistema delle RPC nasconde i dettagli necessari che consentono la comunicazione, assegnando un **segmento di codice di riferimento** (*stub*) alla parte client. Esiste in genere un segmento di codice di riferimento per ogni diversa procedura remota. Quando il client la invoca, il sistema delle RPC richiama l'appropriato segmento di codice di riferimento, passando i parametri della procedura remota. Il segmento di codice di riferimento individua la porta del server e struttura i parametri; la strutturazione dei parametri (*marshalling*) implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Il segmento di codice di riferimento quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo segmento di codice di riferimento nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica.

Una questione da affrontare riguarda le differenze nella rappresentazione dei dati nel client e nel server. Si consideri la rappresentazione a 32 bit dei numeri interi; alcuni sistemi, noti come *big-endian*, usano l'indirizzo di memoria maggiore per contenere il byte più significativo; altri, noti come *little-endian*, lo usano per contenere il byte meno significativo. [Questa terminologia deriva dall'analogia di Cohen (Cohen, D., "On Holy Wars and a Plea for Peace", *IEEE Computer Magazine*, vol. 14, pagg. 48-54, ottobre 1981) con la controversia raccontata nei *Viaggi di Gulliver*, Capitolo IV, riguardante l'estremità – larga o stretta – da cui si dovessero rompere le uova per berle.] Per risolvere questo problema, molti sistemi di RPC definiscono una rappresentazione dei dati indipendente dalla macchina. Uno di questi sistemi di rappresentazione è noto come **rappresentazione esterna dei dati** (*external data representation*, XDR). Nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato XDR; nel server, i dati nel formato XDR si convertono nel formato della macchina server.

Un'altra questione importante riguarda la semantica delle chiamate. Infatti, mentre le chiamate locali falliscono in circostanze estreme, le RPC possono non riuscire, o risultare duplicate e dunque eseguite più volte, semplicemente a causa di comuni errori della rete. Un modo per affrontare il problema è di far sì che il sistema operativo agisca sui messaggi *esattamente una volta*, e non *al massimo una volta*. Questo tipo di semantica è comune per le chiamate locali, ma è più difficile da implementare per le RPC.

Si consideri prima la semantica "al massimo una volta". Essa può essere implementata marcando ogni messaggio con la sua ora di emissione (*timestamp*). Il server dovrà mantenere l'archivio di tutti gli orari di emissione dei messaggi già ricevuti e trattati, o perlomeno un archivio che sia abbastanza ampio da identificare i messaggi duplicati. I messaggi in entrata con orario di emissione già presente nell'archivio sono ignorati. I client avranno allora la si-

curezza che la procedura remota sarà eseguita al massimo una volta, anche nel caso di un invio di più copie di uno stesso messaggio. (La generazione dell'orario di emissione è analizzata nel Paragrafo 18.1.)

Per implementare la semantica “esattamente una volta”, occorre eliminare il rischio che il server non riceva mai la richiesta. Per ottenere questo risultato, il server deve implementare la semantica “al massimo una volta”, ma integrarla con l’invio al client di una ricevuta che attesti l’avvenuta esecuzione della procedura. Ricevute di questo tipo sono molto diffuse nella comunicazione tramite reti. Il client dovrà inviare periodicamente la richiesta RPC finché non ottenga la relativa ricevuta.

Un altro argomento importante riguarda la comunicazione tra server e client. Con le ordinarie chiamate di procedure, durante la fase di collegamento, caricamento o esecuzione di un programma (Capitolo 8), ha luogo una forma di associazione che sostituisce il nome della procedura chiamata con l’indirizzo di memoria della procedura stessa. Lo schema delle RPC richiede una corrispondenza di questo genere tra il client e la porta del server; esiste tuttavia il problema del riconoscimento dei numeri delle porte del server da parte del client. Nessun sistema dispone d’informazioni complete sugli altri sistemi, poiché essi non condividono memoria.

Per risolvere questo problema s’impiegano per lo più due metodi. Con il primo, l’informazione sulla corrispondenza tra il client e la porta del server si può predeterminare fissando gli indirizzi delle porte: una RPC si associa nella fase di compilazione a un numero di porta fisso; il server non può modificare il numero di porta del servizio richiesto. Con il secondo metodo la corrispondenza si può effettuare dinamicamente tramite un meccanismo di *rendezvous*. Generalmente il sistema operativo fornisce un demone di rendezvous (*matchmaker*) a una porta di RPC fissata. Un client invia un messaggio, contenente il nome della RPC, al demone di rendezvous per richiedere l’indirizzo della porta della RPC da eseguire. Il demone risponde col numero di porta, e la richiesta d’esecuzione della RPC si può inviare a quella porta fino al termine del processo (o fino alla caduta del server). Questo metodo richiede un ulteriore carico a causa della richiesta iniziale, ma è più flessibile del primo metodo. La Figura 3.21 illustra un esempio d’interazione.

Lo schema della RPC è utile nella realizzazione di un file system distribuito (Capitolo 17); un sistema di questo tipo si può realizzare come un insieme di demoni e client di RPC. I messaggi s’indirizzano alla porta del file system distribuito su un server in cui deve avvenire l’operazione sui file. I messaggi contengono le operazioni da svolgere nei dischi: `read`, `write`, `rename`, `delete` o `status`, corrispondenti alle normali chiamate di sistema che si usano per i file. Il messaggio di risposta contiene i dati risultanti da quella chiamata, che il demone del DFS esegue su incarico del client. Un messaggio può, per esempio, contenere una richiesta di trasferimento di un intero file a un client, oppure semplici richieste di blocchi. Nel secondo caso per trasferire un intero file possono essere necessarie parecchie richieste di questo tipo.

### 3.6.3 Pipe

Una pipe agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi (IPC) nei sistemi UNIX e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l’uno con l’altro, sebbene con qualche limitazione. Quando si implementa una pipe devono essere prese in considerazione quattro questioni.

1. La comunicazione permessa dalla pipe è unidirezionale o bidirezionale?

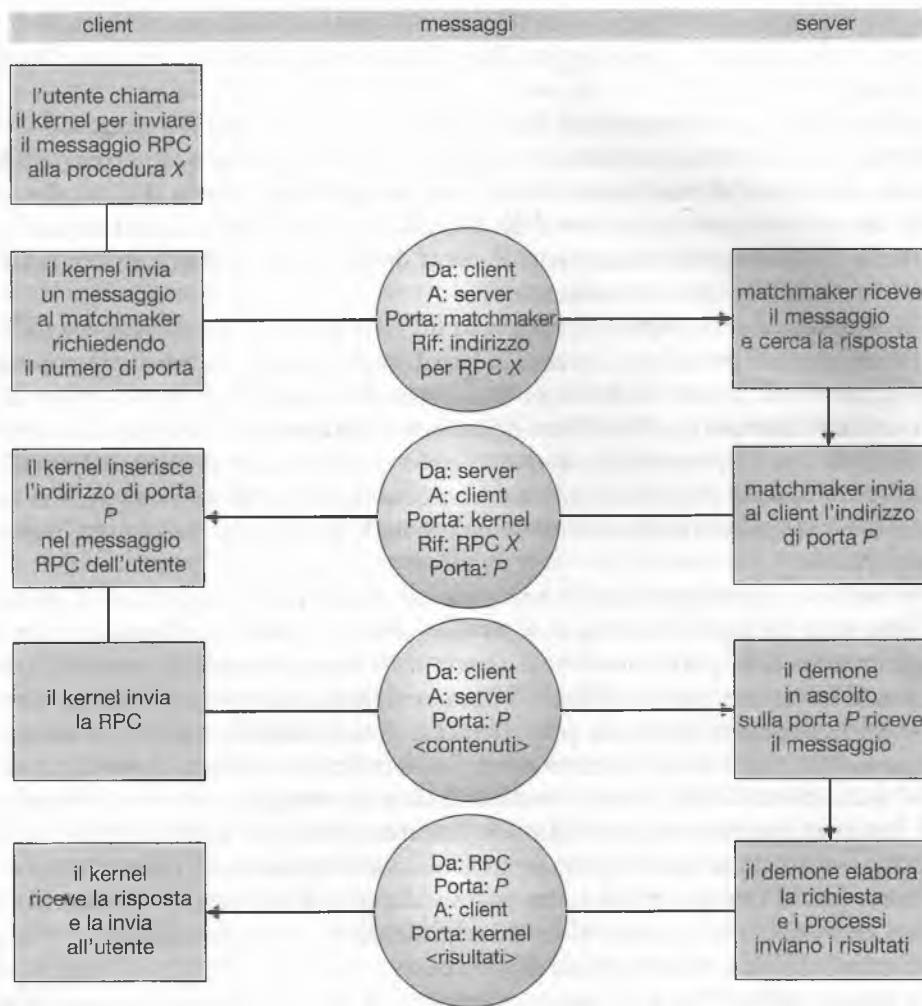


Figura 3.21 Esecuzione di una chiamata di procedura remota (RPC).

2. Se è ammessa la comunicazione a doppio senso, essa è di tipo *half duplex* (i dati possono viaggiare in un'unica direzione alla volta) o *full duplex* (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
3. Deve esistere una relazione (del tipo *padre-figlio*) tra i processi in comunicazione?
4. Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

Nei paragrafi seguenti esploriamo due tipi comuni di pipe utilizzate sia in UNIX sia in Windows.

### 3.6.3.1 Pipe convenzionali

Le pipe convenzionali permettono a due processi di comunicare secondo una modalità standard chiamata del produttore-consamatore. Il produttore scrive a una estremità del canale (l'estremità dedicata alla scrittura, o *write-end*) mentre il consumatore legge dall'altra estremità (l'estremità dedicata alla lettura, o *read-end*). Le pipe convenzionali sono quindi unidirezionali, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la

comunicazione a doppio senso devono essere utilizzate due *pipe*, ognuna delle quali manda i dati in una differente direzione. Illustreremo la costruzione di pipe convenzionali sia in UNIX sia in Windows. In entrambi i programmi di esempio un processo scrive sulla pipe il messaggio **Greetings**, mentre l'altro lo legge dall'altra estremità della pipe.

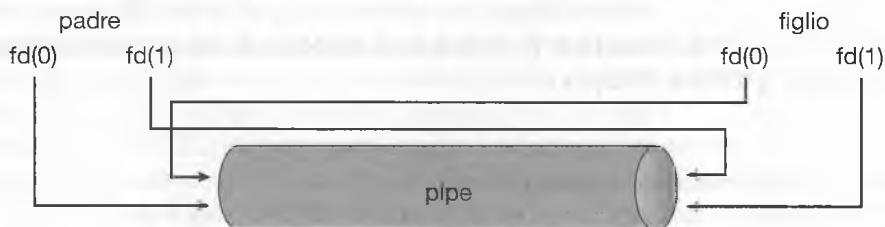
Nel sistema UNIX le pipe convenzionali sono costruite utilizzando la funzione

```
pipe(int fd[])
```

Essa crea una pipe alla quale si può accedere tramite i descrittori del file `int fd[ ]: fd[ 0 ]` è l'estremità dedicata alla lettura, mentre `fd[ 1 ]` è l'estremità dedicata alla scrittura. Il sistema UNIX considera una pipe come un tipo speciale di file; si può così accedere alle pipe tramite le usuali chiamate di sistema `read()` e `write()`.

Non si può accedere a una pipe al di fuori del processo che la crea. Solitamente un processo padre crea una pipe e la utilizza per comunicare con un processo figlio generato con il comando `fork()`. Come già indicato nel Paragrafo 3.3.1, il processo figlio eredita i file aperti dal processo padre. Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre. La Figura 3.22 illustra la relazione dei descrittori di file `fd` rispetto ai processi padre e figlio.

Nel programma UNIX mostrato nella Figura 3.23 (che continua nella Figura 3.24) il processo padre crea una pipe e in seguito esegue una chiamata `fork()`, generando un pro-



**Figura 3.22** Descrittori di file per una pipe convenzionale.

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

#define BUFFER_SIZE 25
#define READ END 0
#define WRITE END 1

int main(void)
{
    char write msg[BUFFER_SIZE] = "Greetings";
    char read msg[BUFFER_SIZE];
    int fd[2];
    pid t pid;
```

Il programma continua nella Figura 3.24

**Figura 3.23** Pipe convenzionali in UNIX.

```

/* crea la pipe */
if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
}

/* crea tramite fork un processo figlio */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
}

if (pid > 0) { /* processo padre */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[READ END]);

    /* scrive sulla pipe */
    write(fd[WRITE END], write msg, strlen(write msg)+1);

    /* chiude l'estremità della pipe dedicata alla scrittura */
    close(fd[WRITE END]);
}

else { /* processo figlio */
    /* chiude l'estremità inutilizzata della pipe */
    close(fd[WRITE END]);

    /* legge dalla pipe */
    read(fd[READ END], read msg, BUFFER SIZE);
    printf("read %s",read msg);

    /* chiude l'estremità della pipe dedicata alla lettura */
    close(fd[READ END]);
}

return 0;
}

```

**Figura 3.24** Continuazione del programma della Figura 3.23.

cesso figlio. Ciò che succede dopo la chiamata `fork()` dipende da come i dati fluiscono nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale. Sebbene il programma mostrato nella Figura 3.23 non richieda questa azione è importante assicurare che un processo che legge dalla pipe possa rilevare il ca-

```

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle, WriteHandle;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char message[BUFFER_SIZE] = "Greetings";
DWORD written;
    Il programma continua nella Figura 3.26
}

```

**Figura 3.25** Pipe anonime in Windows (processo padre).

rattere di terminazione EOF (`read()` restituisce 0) quando chi scrive ha chiuso la sua estremità della pipe.

Nei sistemi Windows le pipe convenzionali sono denominate **pipe anonime** e si comportano analogamente alle loro equivalenti in UNIX: sono unidirezionali e utilizzano relazioni del tipo padre-figlio tra i processi coinvolti nella comunicazione. Inoltre, l'attività di lettura e scrittura sulla pipe può essere eseguita con le usuali funzioni `ReadFile()` e `WriteFile()`. L'API Win32 per creare le pipe è la funzione `CreatePipe()`, che riceve in ingresso quattro parametri: (1) handle separati per leggere e (2) scrivere sulla pipe, oltre a (3) una istanza della struttura `STARTUPINFO`, usata per specificare che il processo figlio erediti gli handle della pipe. Inoltre, può essere specificata (4) la dimensione della pipe (in byte).

La Figura 3.25 (e la continuazione nella Figura 3.26) illustrano un processo padre che crea una pipe anonima per comunicare con il proprio figlio. A differenza dei sistemi UNIX, dove un processo figlio eredita automaticamente una pipe dal proprio padre, Windows richiede al programmatore di specificare quali attributi saranno ereditati dal processo figlio. Ciò avviene innanzitutto inizializzando la struttura `SECURITY_ATTRIBUTES` per permettere che gli handle siano ereditati e poi reindirizzando lo standard input o lo standard output del processo figlio verso l'handle di scrittura o di lettura della pipe, rispettivamente. Dato che il figlio leggerà dalla pipe, il padre deve reindirizzare lo standard input del figlio verso l'handle di lettura della pipe stessa. Inoltre, dal momento che le pipe sono half duplex, è necessario proibire al figlio di ereditare l'handle di scrittura della pipe. La creazione del processo figlio avviene come nel programma nella Figura 3.12, fatta eccezione per il quinto parametro che in questo caso viene impostato al valore `TRUE` per indicare che il processo figlio eredita degli handle specifici dal proprio padre. Prima di scrivere sulla pipe, il padre ne chiude l'estremità di lettura, che è inutilizzata. Il processo figlio che legge dalla pipe è mostrato nella Figura 3.27. Prima di leggere dalla pipe, questo programma ottiene l'handle di lettura invocando `GetStdHandle()`.

Si noti bene che le pipe convenzionali richiedono una relazione di parentela padre-figlio tra i processi comunicanti, sia in UNIX sia in Windows. Ciò significa che queste pipe possono essere utilizzate soltanto per la comunicazione tra processi in esecuzione sulla stessa macchina.

```

/*imposta gli attributi di sicurezza in modo che le pipe siano
ereditate */
SECURITY_ATTRIBUTES sa = {sizeof(SECURITY_ATTRIBUTES),NULL,TRUE};
/* alloca la memoria */
ZeroMemory(&pi, sizeof(pi));

/* crea la pipe */
if (!CreatePipe(&ReadHandle, &WriteHandle, &sa, 0)) {
    fprintf(stderr, "Create Pipe Failed");
    return 1;
}

/* prepara la struttura START_INFO per il processo figlio */
GetStartupInfo(&si);
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);

/* reindirizza lo standard input verso l'estremità della pipe
dedicata alla lettura */
si.hStdInput = ReadHandle;
si.dwFlags = STARTF_USESTDHANDLES;

/* non permette al processo figlio di ereditare l'estremità
della pipe dedicata alla scrittura */
SetHandleInformation(WriteHandle, HANDLE_FLAG_INHERIT, 0);

/* crea il processo figlio */
CreateProcess(NULL, "child.exe", NULL, NULL,
    TRUE, /* inherit handles */
    0, NULL, NULL, &si, &pi);

/* chiude l'estremità inutilizzata della pipe */
CloseHandle(ReadHandle);

/* il padre scrive sulla pipe */
if (!WriteFile(WriteHandle, message,BUFFER_SIZE,&written,NULL))
    fprintf(stderr, "Error writing to pipe.");

/* chiude l'estremità della pipe dedicata alla scrittura */
CloseHandle(WriteHandle);

/* attende la terminazione del processo figlio */
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
return 0;
}

```

**Figura 3.26** Continuazione del programma nella Figura 3.25.

```

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

#define BUFFER_SIZE 25

int main(VOID)
{
HANDLE ReadHandle;
CHAR buffer[BUFFER_SIZE];
DWORD read;

/* riceve l'handle di lettura della pipe */
ReadHandle = GetStdHandle(STD_INPUT_HANDLE);

/* il figlio legge dalla pipe */
if (ReadFile(ReadHandle, buffer, BUFFER_SIZE, &read, NULL))
    printf("child read %s",buffer);
else
    fprintf(stderr, "Error reading from pipe");
return 0;
}

```

**Figura 3.27** Pipe anonime in Windows (processo figlio).

### 3.6.3.2 Named pipe

Le pipe convenzionali offrono un meccanismo semplice di comunicazione tra una coppia di processi. Tuttavia, le pipe convenzionali esistono solo mentre i processi stanno comunicando. Sia in UNIX sia in Windows, una volta che i processi hanno terminato di comunicare, le pipe convenzionali cessano di esistere.

Le **named pipe** costituiscono uno strumento di comunicazione molto più potente; la comunicazione può essere bidirezionale, e la relazione di parentela padre-figlio non è necessaria. Una volta che si sia creata la named pipe, diversi processi possono utilizzarla per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori. In più, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. Sia UNIX sia Windows mettono a disposizione le named pipe, nonostante ci siano grandi differenze nei dettagli dell'implementazione. Di seguito, prendiamo in esame le named pipe in ciascuno dei due sistemi.

Nei sistemi UNIX le named pipe sono dette FIFO. Una volta create, esse appaiono come normali file all'interno del file system. Una FIFO viene creata mediante una chiamata di sistema `mkfifo()` e viene poi manipolata con le usuali chiamate di sistema `open()`, `read()`, `write()` e `close()`; essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system. Nonostante i file FIFO permettano la comunicazione bidirezionale, l'unica tipologia di trasmissione consentita è quella half duplex. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due FIFO. Per utilizzare le FIFO i processi comunicanti devono risiedere sulla stessa macchina: se è richiesta la comunicazione tra più macchine devono essere impiegate le socket (si veda il Paragrafo 3.6.1).

Rispetto alle loro controparti in UNIX, le named pipe su un sistema Windows offrono un meccanismo di comunicazione più ricco. È permessa la comunicazione full duplex e i

### LE PIPE IN PRATICA

Le pipe sono usate abbastanza spesso dalla riga di comando di UNIX in situazioni nelle quali l'output di un comando serve da input per un secondo comando. Ad esempio, il comando `ls` di UNIX elenca il contenuto di una directory. Per directory particolarmente grandi, l'output può scorrere su diverse schermate. Il comando `more` gestisce l'output mostrando solo una schermata alla volta; l'utente deve premere la barra spaziatrice per muoversi da una schermata all'altra. Istituendo una pipe tra i comandi `ls` e `more` (in esecuzione come singoli processi) si fa in modo che l'output di `ls` venga inviato all'input di `more`, permettendo così all'utente di vedere il contenuto di una grande directory una schermata alla volta. Dalla riga di comando si può costruire una pipe utilizzando il carattere `|`. Il comando completo è quindi

```
ls | more
```

In questo scenario, il comando `ls` funge da produttore, e il suo output è consumato dal comando `more`.

I sistemi Windows offrono un comando `more` per la shell DOS con una funzionalità analoga al corrispettivo di UNIX. Anche la shell DOS utilizza il carattere `|` per creare una pipe. L'unica differenza è costituita dal fatto che, per restituire il contenuto di una directory, DOS utilizza il comando `dir` anziché `ls`. Il comando equivalente in DOS è quindi

```
dir | more
```

processi comunicanti possono risiedere sia sulla stessa macchina sia su macchine diverse. Inoltre, attraverso una FIFO di UNIX possono essere trasmessi solo dati byte-oriented, mentre i sistemi Windows permettono la trasmissione di dati sia byte-oriented sia message-oriented. Le named pipe vengono create con la funzione `CreateNamedPipe()` e un client può connettersi a una named pipe tramite `ConnectNamedPipe()`. La comunicazione attraverso le named pipe avviene grazie alle funzioni `ReadFile()` e `WriteFile()`.

## 3.7 Sommario

Un processo è un programma in esecuzione. Nel corso delle sue attività, un processo cambia stato, e tale stato è definito dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati: nuovo, pronto, esecuzione, attesa o arresto. In un sistema operativo ogni processo è rappresentato dal proprio blocco di controllo del processo (PCB).

Un processo, quando non è in esecuzione, è inserito in una coda d'attesa. Le due classi principali di code in un sistema operativo sono le code di richieste di I/O e la coda dei processi pronti per l'esecuzione, quest'ultima contenente tutti i processi pronti per l'esecuzione che si trovano in attesa della CPU. Ogni processo è rappresentato da un PCB; i PCB si possono collegare tra loro in modo da formare una coda dei processi pronti. Lo scheduling a lungo termine (o *job scheduling*) consiste nella scelta dei processi che si contenderanno la CPU. Normalmente lo scheduling a lungo termine è influenzato in modo consistente da considerazioni riguardanti l'assegnazione delle risorse, in particolar modo quelle concernenti la gestione della memoria. Lo scheduling a breve termine (o scheduling della CPU) consiste nella selezione di un processo dalla coda dei processi pronti.

I sistemi operativi devono implementare un meccanismo per generare processi figli da un processo genitore. Genitore e figli possono girare in concomitanza, oppure il genitore po-

trà essere posto in attesa della terminazione dei figli. L'esecuzione concorrente è ampiamente giustificabile dalla necessità di condividere informazioni e dal potenziale aumento della velocità di calcolo, oltre che da considerazioni legate alla modularità e alla convenienza.

I processi in esecuzione nel sistema operativo possono essere indipendenti o cooperanti. I processi cooperanti devono avere i mezzi per comunicare tra loro. Fondamentalmente esistono due schemi complementari di comunicazione: memoria condivisa e scambio di messaggi. Nel metodo con memoria condivisa i processi in comunicazione devono condividere alcune variabili, per mezzo di cui i processi scambiano informazioni; il compito della comunicazione è lasciato ai programmatori di applicazioni; il sistema operativo deve semplicemente offrire la memoria condivisa. Il metodo con scambio di messaggi permette di compiere uno scambio di messaggi tra i processi; in questo caso il compito di attuare la comunicazione è del sistema operativo. Questi due schemi non sono mutuamente esclusivi, e si possono impiegare insieme in uno stesso sistema.

La comunicazione nei sistemi client/server può impiegare (1) socket, (2) chiamate di procedure remote (RPC) o (3) chiamate di metodi remoti (RMI). Una connessione tra una coppia di applicazioni consiste di una coppia di socket, ciascuna a un'estremità del canale di comunicazione. Le RPC sono un'altra forma di comunicazione distribuita: una RPC si verifica quando un processo (o un thread) invoca una procedura in un'applicazione remota. Le RMI sono la versione del linguaggio Java delle RPC; consentono a un thread di invocare un metodo su un oggetto remoto esattamente come se si trattasse di oggetto locale. La differenza principale tra le RPC e le RMI consiste nel fatto che i dati passati a una procedura remota hanno la forma di un'ordinaria struttura dati, mentre le RMI consentono anche il passaggio di oggetti.

## Esercizi pratici

- 3.1 Palm OS non offre strumenti per gestire processi concorrenti. Esponete tre principali complicazioni che la gestione di processi concorrenti crea al sistema operativo.
- 3.2 Il processore Sun UltraSPARC ha diversi insiemi di registri. Descrivete ciò che avviene quando si ha un cambio di contesto nel caso in cui il contesto successivo sia già caricato in un determinato insieme di registri. Che cosa succede se il contesto successivo è nella memoria (invece che nei registri) e tutti i registri sono in uso?
- 3.3 Quando un processo crea un nuovo processo utilizzando l'istruzione `fork()`, quale dei seguenti stati è condiviso tra il processo padre e il processo figlio?
  - a. Stack.
  - b. Heap.
  - c. Segmenti di memoria condivisi.
- 3.4 A proposito del meccanismo RPC, considerate la semantica "exactly once". L'algoritmo che implementa questa semantica funziona correttamente anche se il messaggio ACK che restituisce al client va perso a causa di un problema di rete? Descrivete la sequenza di messaggi scambiati e prendete nota di quando la semantica "exactly once" viene ancora preservata.
- 3.5 Assumendo che un sistema distribuito sia sensibile a malfunzionamento del server, quali meccanismi sarebbero richiesti per garantire la semantica "exactly once" per l'esecuzione di RPC?
- 3.6 Descrivete le differenze tra scheduling a breve termine, a medio termine e a lungo termine.

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

int main()
{
    /* crea mediante fork un processo figlio */
    fork();

    /* crea un altro processo figlio */
    fork();

    /* e ne crea un altro ancora */
    fork();

    return 0;
}
```

**Figura 3.28** Quanti processi vengono creati?

- 3.7 Descrivete le azioni intraprese dal kernel nell'esecuzione di un cambio di contesto tra processi.
- 3.8 Costruite un albero di processi simile a quello nella Figura 3.9. Utilizzate il comando **ps -ael** per ottenere informazioni sui processi in UNIX o in Linux. Utilizzate il comando **man ps** per ottenere più informazioni sul comando **ps**. Per i sistemi Windows, dovrete usare il task manager.
- 3.9 Considerando anche il processo padre iniziale, quanti processi vengono creati dal programma della Figura 3.28?
- 3.10 In riferimento al programma nella Figura 3.29, identificate i valori dei **pid** alle linee A, B, C e D. (Assumete che i **pid** attuali del padre e del figlio siano rispettivamente 2600 e 2603). Si veda la Figura 3.30.
- 3.11 Fornite un esempio di situazione nella quale le pipe convenzionali siano più adatte delle named pipe e un esempio di situazione nella quale le named pipe siano invece più indicate delle pipe convenzionali.
- 3.12 In riferimento al meccanismo RPC, descrivete i possibili effetti negativi della mancata implementazione della semantiche “al massimo una volta” o di quella “esattamente una volta”. Considerate una possibile applicazione di un tale meccanismo che non le implementi.
- 3.13 Descrivete l'output del programma nella Figura 3.30 alla RIGA A.
- 3.14 Analizzate vantaggi e svantaggi delle tecniche elencate di seguito, considerando sia il punto di vista del sistema sia quello del programmatore.
  - a. Comunicazione sincrona e asincrona.
  - b. Gestione automatica o esplicita del buffer.
  - c. Trasmissione per copia e trasmissione per riferimento.
  - d. Messaggi a lunghezza fissa e variabile.

```

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

int main()
{
pid t pid, pid1;

/* crea mediante fork un processo figlio */
pid = fork();

if (pid < 0) { /* errore */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* processo figlio */
    pid1 = getpid();
    printf("child: pid = %d",pid); /* A */
    printf("child: pid1 = %d",pid1); /* B */
}
else { /* processo padre */
    pid1 = getpid();
    printf("parent: pid = %d",pid); /* C */
    printf("parent: pid1 = %d",pid1); /* D */
    wait(NULL);
}

return 0;
}

```

**Figura 3.29** Quali sono i valori dei pid?

## Problemi di programmazione

- 3.15 La successione di Fibonacci comincia con 0, 1, 1, 2, 3, 5, 8, ... . La sua definizione ricorsiva è:

$$\begin{aligned}
 fib_0 &= 0 \\
 fib_1 &= 1 \\
 fib_n &= fib_{n-1} + fib_{n-2}
 \end{aligned}$$

Scrivete un programma C che usi la chiamata di sistema `fork()` per generare la successione di Fibonacci all'interno del processo figlio. Il numero di termini da generare sarà specificato dalla riga di comando. Per esempio, se il parametro passato dalla riga di comando fosse 5, il programma dovrebbe far produrre in uscita al processo figlio i primi 5 termini della successione di Fibonacci. Visto che genitore e figlio hanno copie private dei dati, sarà il figlio a dover produrre i dati in uscita. Il genitore dovrà rimanere in attesa tramite `wait()` fino alla terminazione del figlio. Implementate i necessari controlli per garantire che il valore in ingresso dalla riga di comando sia un intero non negativo.

```

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

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* processo figlio */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* processo padre */
        wait(NULL);
        printf("PARENT: value = %d", value); /* Linea A */
        return 0;
    }
}

```

**Figura 3.30** Quale sarà l'output della Linea A?

- 3.16 Riprendete l'esercizio precedente, usando però la API Win32 e la chiamata `CreateProcess()`. Dovrete specificare un programma a sé stante come parametro di `CreateProcess()`: esso sarà eseguito in qualità di processo figlio, e dovrà produrre in uscita la successione di Fibonacci. Implementate i necessari controlli per garantire che il valore in ingresso dalla riga di comando sia un intero non negativo.
- 3.17 Modificate il server nella Figura 3.19 in modo che fornisca aforismi casuali in luogo di data e ora. Gli aforismi possono richiedere più linee di testo. Il client nella Figura 3.20 può essere adattato alla lettura degli aforismi restituiti dal server.
- 3.18 Un server eco è un server che restituisce ai client esattamente ciò che essi gli inviano. Se un client, per esempio, invia al server la stringa *Ciao*, il server risponderà con gli stessi dati: ossia, invierà al client la stringa *Ciao*. Scrivete un server eco usando la API Java descritta nel Paragrafo 3.6.1. Il server rimarrà in attesa dei client tramite il metodo `accept()`. Dopo aver accettato una connessione, il client eseguirà il ciclo seguente:
  - ◆ leggerà i dati dalla socket connessa con il client, ponendoli in un buffer;
  - ◆ scriverà i contenuti del buffer sulla socket connessa con il client.

Il server uscirà dal ciclo solo dopo aver stabilito che il client ha chiuso la connessione. Il server nella Figura 3.19 impiega la classe Java `java.io.BufferedReader`, che estende la classe `java.io.Reader`, usata per leggere flussi di caratteri. Il server eco, però, potrebbe ricevere dati di altro tipo dal client – per esempio dati binari. La classe `java.io.InputStream` tratta i dati byte per byte, e non carattere per carattere. È quindi necessario che il server eco impieghi una classe che estenda `java.io.InputStream`. Il metodo `read()` di `java.io.InputStream` restituisce `-1` quando il client ha chiuso la connessione.

- 3.19 Nell'Esercizio 3.15 è il processo figlio a dover produrre in uscita la successione di Fibonacci, perché processo genitore e figlio possiedono copie private dei dati. Un approccio alternativo per la stesura di questo programma è impiegare un segmento di memoria condiviso tra genitore e figlio. In questo modo, il figlio potrà scrivere i numeri di Fibonacci richiesti in memoria condivisa, e il genitore potrà fornirli all'utente alla terminazione del figlio. A causa della condivisione della memoria, i dati in questione saranno accessibili al genitore anche dopo la terminazione del figlio. Il programma dovrà essere implementato con la API POSIX descritta nel Paragrafo 3.5.1. Per prima cosa, bisogna prevedere una struttura dati per il segmento di memoria condivisa. La soluzione più semplice è impiegare il costrutto C **struct**, dichiarando due elementi al suo interno: (1) un vettore di dimensione fissa **MAX\_SEQUENCE** che conterrà i valori della successione; e (2) il numero di termini **sequence\_size** che il figlio deve produrre, dove **sequence\_size**  $\leq$  **MAX\_SEQUENCE**. Ecco il codice relativo:

```
#define MAX_SEQUENCE 10

typedef struct {
    long fib_sequence[MAX_SEQUENCE];
    int sequence_size;
} shared_data;
```

Il processo genitore eseguirà i passi seguenti:

- accetterà il parametro passato dalla riga di comando, e controllerà che si tratti di un intero non negativo  $\leq$  **MAX\_SEQUENCE**;
- allocherà l'area di memoria condivisa di dimensione **shared\_data**;
- annetterà l'area condivisa al suo spazio degli indirizzi;
- imposterà il valore di **sequence\_size** al parametro di cui al punto a;
- genererà il processo figlio e invocherà **wait()** per attendere la sua terminazione;
- scriverà in uscita i termini della successione di Fibonacci presenti in memoria condivisa;
- eliminerà l'area di memoria condivisa dal suo spazio degli indirizzi e dal sistema.

Il processo figlio, in quanto copia del processo genitore, avrà accesso alla memoria condivisa. Il processo figlio scriverà gli opportuni termini della successione di Fibonacci in memoria condivisa, per poi eliminarla dal proprio spazio degli indirizzi e terminare l'esecuzione.

Uno dei problemi inerenti alla cooperazione fra processi è la loro sincronizzazione. In questo esercizio, il processo genitore non deve cominciare a scrivere i termini della successione in uscita prima della terminazione del figlio. La sincronizzazione è implementata tramite la chiamata **wait()**, che sospende il genitore fino alla terminazione del figlio.

- 3.20 Utilizzando pipe convenzionali, scrivete un programma nel quale un processo manda una stringa a un secondo processo, il quale cambia le lettere maiuscole del messaggio in minuscole, e viceversa, per poi restituire il risultato al primo processo. Ad esempio, se il primo processo manda il messaggio **Hi There**, il secondo restituirà **hI tHERE**. Ciò richiede l'utilizzo di due pipe, una per mandare il messaggio originale dal primo al secondo processo e l'altra per mandare il messaggio modificato dal secondo processo al primo. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

- 3.21 Scrivete un programma **FileCopy** per la copia di file, usando le pipe convenzionali. Questo programma riceverà in ingresso due parametri: il primo è il nome del file che deve essere copiato, il secondo è il nome del file copiato. Il programma creerà una pipe convenzionale e scriverà i contenuti del file da copiare nella pipe. Il processo figlio leggerà il file dalla pipe e lo scriverà nel suo file di destinazione. Se ad esempio invochiamo il programma come segue:

```
FileCopy input.txt copy.txt
```

il file **input.txt** sarà scritto sulla pipe. Il processo figlio leggerà i contenuti del file e li scriverà nel file di destinazione **copy.txt**. Potete scrivere il programma utilizzando pipe in Windows o in UNIX.

- 3.22 Molte implementazioni di UNIX e Linux offrono il comando **ipcs** per elencare lo stato dei meccanismi di comunicazione POSIX tra processi, riportando anche informazioni sulle aree di memoria condivisa. Il comando attinge molte delle informazioni dalla struttura **shmid\_ds**, contenuta nel file **/usr/include/sys/shm.h**. Alcuni dei campi sono:

- ◆ **int shm\_segsz** – la dimensione dell'area;
- ◆ **short shm\_nattch** – numero di annessioni eseguite;
- ◆ **struct ipc\_perm shm\_perm** – struttura contenente i permessi d'accesso all'area.

La struttura, che si trova nel file **/usr/include/sys/ipc.h**, contiene i campi:

- ◆ **unsigned short uid** – identificatore utente della memoria condivisa;
- ◆ **unsigned short mode** – modalità d'accesso;
- ◆ **key\_t key** (in Linux, **\_key**) – identificatore-chiave specificato dall'utente.

I valori delle modalità d'accesso dipendono dai parametri passati a **shmget()** al momento dell'allocazione della memoria condivisa. La tabella seguente riassume la codifica dei permessi nel campo **unsigned short mode**:

Modalità	Significato
0400	Permesso di lettura per il proprietario.
0200	Permesso di scrittura per il proprietario.
0040	Permesso di lettura per il gruppo.
0020	Permesso di scrittura per il gruppo.
0004	Permesso di lettura per chiunque.
0002	Permesso di scrittura per chiunque.

I permessi possono essere letti usando l'operatore **&**, che esegue un *AND* bit a bit degli operandi. Se, per esempio, l'espressione **mode & 0400** è valutata true, il proprietario del segmento condiviso ha il permesso di leggere dal segmento.

Le aree di memoria condivisa sono individuabili dal valore intero restituito da **shmget()**, o da un identificatore-chiave specificato dall'utente. La struttura **shm\_ds** relativa a un dato identificatore intero si può ottenere con la chiamata di sistema seguente:

```
/* identificatore del segmento di memoria condivisa*/
    int segment_id;
    shm_ds shmbuffer;

    shmctl(segment_id, IPC_STAT, &shmbuffer);
```

Se la chiamata ha successo, il valore restituito è zero, altrimenti è -1.

Scrivete un programma C che accetti in ingresso l'identificatore di un segmento di memoria condivisa. Il programma invocherà `shmctl()` per ottenere la struttura `shm_ds`, e stamperà i valori seguenti relativi al segmento associato all'identificatore in ingresso:

- ◆ ID del segmento.
- ◆ Identificatore-chiave.
- ◆ Modo d'accesso.
- ◆ UID del proprietario.
- ◆ Dimensione.
- ◆ Numero di annessioni.

## Progetti di programmazione

### 3.23 Passare messaggi POSIX

Questo progetto consiste nell'usare code di messaggi POSIX per comunicare temperature tra quattro processi esterni e un processo centrale. Il progetto può essere realizzato su sistemi che mettono a disposizione il passaggio di messaggi POSIX, come UNIX, Linux e Mac OS X.

#### Parte 1: Panoramica

Quattro processi esterni comunicheranno le temperature a un processo centrale, che a sua volta risponderà con la propria temperatura e indicherà se l'intero processo si è stabilizzato. Ogni processo riceverà la propria temperatura iniziale al momento della creazione e ricalcolerà una nuova temperatura secondo le seguenti formule:

$$\begin{aligned} \text{nuova temp processo esterno} &= \\ &(miaTemp * 3 + 2 * tempCentrale) / 5; \\ \text{nuova temp processo centrale} &= \\ &(2 * tempCentrale + quattro temp ricevute dai processi esterni) / 6; \end{aligned}$$

Inizialmente ciascun processo esterno manderà la propria temperatura alla mailbox del processo centrale. Se le quattro temperature coincidono con quelle spedite dai quattro processi durante l'ultima iterazione, il sistema si è stabilizzato. In questo caso il processo centrale notificherà a ogni processo esterno che il suo compito è terminato (insieme al compito del processo centrale stesso), e ogni processo emetterà la temperatura finale stabilizzata. Se il sistema non si è ancora stabilizzato, il processo centrale manderà la sua nuova temperatura alla mailbox di ognuno dei processi esterni e resterà in attesa delle risposte. I processi continueranno l'esecuzione finché le temperature non si saranno stabilizzate.

#### Parte 2: Sistema di passaggio dei messaggi

I processi possono scambiare messaggi usando quattro chiamate di sistema: `msgget()`, `msgsnd()`, `msgrcv()` e `msgctl()`. La funzione `msgget()` converte il nome di una mailbox in un identificatore numerico di una coda di messaggi, chiamato `msqid`. (Il nome di

una mailbox è il nome di una coda di messaggi condivisa tra i processi cooperanti). L'identificatore interno restituito da `msgget()`, `msqid`, deve essere passato a tutte le successive chiamate di sistema che utilizzano la coda di messaggi per realizzare la comunicazione tra processi. L'invocazione tipica di `msgget()` è la seguente:

```
msqid = msgget(1234, 0600 | IPC_CREAT);
```

Il primo parametro è il nome della mailbox; il secondo parametro istruisce il sistema operativo affinché crei la coda di messaggi se ancora non esiste, con privilegi di lettura e scrittura solo per processi con lo stesso user id del processo corrente. Se esiste già una coda di messaggi associata al nome di questa mailbox, `msgget()` restituisce lo `msqid` della mailbox esistente. Per evitare di collegarsi a una coda di messaggi preesistente, un processo può prima provare a collegarsi alla mailbox omettendo `IPC_CREAT` e poi controllare il valore restituito da `msgget()`. Se `msqid` è negativo, si è verificato un errore durante la chiamata di sistema, e può essere consultata la variabile ad accesso globale `errno` per determinare se l'errore è dovuto all'esistenza della coda di messaggi, oppure ad altre ragioni. Se il processo determina che la mailbox non esiste, la può creare, includendo `IPC_CREAT` nella chiamata. (Per questo progetto, la strategia appena descritta non dovrebbe essere necessaria se gli studenti lavorano su PC indipendenti o se utilizzano delle varianti specifiche di nomi di mailbox assegnate dal docente).

Una volta ottenuto uno `msqid` valido, un processo può cominciare a utilizzare `msgsnd()` per inviare messaggi e `msgrecv()` per riceverli. I messaggi hanno un formato simile a quello descritto nel Paragrafo 3.5.2: sono costituiti da una porzione a lunghezza fissa iniziale, seguita da una porzione a lunghezza variabile. Ovviamente, il mittente e il destinatario devono concordare un formato per i messaggi che intendono scambiarsi. Visto che il sistema operativo specifica un campo della porzione a lunghezza fissa di ciascun messaggio, e poiché almeno un'informazione sarà inviata al processo destinatario, è logico creare un'aggregazione di dati per ogni tipo di messaggio utilizzando una `struct`. Il primo campo di ciascuna `struct` deve essere un `long`, e contiene la priorità del messaggio. (Per questo progetto le priorità dei messaggi sono irrilevanti; raccomandiamo quindi che il primo campo di ciascun messaggio abbia sempre lo stesso valore intero, ad esempio 2). Gli altri campi dei messaggi contengono le informazioni da condividere tra i processi comunicanti. Sono necessari tre campi aggiuntivi: (1) la temperatura da comunicare, (2) il numero di processo del processo esterno che invia il messaggio (0 per il processo centrale) e (3) un flag impostato a 0, che il processo centrale imposterà a 1 quando il sistema diverrà stabile. Ecco una `struct` del tipo appena descritto:

```
struct {
    long priority;
    int temp;
    int pid;
    int stable;
} msgp;
```

Assumendo che `msqid` sia già stato creato, un possibile utilizzo di `msgsnd()` e `msgrecv()` è il seguente:

```
int stat, msqid;
stat = msgsnd(msqid, &msgp,
              sizeof(msgp)-sizeof(long), 0);
stat = msgrecv(msqid, &msgp,
               sizeof(msgp)-sizeof(long), 2, 0);
```

Il primo parametro in entrambe le chiamate di sistema deve essere uno `msqid` valido, altrimenti è restituito un valore negativo. (Entrambe le funzioni restituiscono il numero di byte inviati o ricevuti una volta che l'operazione è terminata con successo). Il secondo parametro è l'indirizzo dove trovare o memorizzare il messaggio da inviare o ricevere, seguito dal numero dei byte da inviare o ricevere. Il parametro finale 0 indica che le operazioni saranno sincrone e che il mittente si bloccherà se la coda di messaggi è piena. (Nel caso in cui fossero richieste operazioni asincrone, o se non si volesse il blocco del mittente, si utilizzerebbe `IPC_NOWAIT`). Ogni singola coda di messaggi può contenere un numero massimo di messaggi o di byte. È dunque possibile che la coda si riempia, ragion per cui un mittente può bloccarsi mentre tenta di trasmettere un messaggio). Il 2 che compare prima di questo parametro finale in `msgrecv()` indica il livello minimo di priorità dei messaggi che il processo desidera ricevere. Se si tratta di un'operazione sincrona, il destinatario aspetterà finché un messaggio di quella priorità (o di priorità maggiore) raggiunga la mailbox `msqid`.

Quando un processo ha terminato di utilizzare una coda di messaggi, questa deve essere rimossa, in modo che la mailbox possa essere sfruttata anche da altri processi. A meno che non venga rimossa, la coda di messaggi (e ogni messaggio che non è ancora stato ricevuto) rimarrà nello spazio di memoria riservato dal kernel a questa mailbox. Per rimuovere la coda di messaggi, e cancellare i messaggi non letti che vi sono salvati, è necessario invocare `msgctl()`, come segue:

```
struct msgid_ds dummyParam;
status = msgctl(msqid, IPC_RMID, &dummyParam);
```

Il terzo parametro è necessario perché la funzione lo richiede, ma viene utilizzato solamente se il programmatore desidera richiamare alcune statistiche a proposito dell'uso della coda di messaggi. Ciò avviene impostando `IPC_STAT` come secondo parametro.

Tutti i programmi dovrebbero includere i tre file d'intestazione seguenti, che si trovano in `/usr/include/sys`: `ipc.h`, `types.h`, e `msg.h`. Va menzionato, a questo punto, un dettaglio di implementazione della coda di messaggi che potrebbe forse indurre in confusione. Dopo che una mailbox viene rimossa attraverso `msgctl()`, i tentativi di creare un'altra mailbox con lo stesso nome utilizzando `msgget()` genereranno solitamente uno `msqid` differente.

### Parte 3: Creazione dei processi

Ciascun processo esterno, come pure il server centrale, creerà la propria mailbox con il nome  $X + i$ , dove  $i$  è un identificatore numerico. Per i processi esterni si avrà  $i = 1, \dots, 4$ , mentre per il processo centrale  $i = 0$ . Così, se  $X$  fosse 70, il processo centrale riceverebbe messaggi nella mailbox 70, e manderebbe le sue risposte alle mailbox 71-74. Il processo esterno 2 riceverebbe nella mailbox 72 e invierebbe alla mailbox 70, e così via. In questo modo ogni processo esterno si collega a due mailbox, e il processo centrale si collega a cinque mailbox. Se ciascun processo specifica `IPC_CREAT` quando invoca `msgget()`, il primo processo che invoca `msgget()` crea effettivamente la mailbox, mentre le chiamate successive a `msgget()` creano un collegamento alla mailbox esistente. Il protocollo per la rimozione dovrebbe fare in modo che la mailbox/coda di messaggi dove ogni processo è in ascolto sia l'unica che il processo rimuove, attraverso `msgctl()`.

Ogni processo esterno sarà identificato univocamente tramite un parametro passato dalla riga di comando. Il primo parametro da passare a ogni processo esterno sarà la sua temperatura iniziale; il secondo parametro sarà il suo id numerico: 1, 2, 3 oppure 4. Per il server centrale basterà un unico parametro, cioè la sua temperatura iniziale. Assumendo che

il nome eseguibile del processo esterno sia `external` e quello del server centrale sia `central`, è possibile invocare tutti e cinque i processi come segue:

```
./external 100 1 &
./external 22 2 &
./external 50 3 &
./external 40 4 &
./central 60 &
```

#### Parte 4: Suggerimenti per l'implementazione

È meglio cominciare a implementare l'invio di un singolo messaggio dal processo centrale a un dato processo esterno, e viceversa, prima di provare a scrivere tutto il codice per risolvere questo problema. Sarebbe anche consigliabile controllare tutti i valori restituiti dalle quattro chiamate di sistema che gestiscono le code di messaggi, per verificare le richieste non andate a buon fine, e visualizzare un messaggio appropriato sullo schermo dopo che ciascuna chiamata è stata eseguita con successo. Il messaggio dovrebbe indicare che cosa è stato eseguito e da chi, ad esempio, "la mailbox 71 è stata creata dal processo esterno 1", "messaggio proveniente dal processo esterno 2 ricevuto dal processo centrale", e così via. Questi messaggi possono essere rimossi o commentati dopo la risoluzione del problema. I processi dovranno anche verificare di aver ricevuto il numero corretto di parametri dalla riga di comando (grazie al parametro `argc` in `main()`). Infine, eventuali messaggi estranei che risiedono nella coda possono far apparire erronei alcuni processi cooperanti che funzionano correttamente. Per questa ragione, conviene rimuovere tutte le mailbox che hanno rilevanza per il progetto, in modo da assicurare che le mailbox siano vuote prima dell'inizio del processo stesso. Il modo più semplice per fare ciò è utilizzare il comando `ipcs` per ottenere un elenco di tutte le code di messaggi e poi il comando `ipcrm` per rimuovere tutte le code di messaggi esistenti. Il comando `ipcs` elenca lo `msqid` di tutte le code di messaggi presenti sul sistema. Si può quindi utilizzare `ipcrm` per rimuovere le code di messaggi aventi un determinato `msqid`. Ad esempio, se nell'output di `ipcs` appare `msqid 163845`, la coda avente id 163845 può essere cancellata con il seguente comando:

```
ipcrm -q 163845
```

## 3.8 Note bibliografiche

La comunicazione tra processi nel sistema RC 4000 è trattata in [Binch-Hansen 1970]. [Schlichting e Schneider 1982] descrivono le primitive per lo scambio asincrono di messaggi. Il meccanismo IPC implementato a livello utente è descritto da [Bershad et al. 1990].

[Gray 1977] presenta i dettagli della comunicazione tra processi in UNIX. [Barrera 1991] e [Vahalia 1996] descrivono la comunicazione tra processi nel sistema Mach. [Solomon e Russinovich 2000] e [Stevens 1999] delineano la comunicazione tra processi in Windows 2000 e UNIX, rispettivamente. Hart [2005] esamina in dettaglio la programmazione nei sistemi Windows.

L'implementazione del meccanismo RPC è analizzata in [Birrell e Nelson 1984]. Il progetto di un meccanismo RPC affidabile è trattato da [Shrivastava e Panzieri 1982], mentre [Tay e Ananda 1990] offrono una panoramica di RPC. [Stankovic 1982] e [Staunstrup 1982] mettono a confronto la comunicazione basata su chiamate di procedura e quella fondata sullo scambio di messaggi. Harold [2005] analizza la programmazione delle socket in Java. Hart[2005] e Robbins e Robbins [2003] trattano il tema delle pipe, sia nei sistemi Windows sia nei sistemi UNIX.

## Capitolo 4

# Thread



### OBIETTIVI

- Introduzione del concetto di thread, l'unità fondamentale nell'utilizzo della CPU alla base dei moderni sistemi multithread.
- Analisi delle API per l'uso dei thread in Java, Win32 e Pthread.

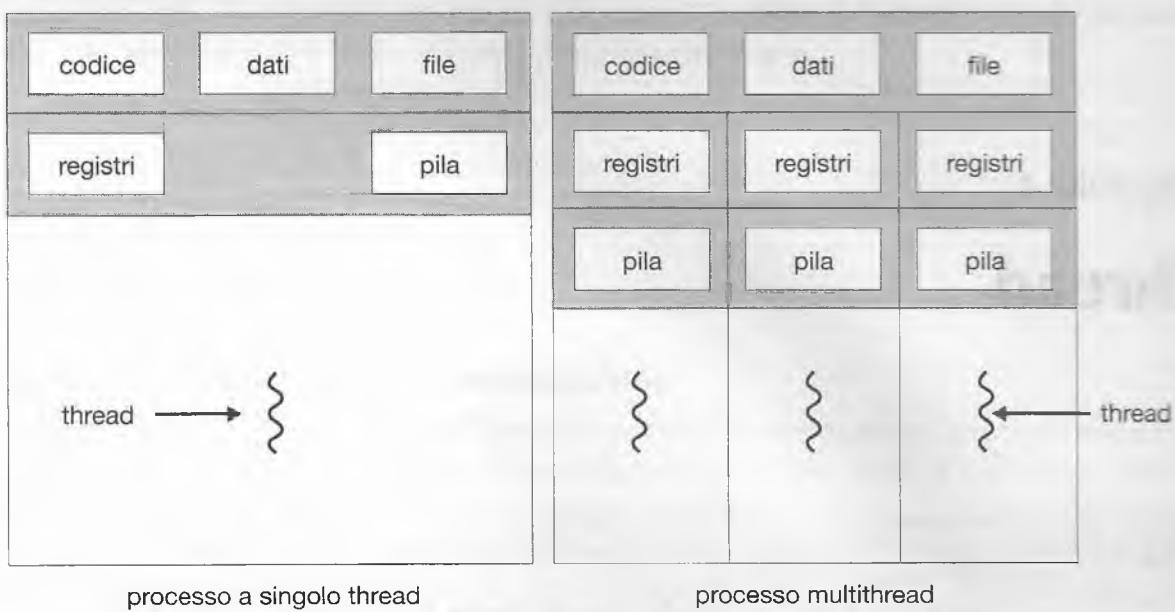
Nel modello introdotto nel Capitolo 3 un processo è considerato come un programma in esecuzione con un unico percorso di controllo. Molti sistemi operativi moderni permettono che un processo possa avere più percorsi di controllo che comunemente si chiamano *thread*. In questo capitolo sono introdotti diversi concetti associati ai sistemi di calcolo *multithread*, tra i quali un'approfondita descrizione dell'API per le librerie di thread di Pthreads, Win32 e Java. Si esaminano molti aspetti legati alla programmazione multithread, e il modo in cui influenza la progettazione dei sistemi operativi; infine, si analizza il modo in cui alcuni sistemi operativi moderni, come Windows XP e Linux, gestiscono i thread a livello kernel.

## 4.1 Introduzione

Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un contatore di programma, un insieme di registri, e una pila (*stack*). Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali. Un processo tradizionale, chiamato anche **processo pesante** (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di lavorare a più compiti in modo concorrente. La Figura 4.1 mostra la differenza tra un processo tradizionale, a **singolo thread**, e uno **multithread**.

### 4.1.1 Motivazioni

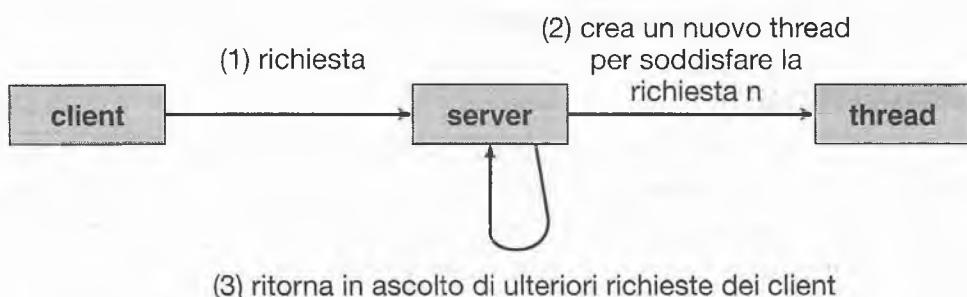
Molti programmi per i moderni PC sono predisposti per essere eseguiti da processi **multithread**. Di solito, un'applicazione si codifica come un processo a sé stante comprendente più thread di controllo: un programma di consultazione del Web potrebbe avere un thread per la rappresentazione sullo schermo di immagini e testo, mentre un altro thread potrebbe occuparsi del reperimento dei dati nella rete; un elaboratore di testi potrebbe avere un thread per la rappresentazione grafica, uno per la lettura dei dati immessi con la tastiera e uno per la correzione ortografica e grammaticale eseguita in sottofondo.



**Figura 4.1** Processi a singolo thread e multithread.

In alcune situazioni, una singola applicazione deve poter gestire molti compiti simili tra loro. Per esempio, un server Web accetta dai client richieste di pagine web, immagini, suoni, e altro. Per un noto server Web potrebbero esservi molti (forse centinaia) client che vi accedono in modo concorrente; se il server Web fosse eseguito come un processo tradizionale a singolo thread, sarebbe in grado di soddisfare un solo client alla volta e un client potrebbe dover aspettare a lungo prima che venga esaudita la sua richiesta.

Una soluzione è eseguire il server come un singolo processo che accetta richieste. Quando ne riceve una, il server crea un processo separato per eseguirla. In effetti, questo metodo di creazione di processi era molto usato prima che si diffondesse la possibilità di gestione dei thread. La creazione dei processi è molto onerosa, sia a livello dei tempi sia dei costi; se il nuovo processo si deve occupare degli stessi compiti del processo corrente, non c'è alcuna ragione di accettare l'intero carico che la sua creazione comporta. Generalmente, per raggiungere lo stesso obiettivo è più conveniente impiegare un processo multithread. Nel caso del server Web, l'adozione di questo metodo porterebbe alla scelta di un processo multithread; il server genererebbe un thread distinto per ricevere eventuali richieste dei client; alla presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla. Il tutto è illustrato nella Figura 4.2.



**Figura 4.2** Architettura di server multithread.

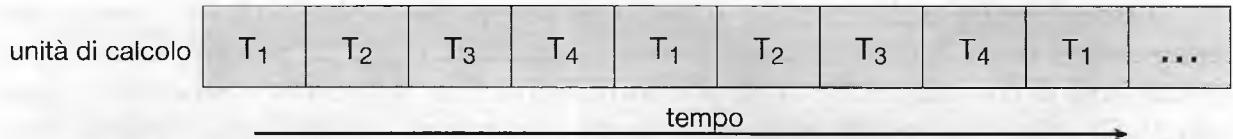
I thread hanno anche un ruolo primario nei sistemi che impiegano le RPC (*remote procedure call*); si tratta di un sistema che permette la comunicazione tra processi, fornendo un meccanismo di comunicazione simile alle normali chiamate di funzione o procedura (Capitolo 3). Di solito, i server RPC sono multithread; quando riceve un messaggio, il server delega la gestione a un thread separato, in questo modo può gestire diverse richieste in modo concorrente.

Infine, molti kernel di sistemi operativi sono ormai multithread, con i singoli thread dedicati a specifici servizi – per esempio, la gestione dei dispositivi periferici o delle interruzioni. È il caso di Solaris, il cui kernel avvia uno specifico insieme di thread specializzati nella gestione delle interruzioni; Linux usa un thread a livello kernel per la gestione della memoria libera del sistema.

### 4.1.2 Vantaggi

I vantaggi della programmazione multithread si possono classificare rispetto a quattro fattori principali.

- 1. Tempo di risposta.** Rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta medio all'utente. Per esempio, un programma di consultazione del Web multithread potrebbe permettere l'interazione con l'utente tramite un thread mentre un'immagine verrebbe caricata da un altro thread.
- 2. Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. Queste tecniche devono essere esplicitamente messe in atto e organizzate dal programmatore. Tuttavia, i thread condividono d'ufficio la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
- 3. Economia.** Assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto. È difficile misurare empiricamente la differenza del carico richiesto per creare e gestire un processo invece che un thread, tuttavia la creazione e gestione dei processi richiede in generale molto più tempo. In Solaris, per esempio, la creazione di un processo richiede un tempo trenta volte maggiore di quello richiesto per la creazione di un thread, un cambio di contesto per un processo richiede un tempo pari a circa cinque volte quello richiesto per un thread.
- 4. Scalabilità.** I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore). Un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo. Esploreremo questo tema nel prossimo paragrafo.



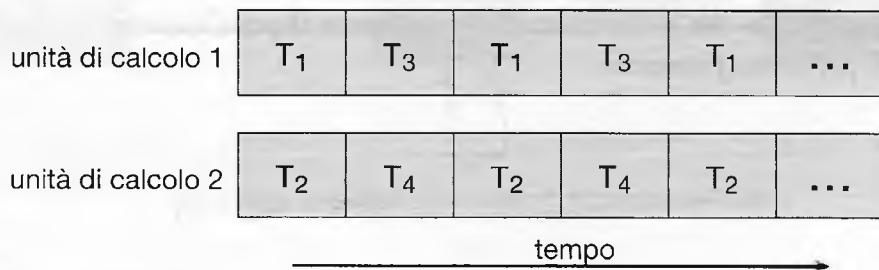
**Figura 4.3** Esecuzione concorrente su un sistema a singola unità di calcolo.

### 4.1.3 Programmazione multicore

Una tendenza recente nel progetto dell'architettura dei sistemi consiste nel montare diverse unità di calcolo (*core*) su un unico processore (un processore *multicore*); ogni unità appare al sistema operativo come un processore separato (Paragrafo 1.3.2). La programmazione multi-thread offre un meccanismo per un utilizzo più efficiente di questi processori e aiuta a sfruttare al meglio la concorrenza. Si consideri un'applicazione con quattro thread. In un sistema con una singola unità di calcolo “esecuzione concorrente” significa solo che l'esecuzione dei thread è stratificata nel tempo, o come anche si dice, *interfogliata (interleaved)* (Figura 4.3), perché la CPU è in grado di eseguire un solo thread alla volta. Su un sistema multicore, invece, “esecuzione concorrente” significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascuna unità di calcolo (Figura 4.4).

La tendenza verso i sistemi multicore ha messo sotto pressione i progettisti di sistemi operativi e i programmatore di applicazioni, affinché entrambi utilizzino al meglio unità di calcolo multiple. I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diverse unità di calcolo per permettere un'esecuzione parallela come quella mostrata nella Figura 4.4. Per i programmatore di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread per trarre vantaggio dai sistemi multicore. In generale, possiamo individuare nelle cinque aree seguenti i principali obiettivi della programmazione dei sistemi multicore.

- Separazione dei task.** Consiste nell'esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possano essere eseguiti in parallelo su unità di calcolo distinte.
- Bilanciamento.** Nell'identificare i task eseguibili in parallelo, i programmatore devono far sì che i vari task eseguano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena utilizzare un'unità di calcolo separata.
- Suddivisione dei dati.** Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte.



**Figura 4.4** Esecuzione parallela su un sistema multicore.

4. Dipendenze dei dati. I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In esempi in cui un task dipende dai dati forniti da un altro, i programmatore devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze.
5. Test e debugging. Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di programmi concorrenti è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

Molti sviluppatori di software sostengono, in ragione degli obiettivi appena esposti, che l'avvento dei sistemi multicore richiederà in futuro un approccio interamente nuovo al progetto dei sistemi software.

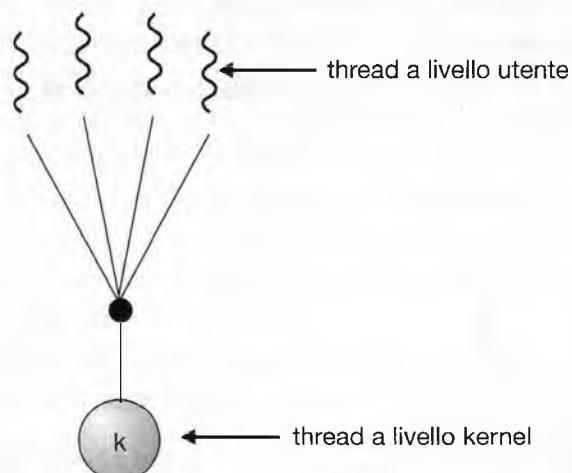
## 4.2 Modelli di programmazione multithread

I thread possono essere distinti in **thread a livello utente** e **thread a livello kernel**: i primi sono gestiti senza l'aiuto del kernel; i secondi, invece, sono gestiti direttamente dal sistema operativo. Praticamente tutti i sistemi operativi moderni dispongono di thread del kernel, compresi Windows XP, Linux, Mac OS X, Solaris, e Tru64 UNIX (precedentemente noto come Digital UNIX).

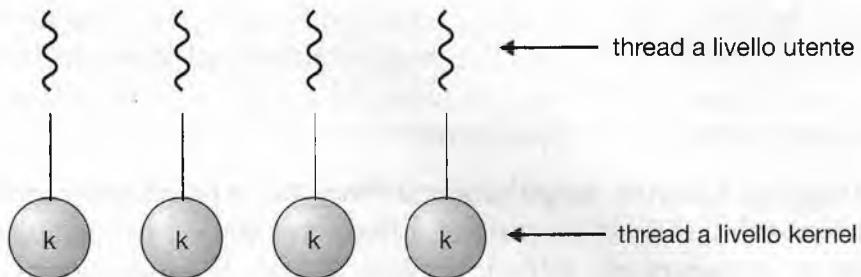
In ultima analisi, deve esistere una relazione tra i thread utente e i thread del kernel: in questo paragrafo ne analizziamo la natura attraverso tre casi comuni.

### 4.2.1 Modello da molti a uno

Il modello da molti a uno (Figura 4.5) fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. Poiché si svolge nello spazio utente, la gestione dei thread risulta efficiente, ma l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poiché un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multiprocessore; la libreria **green threads**, disponibile per Solaris, usa questo modello, come anche **GNU portable thread**.



**Figura 4.5** Modello da molti a uno.



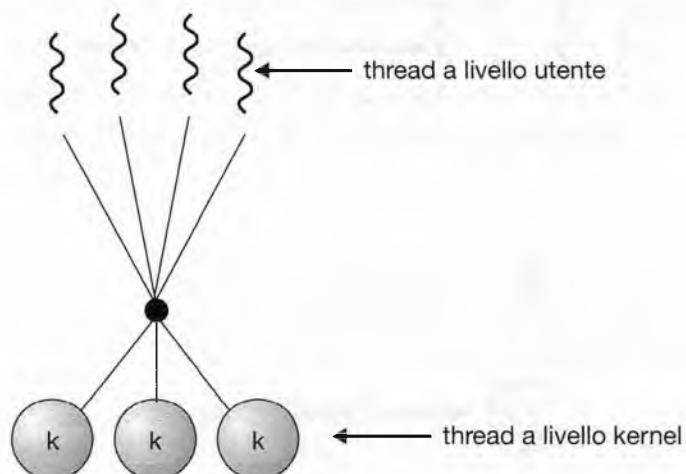
**Figura 4.6** Modello da uno a uno.

### 4.2.2 Modello da uno a uno

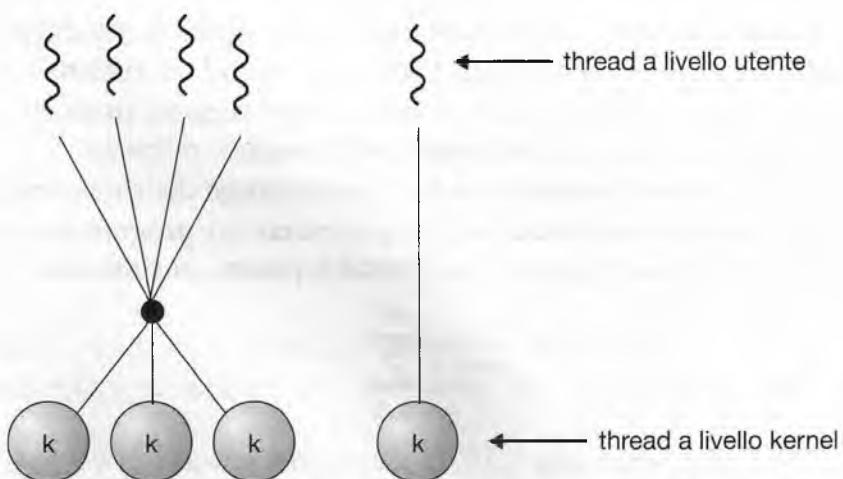
Il modello da uno a uno (Figura 4.6) mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore. L'unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel. Poiché il carico dovuto alla creazione di un thread a livello kernel può compromettere le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread gestibili dal sistema. I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.

### 4.2.3 Modello da molti a molti

Il modello da molti a molti (Figura 4.7) mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel; quest'ultimo può essere specifico per una certa applicazione o per un particolare calcolatore (un'applicazione potrebbe assegnare un numero maggiore di thread a livello kernel a un'architettura multiprocessore rispetto quanti ne assegnerebbe a una con singola CPU). Nonostante il modello da molti a uno permetta ai programmati di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta. Il modello da uno a uno permette una maggiore con-



**Figura 4.7** Modello da molti a molti.



**Figura 4.8** Modello a due livelli.

correnza, ma i programmatori devono stare attenti a non creare troppi thread all'interno di un'applicazione (in qualche caso si possono avere specifiche limitazioni sul numero di thread che si possono creare). Il modello da molti a molti non ha nessuno di questi difetti: i programmatori possono creare liberamente i thread che ritengono necessari, e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata disistema bloccante, il kernel può fare in modo che si esegua un altro thread.

Una diffusa variante del modello da molti a molti mantiene la corrispondenza fra più thread utente con un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread del kernel. La variante, detta a volte *modello a due livelli* (Figura 4.8), è impiegata per esempio da IRIX, HP-UX, e Tru64 UNIX, e dalle versioni di Solaris precedenti a Solaris 9, a partire dalla quale il sistema segue il modello da uno a uno.

## 4.3 Librerie dei thread

La **libreria dei thread** fornisce al programmatore una API per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due. Nel primo, la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio degli utenti. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio degli utenti e non in una chiamata di sistema.

Il secondo metodo consiste nell'implementare una libreria a livello kernel, con l'ausilio diretto del sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della API per la libreria provoca, generalmente, una chiamata di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: (1) Pthreads di POSIX, (2) Win32 e (3) Java. Pthreads, estensione dello standard POSIX, può presentarsi sia come libreria a livello utente sia a livello kernel. La libreria di thread Win32 è una libreria a livello kernel per i sistemi Windows. La API per la creazione dei thread in Java è gestibile direttamente dai programmi Java. Tuttavia, data la peculiarità di funzionamento della JVM, quasi

sempre eseguita all'interno di un sistema operativo che la ospita, la API di Java per i thread è solitamente implementata per mezzo di una libreria dei thread del sistema ospitante. Perciò, i thread di Java sui sistemi Windows sono in effetti implementati mediante la API Win32; sui sistemi UNIX e Linux, invece, si adopera spesso Pthreads.

Nel seguito affrontiamo i fondamenti della generazione dei thread nelle tre librerie a ciò dedicate. Come esempio dimostrativo, progetteremo un programma che computa la somma dei primi  $N$  interi non negativi in un thread separato, in simboli:

$$\text{sum} = \sum_{i=0}^N i$$

Se, per esempio,  $N = 5$ , si avrebbe  $\text{sum} = 15$ , la somma dei numeri da 0 a 5. Ciascuno dei tre programmi funzionerà inserendo nella riga di comando l'indice superiore  $N$  della sommatoria; inserendo 8, quindi, si otterrà come risultato la somma dei valori interi da 0 a 8.

### 4.3.1 Pthreads

Col termine Pthreads ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce la API per la creazione e la sincronizzazione dei thread. Non si tratta di una *realizzazione*, ma di una *definizione* del comportamento dei thread; i progettisti di sistemi operativi possono realizzare le API così definite come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; fra questi, Solaris, Linux, Mac OS X, e Tru64 UNIX. Per i vari sistemi Windows, sono disponibili implementazioni *shareware* di dominio pubblico.

Il programma C multithread nella Figura 4.9 esemplifica la API Pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito. Nei programmi Pthreads, i nuovi thread sono eseguiti a partire da una funzione specificata. Nel programma in esame si tratta della funzione `runner()`. All'inizio dell'esecuzione del programma c'è un unico thread di controllo che parte da `main()`; dopo una fase d'inizializzazione, `main()` crea un secondo thread che inizia l'esecuzione dalla funzione `runner()`. Entrambi i thread condividono i valori globali di `sum`.

Tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione `pthread.h`. La dichiarazione di variabili `pthread_t tid` specifica l'identificatore per il thread da creare. Ogni thread ha un insieme di attributi che includono la dimensione della pila e informazioni di scheduling. La dichiarazione `pthread_attr_t attr` riguarda la struttura dati per gli attributi del thread, i cui valori si assegnano con la chiamata di funzione `pthread_attr_init(&attr)`. Poiché non sono stati esplicitamente forniti valori per gli attributi, si usano quelli predefiniti. (Nel Capitolo 5 saranno esaminati alcuni degli attributi di scheduling offerti dalle API Pthreads.) La chiamata di funzione `pthread_create` crea un nuovo thread. Oltre all'identificatore del thread e ai suoi attributi, si passa anche il nome della funzione da cui il nuovo thread inizierà l'esecuzione, in questo caso la funzione `runner()`, e il numero intero fornito come parametro alla riga di comando e individuato da `argv[1]`.

A questo punto il programma ha due thread: il thread iniziale (o genitore), in `main()`; e il thread che esegue la somma (o figlio), in `runner()`. Dopo aver creato il secondo, il primo thread attende il completamento del secondo chiamando la funzione `pthread_join()`. Il secondo thread termina quando s'invoca la funzione `pthread_exit()`. Quando il thread che esegue la somma termina, il thread iniziale produce in uscita il valore condiviso `sum` della sommatoria.

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

int sum; /* questo dato è condiviso dai thread */
void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* identificatore del thread */
    pthread_attr_t attr; /* insieme di attributi del thread */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* reperisce gli attributi predefiniti */
    pthread_attr_init(&attr);
    /* crea il thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* attende la terminazione del thread */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* Il thread assume il controllo da questa funzione */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figura 4.9 Programma multithread in linguaggio C che impiega la API Pthreads.

```

#include <windows.h>
#include <stdio.h>
DWORD Sum; /* dato condiviso dai thread */
/* il nuovo thread è eseguito in questa funzione apposita */

DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    /* semplice controllo degli errori sui valori in ingresso
 */
    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }

    // crea il nuovo thread
    ThreadHandle = CreateThread(
        NULL, // attributi di default per la sicurezza
        0, // dimensione di default della pila
        Summation, // funzione del thread
        &Param, // parametro della funzione del thread
        0, // flag di creazione di default
        &ThreadId); // restituisce l'identificatore del thread

    if (ThreadHandle != NULL) {
        // attende che il thread figlio termini
        WaitForSingleObject(ThreadHandle,INFINITE);

        // chiude il riferimento al thread figlio
        CloseHandle(ThreadHandle);

        printf("sum = %d\n",Sum);
    }
}

```

Figura 4.10 Programma C multithread che impiega la API Win32.

### 4.3.2 Thread in Win32

La tecnica usata dalla libreria Win32 per la creazione dei thread può richiamare, per molti versi, quella di Pthreads. Illustriamo la API Win32 nel programma C mostrato dalla Figura 4.10. Si noti che per utilizzare la API Win32 è necessario includere il file d'intestazione `windows.h`.

Come nella versione di Pthreads della Figura 4.9, i dati condivisi da thread separati – nella fattispecie, `Sum` – sono globali (il tipo `DWORD` è un intero di 32 bit privo di segno). Dobbiamo inoltre definire la funzione `Summation()` che il nuovo thread eseguirà. A questa funzione è passato un puntatore a `void`, che in Win32 è `LPVOID`. Il thread che esegue questa funzione imposta la variabile globale `Sum` al valore della sommatoria da 0 fino al parametro passato a `Summation()`.

Nella API Win32 i nuovi thread si generano tramite la funzione `CreateThread()`, che – proprio come in Pthreads – accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione della pila e un indicatore (flag) per segnalare se il thread debba avere inizio nello stato d'attesa. Ci serviremo, nel programma, dei valori di default di questi attributi, che inizialmente non pongono il thread in stato d'attesa, bensì lo rendono eseguibile dallo scheduler della CPU. Una volta creato il nuovo thread, il thread iniziale deve attenderne il completamento prima di produrre in uscita il valore di `Sum`, poiché esso è computato dal nuovo thread. Come si ricorderà, nel programma Pthread (Figura 4.9) il thread iniziale era posto in attesa della terminazione del nuovo thread tramite la funzione `pthread_join()`. La chiamata equivalente nella API Win32 è `WaitForSingleObject()`, che ottiene la sospensione del thread iniziale fintanto che il nuovo thread non abbia terminato (Figura 4.10). (Indagheremo più a fondo sulla sincronizzazione nel Capitolo 6.)

### 4.3.3 Thread Java

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria API, è provvisto di una ricca gamma di caratteristiche per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo – persino un semplice programma, costituito soltanto da un metodo `main()`, è eseguito dalla JVM come un singolo thread.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e “sovrascrivere” (*override*) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`, corrispondente a:

```
public interface Runnable
{
    public abstract void run();
}
```

Per implementare `Runnable`, una classe è tenuta a definire il metodo `run()`. Il codice che implementa `run()` sarà eseguito in un thread distinto.

La Figura 4.11 mostra la versione Java di un programma multithread che calcola la somma degli interi da 0 a  $N$ . La classe `Summation` implementa l'interfaccia `Runnable`. La generazione del thread prevede che si crei un'istanza della classe `Thread` passando al costruttore un oggetto `Runnable`.

La creazione di un oggetto di classe `Thread` non equivale a generare un nuovo thread: è il metodo `start()` che avvia effettivamente il nuovo thread. L'invocazione del metodo `start()` ha il duplice effetto di:

1. allocare la memoria e inizializzare un nuovo thread nella JVM;
2. chiamare il metodo `run()`, cosa che rende il thread eseguibile dalla JVM. Si osservi come il metodo `run()` non sia mai chiamato per via diretta, ma solo tramite la mediazione di `start()`.

```

class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}

public class Driver
{
    public static void main(String[] args) {

```

**Figura 4.11** Programma Java per il calcolo di una sommatoria. (continua)



```

if (args.length > 0) {
    if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
    else {
        // crea l'oggetto condiviso dai thread
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper,
sumObject));
        thrd.start();
        try {
            thrd.join();
            System.out.println
                ("The sum of "+upper+" is
"+sumObject.getSum());
        } catch (InterruptedException ie) { }
    }
}
else
    System.err.println("Usage: Summation <integer value>"); }
}

```

**Figura 4.11 (continua)** Programma Java per il calcolo di una sommatoria.

All'avvio del programma che calcola la somma, la JVM crea due thread. Il primo è il thread genitore, che inizia a essere eseguito dal metodo `main()`. Il secondo thread, figlio del primo, ha origine quando il metodo `start()` è invocato sull'oggetto di classe `Thread`. L'esecuzione di questo thread figlio inizia dal metodo `run()` della classe `Summation`. Dopo aver restituito il valore della somma, il thread termina all'uscita dal proprio metodo `run()`.

La condivisione dei dati tra thread diversi è comune in Win32 e in Pthreads: i dati condivisi sono semplicemente dichiarati globali. In quanto linguaggio orientato agli oggetti puro, Java non contempla la nozione di variabile globale: la condivisione dei dati fra più thread in Java avviene tramite il passaggio di riferimenti a uno stesso oggetto. Nel programma in Java della Figura 4.11 il thread principale e quello che calcola la somma condividono un oggetto di classe `Sum`. A tale oggetto condiviso si accede attraverso gli appositi metodi `getSum()` e `setSum()`. Ci si potrebbe chiedere perché non si usi un oggetto di classe `Integer` anziché predisporre una nuova classe `Sum`. La ragione è che la classe `Integer` è immutabile – ovvero, una volta impostato il valore di una sua istanza, non può più cambiare.

Si tenga presente che i thread genitori nelle librerie Pthreads e Win32 usano, rispettivamente, `phtread_join()` e `WaitForSingleObject()` per attendere la conclusione del thread che esegue la somma prima di procedere. Il metodo `join()` in Java fornisce una simile funzionalità. (Si noti che `join()` può sollevare una `InterruptedException`, che nel codice abbiamo deciso di non gestire.)

## LA JVM E IL SISTEMA OPERATIVO RESIDENTE

La macchina virtuale del linguaggio Java (JVM) è solitamente implementata sulla base di un sistema operativo sottostante. Questo assetto permette alla JVM di nascondere i dettagli del sistema operativo e di offrire un ambiente astratto e coerente che consente ai programmi scritti in Java di essere eseguiti su qualsiasi piattaforma che disponga di una JVM. Le specifiche della JVM non prescrivono come i thread Java debbano corrispondere ai servizi del sistema operativo sottostante, lasciando i dettagli all'implementazione. Il sistema operativo Windows XP, per esempio, adotta il modello da uno a uno; perciò, ogni thread Java di una JVM installata su questo sistema corrisponde a un thread a livello kernel. Sui sistemi che adottano il modello da molti a molti, come il Tru64 UNIX, i thread di Java sono associati a thread sottostanti secondo il modello da molti a molti. La JVM per Solaris impiegava inizialmente il modello da molti a uno (la libreria *green threads*, citata in precedenza); versioni successive della JVM per Solaris hanno eletto il modello da molti a molti. A partire da Solaris 9, i thread di Java rimangono associati ai thread sottostanti secondo il modello da uno a uno. Oltre a queste considerazioni, può esistere una relazione tra la libreria dei thread di Java e la libreria dei thread del sistema operativo residente. Le varie versioni della JVM per la famiglia di sistemi operativi Windows, per esempio, potrebbero ricorrere alla API Win32 al fine di implementare thread di Java; i sistemi Linux e Solaris potrebbero impiegare la API Pthreads.

## 4.4 Questioni di programmazione multithread

In questo paragrafo si affrontano alcune problematiche legate ai programmi multithread.

### 4.4.1 Chiamate di sistema `fork()` ed `exec()`

Nel Capitolo 3 è descritto l'uso della chiamata di sistema `fork()` per la creazione di un nuovo processo tramite la duplicazione di un processo esistente. In un programma multithread la semantica delle chiamate di sistema `fork()` ed `exec()` cambia: se un thread in un programma invoca la chiamata di sistema `fork()`, il nuovo processo potrebbe, in generale, contenere un duplicato di tutti i thread oppure del solo thread invocante.

Alcuni sistemi UNIX includono entrambe le versioni. La chiamata di sistema `exec()` di solito funziona nello stesso modo descritto nel Capitolo 3: se un thread invoca la chiamata di sistema `exec()`, il programma specificato come parametro della `exec()` sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della `fork()` dipende dall'applicazione. Se s'invoca la `exec()` immediatamente dopo la `fork()`, la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della `exec()` sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la `exec()` non segue immediatamente la `fork()`, potrebbe essere utile una duplicazione di tutti i thread del processo genitore.

### 4.4.2 Cancellazione

La cancellazione dei thread è l'operazione che permette di terminare un thread prima che completi il suo compito. Per esempio, se più thread eseguono una ricerca in modo concorrente in una base di dati e un thread riporta il risultato, gli altri thread possono essere annullati. Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di

terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina. Spesso il caricamento di una pagina è gestito da un thread distinto; quando l'utente preme il pulsante di terminazione, il thread che sta caricando la pagina viene cancellato.

Un thread da cancellare è spesso chiamato **thread bersaglio** (*target thread*). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

1. **cancellazione asincrona**. Un thread fa immediatamente terminare il thread bersaglio;
2. **cancellazione differita**. Il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscirvi in modo opportuno.

Si presentano difficoltà con la cancellazione nei casi in cui ci siano risorse assegnate a un thread cancellato, o se si cancella un thread mentre sta aggiornando dei dati che condivide con altri thread. Quest'ultimo caso è particolarmente problematico se si tratta di cancellazione asincrona. Il sistema operativo di solito si riappropria delle risorse di sistema usate da un thread cancellato, ma spesso non si riappropria di tutte le risorse. Quindi, la cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La cancellazione differita invece funziona tramite un thread che segnala la necessità di cancellare un certo thread bersaglio; la cancellazione avviene soltanto quando il thread bersaglio verifica se debba essere o meno cancellato. Questo metodo permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi. Nella libreria Pthreads questi punti si chiamano **punti di cancellazione** (*cancellation point*).

### 4.4.3 Gestione dei segnali

Nei sistemi UNIX si usano i **segnali** per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

1. all'occorrenza di un particolare evento si genera un segnale;
2. s'invia il segnale a un processo;
3. una volta ricevuto, il segnale deve essere gestito.

Un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui si chiamano sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti (come `<control><C>`) oppure la scadenza di un timer. Di solito un segnale asincrono s'invia a un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un gestore predefinito di segnali;
2. tramite un gestore di segnali definito dall'utente.

Per ogni segnale esiste un **gestore predefinito del segnale** che il kernel esegue quando deve gestire il segnale. La gestione predefinita è sostituibile da una funzione di **gestione del segnale definita dall'utente**, richiamata per gestire il segnale. Sia i segnali sincroni sia quelli

asincroni sono gestibili in modi diversi: alcuni si possono semplicemente ignorare (per esempio, il ridimensionamento di una finestra); altri si possono gestire terminando l'esecuzione del programma (per esempio, un accesso illegale alla memoria).

Poiché nell'inviare un segnale è sufficiente fare riferimento al processo interessato, per i processi a singolo thread la gestione dei segnali è semplice. Per i processi multithread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce;
2. inviare il segnale a ogni thread del processo;
3. inviare il segnale a specifici thread del processo;
4. definire un thread specifico per ricevere tutti i segnali diretti al processo.

Il metodo per recapitare un segnale dipende dal tipo di segnale. I segnali sincroni, per esempio, si devono inviare al thread che ha generato l'evento causa del segnale e non ad altri thread nel processo. Se si tratta di segnali asincroni la situazione non è invece così chiara; alcuni segnali asincroni, come il segnale che termina un processo (come <control><C>), si devono inviare a tutti i thread. La maggior parte delle versioni multithread del sistema operativo UNIX permettono che per ciascun thread si indichino i segnali da accettare e quelli da bloccare. Quindi, alcuni segnali asincroni si potrebbero recapitare soltanto ai thread che non li bloccano.

Tuttavia, poiché i segnali vanno gestiti una sola volta, di solito un segnale è recapitato solo al primo thread che non lo blocca. La funzione UNIX per recapitare i segnali è `kill(aid_t aid, int signal)`, dove `aid` specifica il processo a cui recapitare il segnale `signal`. La API Pthreads POSIX, però, dispone anche della funzione `pthread_kill(pthread_t tid, int signal)` che permette di specificare il thread (`tid`) cui recapitare il segnale.

Sebbene Windows non preveda la gestione esplicita dei segnali, questi si possono emulare con le **chiamate di procedure asincrone** (*asynchronous procedure call*, APC). Le funzioni APC permettono a un thread a livello utente di specificare la funzione da richiamare quando il thread riceve la comunicazione di un particolare evento. Come s'intuisce dal nome, una APC è grosso modo equivalente a un segnale asincrono di UNIX. Mentre tuttavia in un ambiente multithread UNIX necessita di un criterio di gestione dei segnali, il sistema delle APC è più semplice, poiché una APC è rivolta a un particolare thread e non a un processo.

#### 4.4.4 Gruppi di thread

Nel Paragrafo 4.1 è descritto lo scenario di un server Web multithread in cui, per ogni richiesta ricevuta, il server crea un thread distinto per fornire il servizio richiesto. Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, un server multithread presenta diversi problemi. Il primo riguarda il tempo richiesto per la creazione del thread prima di poter soddisfare la richiesta, considerando anche il fatto che questo thread sarà terminato non appena avrà completato il proprio lavoro. La seconda questione è più problematica: se si permette che tutte le richieste concorrenti siano servite da un nuovo thread, non si è posto un limite al numero di thread attivi in modo concorrente nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di CPU o la memoria. L'impiego dei **gruppi di thread** (*thread pool*) è una possibile soluzione a questo problema.

L'idea generale è quella di creare un certo numero di thread alla creazione del processo, e organizzarli in un gruppo (*pool*) in cui attendano il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread del gruppo – se ce n'è uno disponibile – e

gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nel gruppo d'attesa. Se il gruppo non contiene alcun thread disponibile, il server attende fino al rientro di un thread. I vantaggi offerti sono i seguenti:

1. di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread;
2. un gruppo di thread limita il numero di thread esistenti a un certo istante; ciò è particolarmente rilevante per sistemi che non possono sostenere un elevato numero di thread concorrenti.

Il numero di thread di un gruppo si può determinare tramite euristiche che considerano fattori come il numero di CPU nel sistema, la quantità di memoria fisica e il numero atteso di richieste concorrenti da parte dei client. Architetture più raffinate per la gestione dei gruppi di thread possono correggere dinamicamente il numero di thread di un gruppo secondo schemi d'uso. Queste architetture hanno l'ulteriore vantaggio di presentare gruppi più piccoli – comportando quindi un minore impegno di memoria – quando il carico del sistema è basso.

La API Win32 mette a disposizione diverse funzioni legate ai gruppi di thread, il cui uso è simile alla creazione di un thread tramite la funzione `Thread Create()` presentata al Paragrafo 4.3.2. Si definisce una funzione da eseguire in un nuovo thread, come per esempio:

```
DWORD WINAPI PoolFunction(VOID Param) {
    /**
     * Questa funzione gira come nuovo thread.
     */
}
```

Un puntatore a `PoolFunction()` è poi passato a una delle apposite funzioni nella API per i gruppi di thread, il che avvierà uno dei thread del gruppo. Una di tali apposite funzioni è `QueueUserWorkItem()`, che accetta tre parametri:

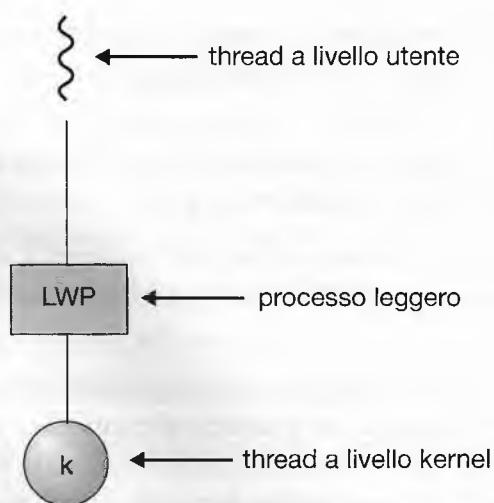
- ◆ `LPTHREAD_START_ROUTINE Function` – un puntatore alla funzione da eseguire in un nuovo thread;
- ◆ `PVOID Param` – il parametro passato a `Function`;
- ◆ `ULONG Flags` – indica come il gruppo di thread debba creare e gestire l'esecuzione del nuovo thread.

Un esempio di chiamata è:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

A seguito di questa invocazione, uno dei thread del gruppo richiamerà `PoolFunction()` per conto del programmatore. In questo esempio, `PoolFunction()` non riceve parametri, e il valore 0 di `Flags` indica l'assenza di indicazioni particolari per la creazione del thread.

Altre funzioni della API Win32 per i gruppi di thread offrono servizi di invocazione periodica di funzioni, o invocazioni guidate dal completamento di un'operazione di I/O asincrona. Anche il package `java.util.concurrent` di Java 1.5 offre funzionalità per la gestione di gruppi di thread.



**Figura 4.12** Processo leggero LWP.

#### 4.4.5 Dati specifici dei thread

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati **dati specifici di thread**. Per esempio, in un sistema per transazioni si può svolgere ciascuna transazione tramite un thread distinto e un identificatore unico per ogni transazione. Per associare ciascun thread al relativo identificatore si possono usare dati specifici dei thread. La maggior parte delle librerie di thread – incluse la Win32 e la Pthreads – e l’ambiente del linguaggio Java consentono l’impiego dei dati specifici di thread.

#### 4.4.6 Attivazione dello scheduler

Un’ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel modello a due livelli e in quello da molti a molti (Paragrafo 4.2.3). È proprio grazie a questa forma di coordinamento che il numero dei thread nel kernel è modificabile dinamicamente, con l’obiettivo di conseguire le migliori prestazioni.

Molti sistemi che implementano o il modello da molti a molti o quello a due livelli collocano una struttura dati intermedia tra i thread del kernel e dell’utente. Questa struttura dati, nota come **processo leggero** o LWP (acronimo di *Lightweight Process*) è mostrata nella Figura 4.12. Dal punto di vista della libreria di thread a livello utente, LWP si presenta come un *processore virtuale* a cui l’applicazione può richiedere lo scheduling di un thread a livello dell’utente. Ciascun LWP è associato a un thread del kernel, e sono proprio i thread del kernel che il sistema operativo pone in esecuzione sui processori fisici. Se un thread del kernel si arresta (mentre attende il completamento di un’operazione di I/O, per esempio) anche LWP si blocca. L’effetto a catena risale fino al thread a livello utente associato a LWP, che si blocca anch’esso.

Per un’efficiente esecuzione un’applicazione può aver bisogno di un numero imprecisato di LPW. Si consideri un processo con prevalenza di elaborazione eseguito da un singolo processore. In questa situazione è eseguibile solo un thread per volta, dunque un LWP è sufficiente. Un’applicazione con prevalenza di I/O potrebbe, tuttavia, richiedere l’esecuzione di

molteplici LWP. Di solito, è necessario un LWP per ogni chiamata di sistema concorrente bloccante. Supponiamo, per esempio, che giungano allo stesso tempo cinque richieste differenti per la lettura di file. Sono necessari cinque LWP, nel caso che tutte le richieste risiedano nel kernel in attesa di essere completate. Se un processo ha soltanto quattro LWP, la quinta richiesta deve attendere che uno degli LWP sia rilasciato dal kernel.

Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come **attivazione dello scheduler**. Il suo funzionamento è il seguente: il kernel fornisce all'applicazione una serie di processori virtuali (LWP), mentre l'applicazione esegue lo scheduling dei thread dell'utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l'applicazione se si verificano determinati eventi, seguendo una procedura nota come **upcall**. Le upcall sono gestite dalla libreria dei thread mediante un apposito gestore, eseguito su un processore virtuale. Una situazione capace di innescare una upcall si verifica quando il thread di un'applicazione è sul punto di bloccarsi. In questo caso il kernel, tramite una upcall, informa l'applicazione che un thread è prossimo a bloccarsi, e identifica il thread in oggetto. Il kernel, quindi, assegna all'applicazione un nuovo processore virtuale. L'applicazione esegue un gestore della upcall su questo nuovo processore: il gestore salva lo stato del thread bloccante e rilascia il processore virtuale su cui era stato eseguito. Il gestore della upcall pianifica allora l'esecuzione di un altro thread sul processore virtuale che si è appena liberato. Quando si verifica l'evento atteso dal thread bloccante, il kernel fa un'altra upcall alla libreria dei thread per comunicare che il thread bloccato è nuovamente in condizione di essere eseguito. Il gestore di questa upcall necessita anch'esso di un processore virtuale: il kernel può creare uno *ex novo*, o sottrarlo a un thread utente per prelazione. L'applicazione contrassegna il thread fino ad allora bloccato come pronto per l'esecuzione, ed esegue lo scheduling di un thread pronto per l'esecuzione su un processore virtuale disponibile.

## 4.5 Esempi di sistemi operativi

In questo paragrafo si descrive il modo in cui i thread sono implementati nei sistemi Windows XP e Linux.

### 4.5.1 Thread nel sistema Windows XP

Il sistema operativo Windows XP offre la API Win32; si tratta dell'API principale della famiglia dei sistemi operativi di Microsoft (Windows 95, 98, NT, 2000 e XP). La maggior parte del contenuto di questo paragrafo si applica all'intera famiglia Microsoft.

Un'applicazione per l'ambiente Windows XP si esegue come un processo separato; ogni processo può contenere uno o più thread. La API Win32 per la creazione dei thread è descritta nel Paragrafo 4.3.2. Il sistema Windows XP impiega il modello da uno a uno, descritto nel Paragrafo 4.2.2, secondo cui ogni thread a livello utente si associa a un thread del kernel. Tuttavia è disponibile anche la libreria fiber, che implementa il modello da molti a molti (Paragrafo 4.2.3). Ogni thread che appartiene a un processo può accedere allo spazio d'indirizzi di quel processo.

I componenti generali di un thread includono:

- un identificatore di thread (ID), che identifica univocamente il thread;
- un insieme di registri che rappresentano lo stato del processore;

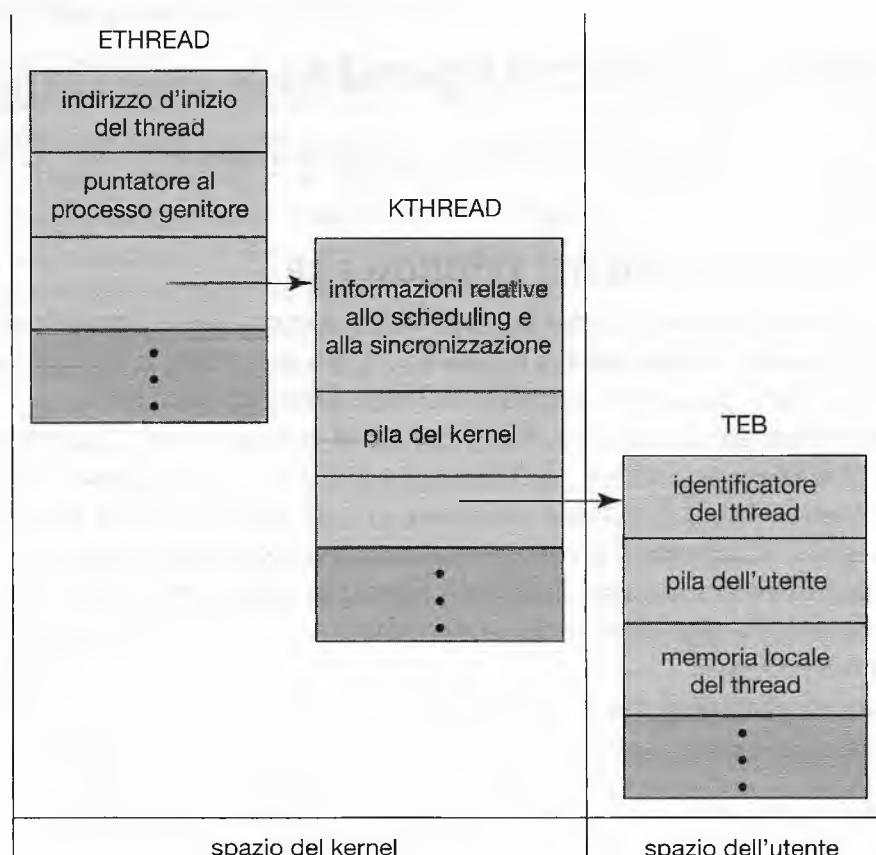
- una pila (stack) utente, usata quando il thread è eseguito in modalità utente, e una pila del kernel, usata quando il thread è eseguito in modalità kernel;
  - un'area di memoria privata, usata da diverse librerie di fase d'esecuzione e dinamiche (DLL).

L'insieme di registri, le pile e la memoria privata è detto **contesto** del thread. Le strutture dati principali di un thread includono:

- ◆ ETHREAD (*executive thread block*);
  - ◆ KTHREAD (*kernel thread block*);
  - ◆ TEB (*thread environment block*).

I componenti chiave dell'ETHREAD sono un puntatore al processo a cui il thread appartiene e l'indirizzo della funzione in cui il thread assume il controllo. La struttura ETHREAD contiene anche un puntatore alla corrispondente struttura KTHREAD. Quest'ultima include informazioni per il thread relative allo scheduling e alla sincronizzazione. Inoltre, KTHREAD contiene la pila del kernel (usata quando il thread viene eseguito in modalità kernel) e un puntatore alla struttura TEB.

Le strutture ETHREAD e KTHREAD risiedono interamente nello spazio del kernel; ciò implica che solo il kernel vi può accedere. La struttura dati TEB appartiene invece allo spazio utente e vi si accede quando il thread è eseguito in modalità utente. Tra gli altri campi, il TEB contiene una pila per la modalità utente e un vettore per dati specifici del thread che Windows XP chiama **memoria locale del thread** (*thread-local storage*). La struttura di un thread di Windows XP è illustrata nella Figura 4.13.



**Figura 4.13** Strutture dati di un thread Windows XP.

## 4.5.2 Thread di Linux

Come si è visto nel Capitolo 3 Linux offre la chiamata di sistema `fork()` per duplicare un processo, e prevede inoltre la chiamata di sistema `clone()` per generare nuovo thread. Tuttavia Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine **task** (*operazione*) in riferimento al flusso del controllo di un programma. Quando `clone()` è invocata, riceve come parametro un insieme di indicatori (*flag*), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio. Alcuni di questi flag sono illustrati nello schema seguente.

flag	significato
<code>CLONE_FS</code>	Condivisione delle informazioni sul file system
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dei file aperti

Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il task genitore e il task figlio condivideranno le medesime informazioni sul file system (come la directory attiva), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Adoperare `clone()` in questo modo è equivalente a creare thread, dal momento che il task genitore condivide la maggior parte delle proprie risorse con il task figlio. Tuttavia, se nessuno dei flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione, e la funzionalità ottenuta diventa simile a quella fornita dalla chiamata di sistema `fork()`.

Questa condivisione a intensità variabile è resa possibile dal modo in cui un task è rappresentato nel kernel di Linux. Per ogni task, nel kernel esiste un'unica struttura dati (e precisamente, `struct task_struct`). Questa struttura, invece di memorizzare i dati del task relativo, utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti: per esempio, strutture dati che rappresentano l'elenco dei file aperti, le informazioni per la gestione dei segnali e la memoria virtuale. Quando si invoca `fork()`, si crea un nuovo task insieme con una *copia* di tutte le strutture dati del task genitore. Anche quando s'invoca la chiamata `clone()` si crea un nuovo task, ma anziché ricevere una copia di tutte le strutture dati, il nuovo task *punta* a queste o a quelle strutture dati del task genitore, a seconda dell'insieme di flag passati a `clone()`.

## 4.6 Sommario

Un thread è un percorso di controllo d'esecuzione all'interno di un processo. Un processo multithread contiene più percorsi di controllo diversi, ma che condividono lo stesso spazio d'indirizzi. I vantaggi della programmazione multithread includono un miglioramento del tempo di risposta, la condivisione di risorse all'interno del processo, il risparmio e la capacità di sfruttare le architetture dotate di più unità d'elaborazione.

I thread a livello utente sono thread visibili al programmatore e sconosciuti al kernel. Il kernel del sistema operativo gestisce thread a livello kernel. In generale, i thread a livello

utente richiedono minor tempo per essere creati e gestiti rispetto a quelli a livello kernel, senza necessità d'intervento da parte del kernel. Ci sono tre tipi diversi di modelli che descrivono le relazioni fra thread a livello utente e a livello kernel: il modello da molti a uno associa più thread a livello utente a un singolo thread a livello kernel; il modello da uno a uno associa ciascun thread a livello utente a un corrispondente thread a livello kernel; il modello da molti a molti associa dinamicamente più thread a livello utente a un numero minore o uguale di thread a livello kernel.

Molti sistemi operativi moderni prevedono la gestione dei thread a livello kernel: tra questi i sistemi Windows 98, NT, 2000 e XP, oltre a Solaris e Linux.

Per la creazione e la gestione dei thread le relative librerie forniscono una API al programmatore di applicazioni. Le tre più comuni librerie di thread sono: POSIX Pthread, Win32 per i sistemi Windows e i thread Java.

I programmi multithread presentano molti aspetti critici per il programmatore, tra cui la semantica delle chiamate di sistema `fork()` ed `exec()`; altri aspetti sono per esempio la cancellazione, la gestione dei segnali e i dati privati dei thread.

## Esercizi pratici

- 4.1 Fornite due esempi di programmi nei quali il multithread offre prestazioni migliori rispetto a soluzioni con un singolo thread.
- 4.2 Quali sono due differenze tra i thread a livello utente e i thread a livello kernel? In quali circostanze un tipo è meglio dell'altro?
- 4.3 Descrivete le azioni intraprese da un kernel per cambiare contesto tra i thread a livello kernel.
- 4.4 Quali risorse vengono utilizzate quando si crea un thread? Come differiscono da quelle utilizzate quando si crea un processo?
- 4.5 Assumete che un sistema operativo mappi i thread a livello utente sul kernel utilizzando il modello molti a molti e che la mappatura avvenga tramite LWP. Assumete inoltre che il sistema permetta agli sviluppatori di creare dei thread real-time da utilizzare in sistemi real-time. È necessario vincolare un thread real-time a un LWP? Fornite una spiegazione.
- 4.6 Nel Paragrafo 4.3.1 si è descritto un programma Pthread che esegue una somma. Riscrivete il programma in Java.

## Esercizi

- 4.7 Descrivete due esempi di programmazione multithread che offrono prestazioni migliori rispetto alla corrispondente soluzione a singolo thread.
- 4.8 Descrivete le azioni intraprese da una libreria di thread per il cambio di contesto tra thread a livello utente.
- 4.9 In quali circostanze una soluzione basata sulla programmazione multithread che sfrutta thread multipli del kernel offre prestazioni migliori di una soluzione a singolo thread implementata su un sistema monoprocesso?

- 4.10 Quali tra i seguenti componenti dello stato di un programma sono condivisi tra thread in un processo multithread?
- Valori dei registri.
  - Memoria heap.
  - Variabili globali.
  - Pila.
- 4.11 È possibile che una soluzione multithread, impiegando thread multipli a livello utente, consegua prestazioni migliori su un sistema multiprocessore piuttosto che su un sistema a singolo processore?
- 4.12 Come descritto nel Paragrafo 4.5.2 Linux non fa distinzione tra processi e thread, cosicché un task apparirà più affine a un processo, oppure a un thread, a seconda dell'insieme di flag passati alla chiamata di sistema `clone()`. Tuttavia, Windows XP e Solaris, come molti altri sistemi operativi, trattano processi e thread in maniera diversa. In genere, per descrivere un processo, questi sistemi usano strutture dati contenenti un puntatore per ciascun thread appartenente al processo. Ponete a confronto queste due tecniche per rappresentare i processi e i thread nel kernel.
- 4.13 Il programma contenuto nella Figura 4.14 utilizza la API Pthreads. Quali dati in uscita verrebbero prodotti dal programma alla RIGA C e alla RIGA P?
- 4.14 Considerate un sistema multiprocessore e un programma multithread scritto con il modello da molti a molti. Ipotizziamo un numero più alto di thread a livello utente nel programma rispetto al numero di processori nel sistema. Analizzate che cosa implica, in termini di efficienza, ciascuna delle seguenti possibilità.
- Il numero di thread del kernel assegnati al programma è minore del numero di processori.
  - I thread del kernel assegnati al programma sono in numero uguale al numero dei processori.
  - Il numero di thread del kernel assegnati al programma è maggiore del numero di processori, ma minore del numero di thread a livello utente.
- 4.15 Scrivete un programma multithread che impieghi la libreria Pthreads, WIN32 o Java per la generazione di numeri primi. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce un numero alla riga di comando; il programma crea un thread distinto che riporta tutti i numeri primi minori o uguali al numero inserito dall'utente.
- 4.16 Modificate il server basato sulle socket della Figura 3.19 nel Capitolo 3 in modo che esso dedichi un thread separato a ciascuna richiesta del client.
- 4.17 La successione di Fibonacci inizia con 0, 1, 1, 2, 3, 5, 8, ....  
Essa è definita da:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

```

#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* il thread */

int main(int argc, char *argv[])
{
    int pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();

    if (pid == 0) /* processo figlio */
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,runner,NULL);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d",value); /* RIGA C */
    }
    else if (pid > 0) /* processo genitore */
        wait(NULL);
        printf("PARENT: value = %d",value); /* RIGA P */
    }
}

void *runner(void *param) {
    value = 5;
    pthread_exit(0);
}

```

**Figura 4.14** Programma C dell'Esercizio 4.13.

Scrivete un programma multithread usando la libreria Java, oppure quella di Pthreads o di Win32, che generi la successione di Fibonacci. Il programma deve avere il seguente comportamento: l'utente avvia l'esecuzione del programma e inserisce alla riga di comando il numero di termini della successione di Fibonacci che il programma deve generare. Il programma crea un thread separato per la generazione dei numeri di Fibonacci, ma colloca i termini della successione in dati condivisi dai thread (un vettore è probabilmente la struttura dati più adatta). Quando il thread figlio conclude l'esecuzione, il thread genitore emette la sequenza generata dal figlio. Poiché il thread genitore non può produrre in uscita la sequenza prima che il thread figlio abbia terminato, sarà necessario applicare la tecnica illustrata nel Paragrafo 4.3, per sincronizzare il thread genitore con il thread figlio.

- 4.18** Nell'Esercizio 3.18 del Capitolo 3 si spiega come progettare un server eco tramite la API di Java per i thread. Tale server, tuttavia, è a singolo thread, ossia non è in grado di rispondere a richieste simultanee dei client finché il client attualmente servito non termini. Si modifichi la soluzione dell'Esercizio 3.18 affinché il server eco possa servire separatamente ciascun client.

## Progetti di programmazione

I progetti che seguono affrontano due distinti argomenti: il servizio di risoluzione dei nomi e la moltiplicazione fra matrici.

### Progetto 1: Progetto sul servizio di risoluzione dei nomi

Un servizio di risoluzione dei nomi come DNS (*domain name system*, sistema dei nomi di dominio) può essere utilizzato per tradurre nomi IP in indirizzi IP. Ad esempio, nell'accedere all'host `www.westminstercollege.edu` si utilizza un servizio di risoluzione dei nomi per determinare l'indirizzo IP associato al nome `www.westminstercollege.edu`. Questo progetto consiste nello scrivere un servizio di risoluzione dei nomi multithread in Java utilizzando delle socket (Paragrafo 3.6.1).

Per risolvere i nomi IP la API `java.net` fornisce il seguente meccanismo:

```
InetAddress hostAddress =
    InetAddress.getByName("www.westminstercollege.edu");
String IPaddress = hostAddress.getHostAddress();
```

dove `getByName()` solleva una `UnknownHostException` se non è in grado di risolvere il nome host.

### Il server

Il server ascolterà sulla porta 6052, in attesa della connessione dei client. Una volta instaurata una connessione con un client, il server erogherà il servizio di risoluzione dei nomi tramite un thread separato e ritornerà in ascolto sulla porta 6052 per ricevere ulteriori richieste di connessione dei client. Dopo la connessione al server, il client scriverà sulla socket il nome IP che vuole far risolvere dal server (ad esempio, `www.westminstercollege.edu`). Il thread del server leggerà questo nome IP dalla socket e tradurrà il suo indirizzo IP oppure, se non potrà localizzare l'indirizzo host, solleverà una `UnknownHostException`. Il server restituirà l'indirizzo IP al client o, in caso di `UnknownHostException`, scriverà il messaggio “`Unable to resolve host <host name>`.” Ciò fatto, il server chiuderà la socket.

### Il client

Inizialmente, scrivete solo l'applicazione server e connettetevi tramite telnet. Ad esempio, supponendo che il server stia funzionando su localhost, una sessione telnet apparirebbe come segue. (Riportiamo in grassetto quanto digitato dall'utente).

```
telnet localhost 6052
Connected to localhost.
Escape character is '^]'.
```

```
www.westminstercollege.edu
```

```
146.86.1.17
```

```
Connection closed by foreign host.
```

Usando inizialmente telnet in funzione di client, potrete eseguire più facilmente il debugging di problemi che possono insorgere con il server. Una volta che sarete sicuri del corretto funzionamento del server, potrete scrivere l'applicazione client. Il client riceverà come parametro il nome IP che deve essere risolto, aprirà una connessione socket al server, scriverà il nome IP da risolvere e leggerà la risposta restituita dal server. Ad esempio, se il client si chiama `NSClient`, potrà essere invocato come segue:

```
java NSClient www.westminstercollege.edu
```

e il server risponderà con il corrispondente indirizzo IP oppure con il messaggio “host sconosciuto”. Il client chiuderà la sua connessione socket dopo aver restituito l'indirizzo IP.

## Progetto 2: Progetto di moltiplicazione di matrici

Date due matrici  $A$  e  $B$ , dove  $A$  è una matrice con  $M$  righe e  $K$  colonne, mentre la matrice  $B$  contiene  $K$  righe e  $N$  colonne, il prodotto di  $A$  e  $B$  è la matrice  $C$ , contenente  $M$  righe e  $N$  colonne, definita come segue. L'elemento  $C_{i,j}$  della matrice  $C$  alla riga  $i$  e colonna  $j$  è la somma dei prodotti degli elementi che appartengono alla riga  $i$  nella matrice  $A$  e alla colonna  $j$  nella matrice  $B$ . Quindi,

$$C_{i,j} = \sum_{n=1}^K A_{i,n} \times B_{n,j}$$

Posto, per esempio, che  $A$  sia una matrice 3 per 2 e  $B$  una matrice 2 per 3, l'elemento  $C_{3,1}$  risulterebbe dalla somma di  $A_{3,1} \times B_{1,1}$  e  $A_{3,2} \times B_{2,1}$ .

In questo progetto dovrete calcolare ogni elemento  $C_{i,j}$  in un *thread di lavoro* separato. Sarà dunque opportuno creare  $M \times N$  thread di lavorazione. Il thread principale – o genitore – inizializzerà le matrici  $A$  e  $B$ , allocando memoria in quantità sufficiente per la matrice  $C$ , che conterrà il prodotto delle matrici  $A$  e  $B$ . Tali matrici saranno dichiarate come dati globali, di modo che ciascun thread di lavorazione abbia accesso ad  $A$ ,  $B$  e  $C$ .

Le matrici  $A$  e  $B$  possono essere inizializzate staticamente, come illustrato di seguito.

```
#define M 3
#define K 2
#define N 3
int A [M][K] = { {1,4}, {2,5}, {3,6} };
int B [K][N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

In alternativa, possono essere riempite leggendo i valori da un file.

## Passaggio dei parametri a ciascun thread

Il thread genitore darà luogo a  $M \times N$  thread di lavoro, passando a ciascun di loro i valori della riga  $i$  e della colonna  $j$  di cui ha bisogno per determinare il prodotto della matrice. È quindi necessario passare due parametri a ogni thread. Il modo più semplice con Pthreads e

in Win32 è creare una struttura dati tramite `struct`. I membri di questa struttura sono `i` e `j`, e l'aspetto della struttura è il seguente:

```
/* struttura per il passaggio dei dati ai thread */
struct v
{
    int i; /* riga */
    int j; /* colonna */
};
```

I programmi Pthreads e Win32 useranno una strategia simile a quella riportata di seguito per creare i thread di lavoro.

```
/* Occorre creare M * N thread di lavoro */
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++) {
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data->i = i;
        data->j = j;
        /* Adesso create il thread passandogli data come parametro */
    }
}
```

Il puntatore `data` verrà passato alla funzione `pthread_create()` (Pthreads) o alla funzione `CreateThread()` (Win32), che a sua volta lo passerà come parametro alla funzione da eseguire in qualità di thread separato.

La condivisione dei dati tra i thread di Java differisce da quanto appena visto. Una possibilità è di creare e inizializzare le matrici  $A$ ,  $B$  e  $C$  nel thread principale (contenente il `main()`), che genererà quindi i thread di lavoro, passando le tre matrici – insieme alla riga  $i$  e alla colonna  $j$  – al costruttore di ogni thread di lavoro. Un thread di lavoro assume, dunque, l'aspetto riportato nella Figura 4.15.

## Attesa per il completamento dei thread

Una volta che tutti i thread di lavoro abbiano terminato, il thread principale restituirà il prodotto contenuto nella matrice  $C$ . Esso dovrà perciò attendere la conclusione di tutti i thread di lavoro prima di poter emettere il valore del prodotto della matrice. Esistono molti modi diversi per sospendere un thread nell'attesa che gli altri si concludano. Il Paragrafo 4.3 spiega come un thread genitore possa attendere che il thread figlio termini usando le librerie Win32, Pthreads e Java. Win32 fornisce la funzione `WaitForSingleObject()`, mentre Pthreads e Java adoperano, rispettivamente, `phtread_join()` e `join()`. Negli esempi esaminati fin qui, però, il thread genitore attende che un solo thread figlio termini; lo svolgimento dell'esercizio implica, invece, l'attesa del completamento di più thread.

Nel Paragrafo 4.3.2 diamo conto della funzione `WaitForSingleObject()`, che ha lo scopo di attendere la terminazione di un singolo thread. Ma la API Win32 possiede anche la funzione `WaitForMultipleObjects()`, usata quando è necessario attendere che terminino diversi thread. Essa accetta quattro parametri.

```

public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

    public WorkerThread(int row, int col, int[][] A,
                        int[][] B, int[][] C) {
        this.row = row;
        this.col = col;
        this.A = A;
        this.B = B;
        this.C = C;
    }
    public void run() {
        /* calcola il prodotto riga per colonna */
        /* e ponetelo in C[row] [col] */
    }
}

```

**Figura 4.15** Codice Java di WorkerThread.

1. Il numero di oggetti da attendere.
2. Un puntatore al vettore degli oggetti.
3. Un flag che indichi se occorre attendere la terminazione di tutti i thread, o di almeno uno di loro.
4. Un limite di tempo massimo per l'attesa, o il valore convenzionale **INFINITE**.

Supponiamo, per esempio, che **THandles** sia un vettore di dimensione **N** contenente thread rappresentati da oggetti di tipo **HANDLE**. In questa ipotesi, il thread genitore può attendere che tutti i propri thread figli terminino con l'istruzione:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

Una soluzione comoda per attendere la terminazione di più thread usando **pthread\_join()** di Pthreads o **join()** di Java è racchiudere questi comandi in un semplice ciclo **for**. Potreste, per esempio, attendere la terminazione di dieci thread usando il codice Pthreads riportato nella Figura 4.16. Nel caso di Java, il codice equivalente è mostrato nella Figura 4.17.

```
#define NUM_THREADS 10

/* Array dei thread di cui si attende la terminazione */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

**Figura 4.16** Codice Pthread per attendere la terminazione di dieci thread.

```
final static int NUM_THREADS = 10;

/* Array dei thread di cui si attende la terminazione */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
    try {
        workers[i].join();
    } catch (InterruptedException ie) {}
}
```

**Figura 4.17** Codice Java per attendere la terminazione di dieci thread.

## 4.7 Note bibliografiche

I thread hanno avuto una lunga evoluzione, a partire dalla “concorrenza a basso costo” nei linguaggi di programmazione, passando dai “processi leggeri”, con i primi esempi tra i quali il sistema Thoth (Cheriton et al. [1979] e il sistema Pilot (Redell et al. [1980]). Binding [1985] ha descritto come i thread siano stati incorporati all’interno del kernel UNIX. Mach (Accetta et al. [1986], Tevanian et al. [1987a] e V (Cheriton [1988]) hanno fatto ampio uso dei thread. Alla fine di questo percorso, quasi tutti i principali sistemi operativi hanno implementato i thread in una forma o nell’altra.

Gli aspetti relativi alle prestazioni dei thread sono affrontati in [Anderson et al. 1989] e in [Anderson et al. 1991], dove si valutano le prestazioni dei thread a livello utente. [Bershad et al. 1990] trattano la combinazione di thread e RPC. [Engelschall 2000] illustra una tecnica per l’implementazione dei thread a livello utente. Un’analisi della dimensione ottimale dei gruppi di thread (thread pool) si trova in Ling et al. [2000]. Le attivazioni dello scheduler sono state presentate per la prima volta in Anderson et al. [1991], mentre Williams [2002] ha discusso le attivazioni dello scheduler nel sistema NetBSD. Altri meccanismi per la cooperazione fra la libreria dei thread a livello utente e del kernel sono esposti in [Marsh et al. 1991], [Govindan e Anderson 1991], [Draves et al. 1991] e [Black 1990]. [Zabatta e Young 1998] confrontano i thread del sistema Windows NT e di Solaris su un’architettura SMP. [Pinilla e Gill 2003] confrontano le prestazioni dei thread di Java sui sistemi Linux, Windows e Solaris.

[Vahalia 1996] tratta l’uso dei thread in diverse versioni di UNIX. [Mauro e McDougall 2007] descrivono i recenti sviluppi relativi all’uso dei thread nel kernel del Solaris. [Solomon e Russinovich 2000] descrivono la realizzazione dei thread nel sistema operativo Windows 2000. [Bovet e Cesati

2006] e Love [2004] descrivono il threading in Linux, mentre Singh [2007] tratta lo stesso argomento per Mac OS X.

Informazioni sulla programmazione Pthreads si trovano in [Lewis e Berg 1998], oltre che in [Butenhof 1997]. [Oaks e Wong 1999], [Lewis e Berg 2000] e [Holub 1998] analizzano la programmazione multithread nel linguaggio Java. Goetz et al. [2006] presentano una trattazione dettagliata della programmazione concorrente in Java. [Beveridge e Wiener 1997] e [Cohen e Woodring 1997] affrontano il tema della programmazione multithread con Win32.

## Capitolo 5

# Scheduling della CPU



### OBIETTIVI

- Introduzione allo scheduling della CPU, base dei sistemi operativi multiprogrammati.
- Descrizione di vari algoritmi di scheduling della CPU.
- Analisi dei criteri di valutazione nella scelta degli algoritmi di scheduling della CPU per particolari sistemi.

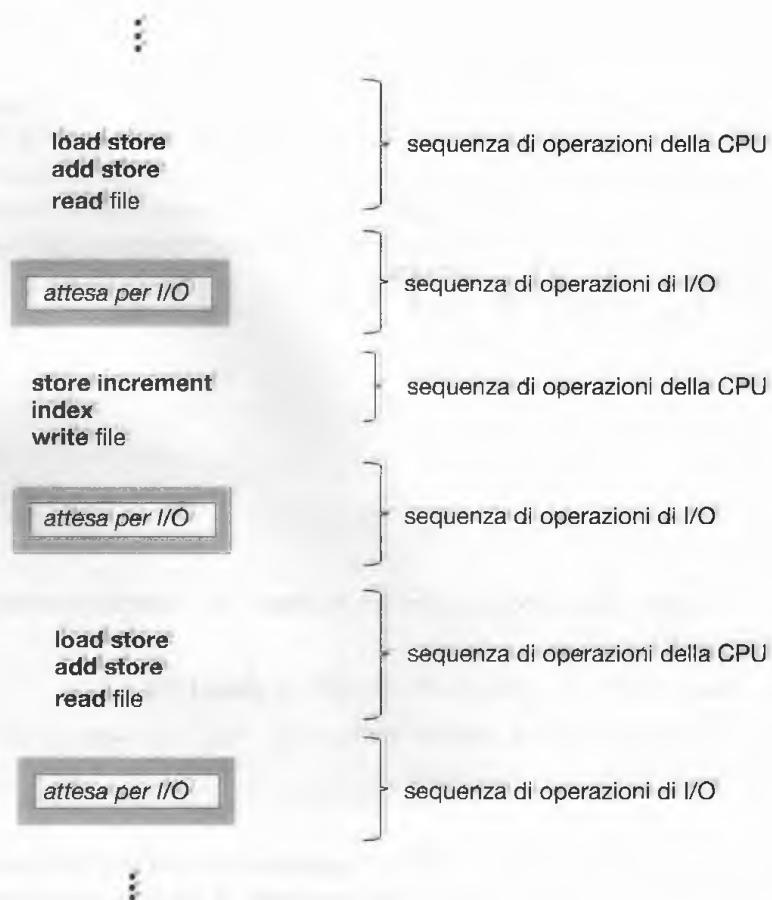
Lo scheduling della CPU è alla base dei sistemi operativi multiprogrammati: attraverso la commutazione del controllo della CPU tra i vari processi, il sistema operativo può rendere più produttivo il calcolatore. In questo capitolo s'introducono i concetti fondamentali dello scheduling e si descrivono vari algoritmi di scheduling della CPU. Si affronta inoltre il problema della scelta dell'algoritmo da impiegare per un dato sistema.

Nel Capitolo 4 abbiamo arricchito il concetto di processo introducendo i thread. Nei sistemi operativi che li contemplano, sono loro, in effetti, l'oggetto dell'attività di scheduling, e non i processi. Ciononostante, le locuzioni **scheduling dei processi** e **scheduling dei thread** sono spesso considerate equivalenti. In questo capitolo useremo la prima nell'analizzare i principi generali dello scheduling, e la seconda nel trattare idee specificamente inerenti ai thread.

## 5.1 Concetti fondamentali

In un sistema monoprocesso si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la CPU sia libera e possa essere nuovamente sottoposta a scheduling.

L'idea della multiprogrammazione è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. Con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della CPU per cederlo a un altro processo.



**Figura 5.1** Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la CPU è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

### 5.1.1 Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un **ciclo** d'elaborazione (svolta dalla CPU) e d'attesa del completamento delle operazioni di I/O. I processi si alternano tra questi due stati. L'esecuzione di un processo comincia con una sequenza (una "raffica") di operazioni d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*I/O burst*), quindi un'altra sequenza di operazioni della CPU, di nuovo una sequenza di operazioni di I/O, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione (Figura 5.1).

Le durate delle sequenze di operazioni della CPU sono state misurate, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella Figura 5.2. La curva è generalmente di tipo esponenziale o iperesponenziale, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe. Un programma con prevalenza di I/O (*I/O bound*) produce generalmente molte sequenze di operazioni della CPU di breve durata. Un programma con prevalenza d'elaborazione (*CPU bound*), invece, produce poche sequenze di operazioni della CPU molto lunghe. Queste caratteristiche possono essere utili nella scelta di un appropriato algoritmo di scheduling della CPU.



Figura 5.2 Diagramma delle durate delle sequenze di operazioni della CPU.

### 5.1.2 Scheduler della CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella coda dei processi pronti. In particolare, è lo **scheduler a breve termine**, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

La coda dei processi pronti non è necessariamente una coda in ordine d'arrivo (*first-in, first-out*, FIFO). Come si nota analizzando i diversi algoritmi di scheduling, una coda dei processi pronti si può realizzare come una coda FIFO, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i processi della coda dei processi pronti sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

### 5.1.3 Scheduling con diritto di prelazione

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:

1. un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o richiesta di attesa (*wait*) per la terminazione di uno dei processi figli);
2. un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione);
3. un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O);
4. un processo termina.

I casi 1 e 4 in quanto tali non comportano alcuna scelta di scheduling; a essi segue la scelta di un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella coda dei processi pronti per l'esecuzione. Una scelta si deve invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza diritto di prelazione** (*nonpreemptive*) o **cooperativo** (*cooperative*); altrimenti, lo schema di scheduling è **con diritto di prelazione** (*preemptive*). Nel caso dello scheduling senza diritto di prelazione, quando si assegna la CPU a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al

passaggio nello stato di attesa. Questo metodo di scheduling è stato adottato da Microsoft Windows 3.x; lo scheduling con diritto di prelazione è stato introdotto a partire da Windows 95 e in tutte le versioni successive. È adottato anche dal sistema Mac OS X; le versioni precedenti del sistema operativo Macintosh si basavano sullo scheduling cooperativo. Poiché, diversamente dallo scheduling con diritto di prelazione, non richiede dispositivi particolari (per esempio i timer), lo scheduling cooperativo è l'unico metodo utilizzabile su alcune piattaforme.

Sfortunatamente lo scheduling con diritto di prelazione presenta un inconveniente. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi; questo argomento è trattato nel Capitolo 6.

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività in favore di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos. Alcuni sistemi operativi, tra cui la maggior parte delle versioni dello UNIX, affrontano questo problema attendendo il completamento della chiamata di sistema o che si abbia il blocco dell'I/O prima di eseguire un cambio di contesto. Quindi, il kernel non può esercitare la prelazione su un processo mentre le strutture dati del kernel si trovano in uno stato incoerente. Sfortunatamente, questo modello d'esecuzione del kernel non è adeguato alle computazioni in tempo reale e alla multielaborazione. Questi problemi e le relative soluzioni sono descritti nei Paragrafi 5.5 e 19.5.

Poiché le interruzioni si possono, per definizione, verificare in ogni istante e il kernel non può sempre ignorare, le sezioni di codice eseguite per effetto delle interruzioni devono essere protette da un uso simultaneo. Il sistema operativo deve ignorare raramente le interruzioni, altrimenti si potrebbero perdere dati in ingresso, o si potrebbero sovrascrivere dati in uscita. Per evitare che più processi accedano in modo concorrente a tali sezioni di codice, queste disattivano le interruzioni al loro inizio e le riattivano alla fine. Le sezioni di codice che disabilitano le interruzioni si verificano raramente e, in genere, non contengono molte istruzioni.

### 5.1.4 Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della CPU è il **dispatcher**; si tratta del modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Questa funzione riguarda quel che segue:

- ◆ il cambio di contesto;
- ◆ il passaggio alla modalità utente;
- ◆ il salto alla giusta posizione del programma utente per riavviare l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è nota come **latenza di dispatch**.

## 5.2 Criteri di scheduling

Diversi algoritmi di scheduling della CPU hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi.

Per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore. Di seguito si riportano alcuni criteri.

- ◆ **Utilizzo della CPU.** La CPU deve essere più attiva possibile. Teoricamente, l'utilizzo della CPU può variare dallo 0 al 100 per cento. In un sistema reale può variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.
- ◆ **Produttività.** La CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta **produttività** (*throughput*). Per processi di lunga durata questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di 10 processi al secondo.
- ◆ **Tempo di completamento.** Considerando un processo particolare, un criterio importante può essere relativo al tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato **tempo di completamento** (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione nella CPU e nelle operazioni di I/O.
- ◆ **Tempo d'attesa.** L'algoritmo di scheduling della CPU non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo d'attesa nella coda dei processi pronti. Il **tempo d'attesa** è la somma degli intervalli d'attesa passati nella coda dei processi pronti.
- ◆ **Tempo di risposta.** In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continui a calcolare i nuovi risultati mentre quelli precedenti sono in fase d'emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione di una richiesta e la prima risposta prodotta. Questa misura è chiamata **tempo di risposta**, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo d'emissione. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo d'emissione dei dati.

È auspicabile aumentare al massimo utilizzo e produttività della CPU, mentre il tempo di completamento, il tempo d'attesa e il tempo di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi; in alcune circostanze è più opportuno ottimizzare i valori minimi o massimi, anziché i valori medi; per esempio, per garantire che tutti gli utenti ottengano un buon servizio, può essere utile ridurre il massimo tempo di risposta.

Per i sistemi interattivi, come i sistemi a tempo ripartito, alcuni analisti suggeriscono che sia più importante ridurre al minimo la **varianza** del tempo di risposta anziché il tempo medio di risposta. Un sistema il cui tempo di risposta sia ragionevole e prevedibile può essere considerato migliore di un sistema mediamente più rapido, ma molto variabile. Tuttavia, è stato fatto poco sugli algoritmi di scheduling della CPU al fine di ridurre al minimo la varianza.

Poiché si analizzano diversi algoritmi di scheduling della CPU, è opportuno che se ne illustri anche il funzionamento. Una spiegazione approfondita richiederebbe il ricorso a molti processi, ognuno dei quali costituito da parecchie centinaia di sequenze di operazioni della CPU e di sequenze di operazioni di I/O. Per motivi di semplicità, negli esempi si considera una sola sequenza di operazioni della CPU (la cui durata è espressa in millisecondi) per ogni processo. La misura di confronto adottata è il tempo d'attesa medio. Meccanismi di valutazione più raffinati sono trattati nel Paragrafo 5.7.

## 5.3 Algoritmi di scheduling

Lo scheduling della CPU riguarda la scelta dei processi presenti nella coda dei processi pronti cui assegnare la CPU. In questo paragrafo si descrivono alcuni fra i tanti algoritmi di scheduling della CPU.

### 5.3.1 Scheduling in ordine d'arrivo

Il più semplice algoritmo di scheduling della CPU è l'algoritmo di **scheduling in ordine d'arrivo** (*scheduling first-come, first-served*, FCFS). Con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si fonda su una coda FIFO. Quando un processo entra nella coda dei processi pronti, si collega il suo PCB all'ultimo elemento della coda. Quando è libera, si assegna la CPU al processo che si trova alla testa della coda dei processi pronti, rimuovendolo da essa. Il codice per lo scheduling FCFS è semplice sia da scrivere sia da capire.

Il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo. Si consideri il seguente insieme di processi, che si presenta al momento 0, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
$P_1$	24
$P_2$	3
$P_3$	3

Se i processi arrivano nell'ordine  $P_1$ ,  $P_2$ ,  $P_3$  e sono serviti in ordine FCFS, si ottiene il risultato illustrato dal seguente diagramma di Gantt, un istogramma che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.



Il tempo d'attesa è 0 millisecondi per il processo  $P_1$ , 24 millisecondi per il processo  $P_2$  e 27 millisecondi per il processo  $P_3$ . Quindi, il tempo d'attesa medio è  $(0 + 24 + 27)/3 = 17$  mil-

lisecondi. Se i processi arrivassero nell'ordine  $P_2, P_3, P_1$ , i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:



Il tempo di attesa medio è ora di  $(6 + 0 + 3)/3 = 3$  millisecondi. Si tratta di una notevole riduzione. Quindi, il tempo medio d'attesa in condizioni di FCFS non è in genere minimo, e può variare sostanzialmente al variare della durata delle sequenze di operazioni della CPU dei processi.

Si considerino inoltre le prestazioni dello scheduling FCFS in una situazione dinamica. Si supponga di avere un processo con prevalenza d'elaborazione e molti processi con prevalenza di I/O. Via via che i processi fluiscono nel sistema si può riscontrare come il processo con prevalenza d'elaborazione occupi la CPU. Durante questo periodo tutti gli altri processi terminano le proprie operazioni di I/O e si spostano nella coda dei processi pronti, nell'attesa della CPU. Mentre i processi si trovano nella coda dei processi pronti, i dispositivi di I/O sono inattivi. Successivamente il processo con prevalenza d'elaborazione termina la propria sequenza di operazioni della CPU e passa a una fase di I/O. Tutti i processi con prevalenza di I/O, caratterizzati da sequenze di operazioni della CPU molto brevi, sono eseguiti rapidamente e tornano alle code di I/O, lasciando inattiva la CPU. Il processo con prevalenza d'elaborazione torna nella coda dei processi pronti e riceve il controllo della CPU; così, finché non termina l'esecuzione del processo con prevalenza d'elaborazione, tutti i processi con prevalenza di I/O si trovano nuovamente ad attendere nella coda dei processi pronti. Si ha un **effetto convoglio**, tutti i processi attendono che un lungo processo liberi la CPU, che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è senza prelazione; una volta che la CPU è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O. L'algoritmo FCFS risulta particolarmente problematico nei sistemi a tempo ripartito, dove è importante che ogni utente disponga della CPU a intervalli regolari. Permettere a un solo processo di occupare la CPU per un lungo periodo condurrebbe a risultati disastrosi.

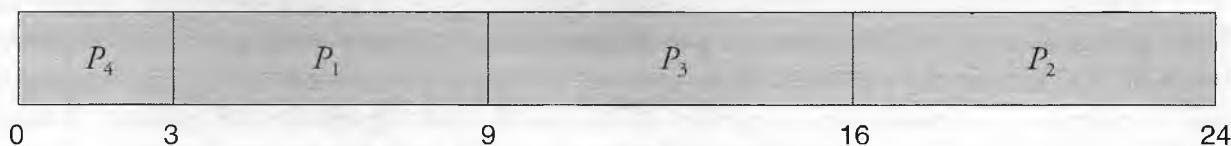
### 5.3.2 Scheduling per brevità

Un criterio diverso di scheduling della CPU si può ottenere con l'algoritmo di **scheduling per brevità** (*shortest-job-first*, SJF). Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della CPU della stessa lunghezza si applica lo scheduling FCFS. Si noti che sarebbe più appropriato il termine *shortest next CPU first*, infatti lo scheduling si esegue esaminando la lunghezza della successiva sequenza di operazioni della CPU del processo e non la sua lunghezza totale. Tuttavia, poiché è usato nella maggior parte dei libri di testo, anche qui si fa uso del termine SJF.

Come esempio si consideri il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

Con lo scheduling SJF questi processi si ordinerebbero secondo il seguente diagramma di Gantt.



Il tempo d'attesa è di 3 millisecondi per il processo  $P_1$ , di 16 millisecondi per il processo  $P_2$ , di 9 millisecondi per il processo  $P_3$  e di 0 millisecondi per il processo  $P_4$ . Quindi, il tempo d'attesa medio è di  $(3 + 16 + 9 + 0)/4 = 7$  millisecondi. Usando lo scheduling FCFS, il tempo d'attesa medio sarebbe di 10,25 millisecondi.

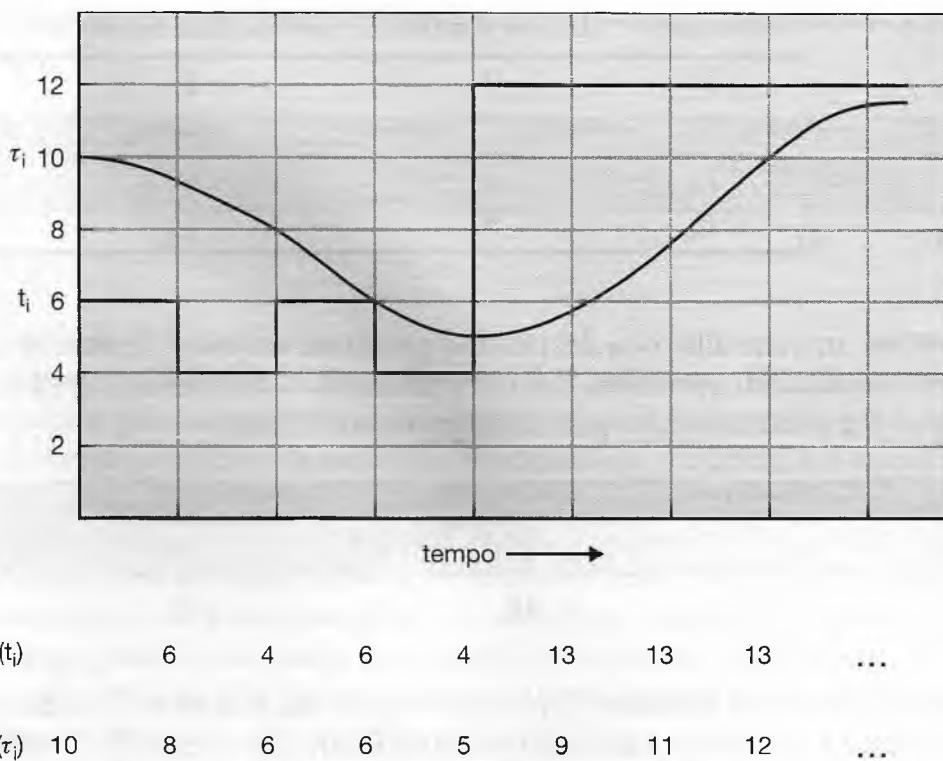
Si può dimostrare che l'algoritmo di scheduling SJF è *ottimale*, nel senso che rende minimo il tempo d'attesa medio per un dato insieme di processi. Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa *medio* diminuisce.

La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Per lo scheduling a lungo termine (*job scheduling*) di un sistema a lotti si può usare come durata il tempo limite d'elaborazione che gli utenti specificano nel sottoporre il processo. Gli utenti sono quindi motivati a stabilire con precisione tale limite, poiché un valore inferiore può significare una risposta più rapida. Occorre però notare che un valore troppo basso causa un errore di superamento del tempo limite e richiede una nuova esecuzione. Lo scheduling SJF si usa spesso nello scheduling a lungo termine.

Sebbene sia ottimale, l'algoritmo SJF non si può realizzare a livello dello scheduling della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU. Un possibile metodo consiste nel tentare di approssimare lo scheduling SJF: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della CPU, si può cercare di *predire* il suo valore; è probabile, infatti, che sia simile ai precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

La lunghezza della successiva sequenza di operazioni della CPU generalmente si ottiene calcolando la **media esponenziale** delle effettive lunghezze delle precedenti sequenze di operazioni della CPU. La media esponenziale si definisce con la formula seguente. Denotando con  $t_n$  la lunghezza dell' $n$ -esima sequenza di operazioni della CPU e con  $\tau_{n+1}$  il valore previsto per la successiva sequenza di operazioni della CPU, con  $\alpha$  tale che  $0 \leq \alpha \leq 1$ , si definisce

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$



**Figura 5.3** Predizione della lunghezza della successiva sequenza di operazioni della CPU (CPU burst).

Il valore di  $t_n$  contiene le informazioni più recenti;  $\tau_n$  registra la storia passata. Il parametro  $\alpha$  controlla il peso relativo sulla predizione della storia recente e di quella passata. Se  $\alpha = 0$ , allora,  $\tau_{n+1} = \tau_n$ , e la storia recente non ha effetto; si suppone, cioè, che le condizioni attuali siano transitorie; se  $\alpha = 1$ , allora  $\tau_{n+1} = t_n$ , e ha significato solo la più recente sequenza di operazioni della CPU: si suppone, cioè, che la storia sia vecchia e irrilevante. Più comune è la condizione in cui  $\alpha = 1/2$ , valore che indica che la storia recente e la storia passata hanno lo stesso peso. Il  $\tau_0$  iniziale si può definire come una costante o come una media complessiva del sistema. Nella Figura 5.3 è illustrata una media esponenziale con  $\alpha = 1/2$  e  $\tau_0 = 10$ .

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula per  $\tau_{n+1}$  sostituendo in  $\tau_n$ , in modo da ottenere

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j\alpha t_{n-j} + \dots + (1 - \alpha)^{n+1}\tau_0$$

Poiché sia  $\alpha$  sia  $(1 - \alpha)$  sono minori o uguali a 1, ogni termine ha peso inferiore a quello del suo predecessore.

L'algoritmo SJF può essere sia *con prelazione* sia *senza prelazione*. La scelta si presenta quando alla coda dei processi pronti arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può avere una successiva sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo SJF con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo SJF senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della CPU. (Lo scheduling SJF con prelazione è talvolta chiamato *scheduling shortest-remaining-time-first*.)

Come esempio, si considerino i quattro processi seguenti, dove la durata delle sequenze di operazioni della CPU è data in millisecondi:

Processo	Istante d'arrivo	Durata della sequenza
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

Se i processi arrivano alla coda dei processi pronti nei momenti indicati e richiedono le durate delle sequenze di operazioni della CPU illustrate, dallo scheduling SJF con prelazione risulta quel che è indicato dal seguente diagramma di Gantt.



26

All'istante 0 si avvia il processo  $P_1$ , poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo  $P_2$ . Il tempo necessario per completare il processo  $P_1$  (7 millisecondi) è maggiore del tempo richiesto dal processo  $P_2$  (4 millisecondi), perciò si ha la prelazione sul processo  $P_1$  sostituendolo col processo  $P_2$ . Il tempo d'attesa medio per questo esempio è  $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6,5$  millisecondi. Con uno scheduling SJF senza prelazione si otterebbe un tempo d'attesa medio di 7,75 millisecondi.

### 5.3.3 Scheduling per priorità

L'algoritmo SJF è un caso particolare del più generale **algoritmo di scheduling per priorità**: si associa una priorità a ogni processo e si assegna la CPU al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS. Un algoritmo SJF è semplicemente un algoritmo con priorità in cui la priorità ( $p$ ) è l'inverso della lunghezza (prevista) della successiva sequenza di operazioni della CPU. A una maggiore lunghezza corrisponde una minore priorità, e viceversa.

Occorre notare che la discussione si svolge nei termini di priorità *alta* e priorità *bassa*. Generalmente le priorità sono indicate da un intervallo fisso di numeri, come da 0 a 7, oppure da 0 a 4.095. Tuttavia, non si è ancora stabilito se attribuire allo 0 la priorità più alta o quella più bassa; alcuni sistemi usano numeri bassi per rappresentare priorità basse, altri usano numeri bassi per priorità alte. In questo testo i numeri bassi indicano priorità alte.

Come esempio, si consideri il seguente insieme di processi, che si suppone siano arrivati al tempo 0, nell'ordine  $P_1, P_2 \dots P_5$ , e dove la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Processo	Durata della sequenza	Priorità
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

Usando lo scheduling per priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt.



Il tempo d'attesa medio è di 8,2 millisecondi.

Le priorità si possono definire sia internamente sia esternamente. Quelle definite internamente usano una o più quantità misurabili per calcolare la priorità del processo; per esempio, i limiti di tempo, i requisiti di memoria, il numero dei file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni di I/O e la lunghezza media delle sequenze di operazioni della CPU. Le priorità esterne si definiscono secondo criteri esterni al sistema operativo, come l'importanza del processo, il tipo e la quantità dei fondi pagati per l'uso del calcolatore, il dipartimento che promuove il lavoro e altri fattori, spesso di ordine politico.

Lo scheduling per priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla coda dei processi pronti, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling per priorità e con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità dell'ultimo processo arrivato è superiore. Un algoritmo di scheduling senza diritto di prelazione si limita a porre l'ultimo processo arrivato alla testa della coda dei processi pronti.

Un problema importante relativo agli algoritmi di scheduling per priorità è l'**attesa indefinita** (*starvation*, letteralmente, "inedia"). Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling per priorità può lasciare processi con bassa priorità nell'attesa indefinita della CPU. Un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU. Generalmente accade che o il processo è eseguito, alle ore 2 del mattino della domenica, quando il sistema ha finalmente ridotto il proprio carico, oppure il calcolatore si sovraccarica al punto da perdere tutti i processi con bassa priorità non terminati. Corre voce che, quando fu fermato l'IBM 7094 al MIT, nel 1973, si scoprì che un processo con bassa priorità sottoposto nel 1967 non era ancora stato eseguito.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento** (*aging*); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo. Per esempio, se le priorità variano da 127 (bassa) a 0 (alta), si potrebbe decrementare di 1 ogni 15 minuti la priorità di un processo in attesa. Anche un processo con priorità iniziale 127 può ottenere la priorità massima nel sistema e quindi essere eseguito: un processo con priorità 127 non impiega più di 32 ore per raggiungere la priorità 0.

### 5.3.4 Scheduling circolare

L'algoritmo di scheduling circolare (*round-robin*, RR) è stato progettato appositamente per i sistemi a tempo ripartito; simile allo scheduling FCFS, ha tuttavia in più la capacità di prelazione per la commutazione dei processi. Ciascun processo riceve una piccola quantità fissata del tempo della CPU, chiamata **quanto di tempo** o **porzione di tempo** (*time slice*), che varia generalmente da 10 a 100 millisecondi; la coda dei processi pronti è trattata come una coda circolare. Lo scheduler della CPU scorre la coda dei processi pronti, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

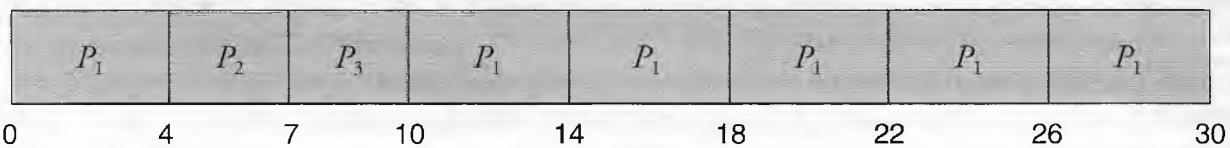
Per realizzare lo scheduling RR si gestisce la coda dei processi pronti come una coda FIFO. I nuovi processi si aggiungono alla fine della coda dei processi pronti. Lo scheduler della CPU individua il primo processo dalla coda dei processi pronti, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per l'effettiva esecuzione del processo.

A questo punto si può verificare una delle seguenti situazioni: il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della coda dei processi pronti; oppure la durata della sequenza di operazioni della CPU del processo attualmente in esecuzione è più lunga di un quanto di tempo; in questo caso si raggiunge la scadenza del quanto di tempo e il timer invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto, aggiunge il processo alla fine della coda dei processi pronti e, tramite lo scheduler della CPU, seleziona il processo successivo nella coda dei processi pronti. Il tempo d'attesa medio per il criterio di scheduling RR è spesso abbastanza lungo.

Si consideri il seguente insieme di processi che si presenta al tempo 0, con la durata delle sequenze di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
$P_1$	24
$P_2$	3
$P_3$	3

Se si usa un quanto di tempo di 4 millisecondi, il processo  $P_1$  ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la CPU passa al processo successivo della coda, il processo  $P_2$ . Poiché il processo  $P_2$  non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la CPU al processo successivo, il processo  $P_3$ . Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la CPU al processo  $P_1$  per un ulteriore quanto di tempo. Dallo scheduling RR risulta quanto segue.



Calcoliamo ora il tempo di attesa medio per questo scheduling.  $P_1$  resta in attesa per 6 millisecondi (10-4),  $P_2$  per 4 millisecondi e  $P_3$  per 7 millisecondi. Il tempo d'attesa medio è di  $17/3 = 5,66$  millisecondi.

Nell'algoritmo di scheduling RR la CPU si assegna a un processo per non più di un quanto di tempo per volta. Se la durata della sequenza di operazioni della CPU di un processo eccede il quanto di tempo, il processo viene sottoposto a *prelazione* e riportato nella coda dei processi pronti. L'algoritmo di scheduling RR è pertanto con prelazione.

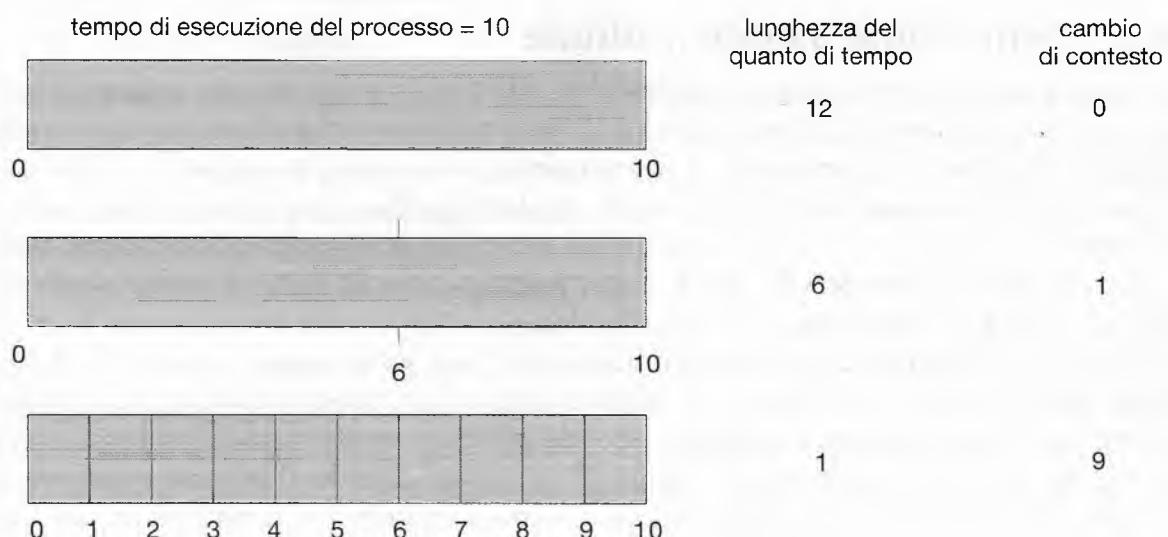
Se nella coda dei processi pronti esistono  $n$  processi e il quanto di tempo è pari a  $q$ , ciascun processo ottiene  $1/n$ -esimo del tempo di elaborazione della CPU in frazioni di, al massimo,  $q$  unità di tempo. Ogni processo non deve attendere per più di  $(n - 1) \times q$  unità di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa 20 millisecondi ogni 100 millisecondi.

Le prestazioni dell'algoritmo RR dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo (indefinito), il criterio di scheduling RR si riduce al criterio di scheduling FCFS. Se il quanto di tempo è molto breve (per esempio, un microsecondo), il criterio RR si chiama **condivisione della CPU (processor sharing)** e teoricamente gli utenti hanno l'impressione che ciascuno degli  $n$  processi disponga di una propria CPU in esecuzione a  $1/n$  della velocità della CPU reale. Questo metodo fu usato nell'architettura del sistema della Control Data Corporation (CDC) per simulare 10 unità d'elaborazione con 10 gruppi di registri e una sola CPU, che eseguiva un'istruzione per un gruppo di registri, quindi procedeva col successivo. Questo ciclo continuava, con il risultato che si avevano 10 unità d'elaborazione lente al posto di una CPU veloce. (In effetti, poiché la CPU era molto più rapida della memoria e ogni istruzione faceva riferimento alla memoria, le unità d'elaborazione simulate non erano molto più lente di quanto sarebbero state dieci vere unità d'elaborazione.)

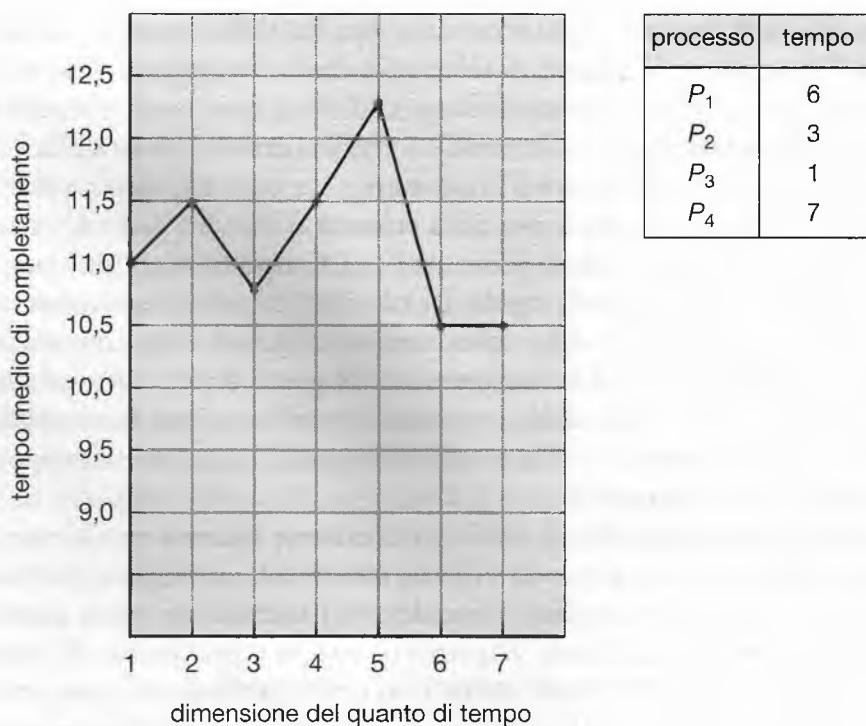
Riguardo alle prestazioni dello scheduling RR, occorre tuttavia considerare l'effetto dei cambi di contesto. Dato un solo processo della durata di 10 unità di tempo, se il quanto di tempo è di 12 unità, il processo impiega meno di un quanto di tempo; se però il quanto di tempo è di 6 unità, il processo richiede 2 quanti di tempo e un cambio di contesto; e se il quanto di tempo è di un'unità di tempo, occorrono nove cambi di contesto, con proporzionale rallentamento dell'esecuzione del processo. Tale situazione è visibile nello schema della Figura 5.4.

Da ciò che si è affermato segue che il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto; se, per esempio, questa è pari al 10 per cento del quanto di tempo, allora s'impiega nel cambio di contesto circa il 10 per cento del tempo d'elaborazione della CPU. In pratica, nella maggior parte dei sistemi moderni un quanto di tempo va dai 10 ai 100 millisecondi. Il tempo richiesto per un cambio di contesto non eccede solitamente i 10 microsecondi, risultando quindi una modesta frazione del quanto di tempo.

Anche il tempo di completamento (*turnaround time*) dipende dalla dimensione del quanto di tempo: com'è evidenziato nella Figura 5.5, il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della dimensione del quanto di tempo. In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva sequenza di operazioni della CPU in un solo



**Figura 5.4** Aumento del numero dei cambi di contesto al diminuire del quanto di tempo.



**Figura 5.5** Variazione del tempo di completamento in funzione del quanto di tempo.

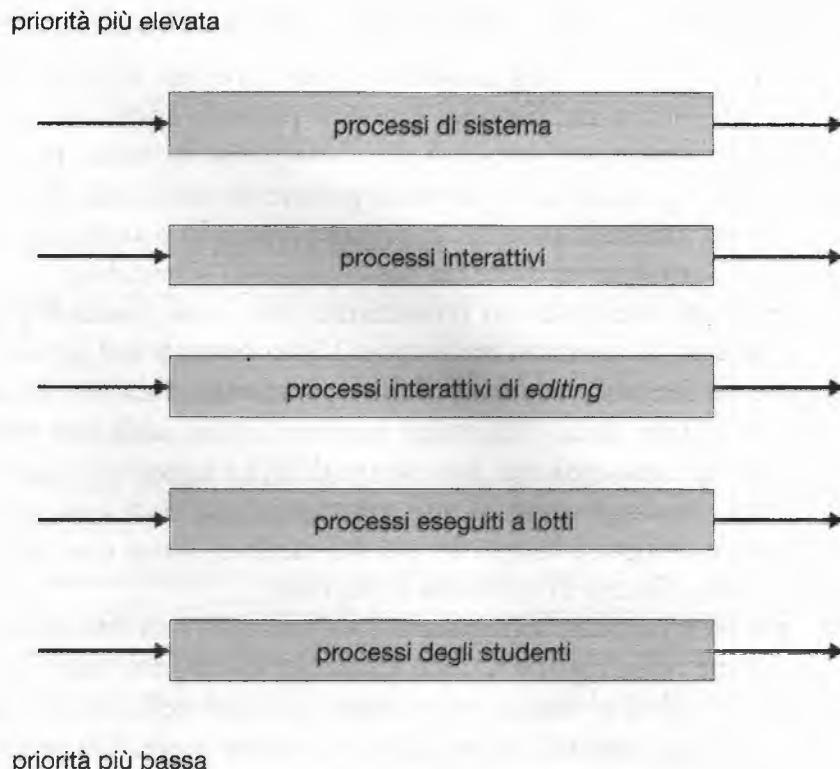
quanto di tempo. Per esempio, dati tre processi della durata di 10 unità di tempo ciascuno e un quanto di una unità di tempo, il tempo di completamento medio è di 29 unità. Se però il quanto di tempo è di 10 unità, il tempo di completamento medio scende a 20 unità. Aggiungendo il tempo del cambio di contesto, con un piccolo quanto di tempo, il tempo di completamento medio aumenta poiché sono richiesti più cambi di contesto.

È bene comunque che il quanto di tempo sia maggiore del tempo necessario al cambio di contesto: d'altro canto, è opportuno che lo scarto non sia eccessivo: anche se il quanto di tempo è molto ampio, il criterio di scheduling RR tende al criterio FCFS. Empiricamente si può stabilire che l'80 per cento delle sequenze di operazioni della CPU debba essere più breve del quanto di tempo.

### 5.3.5 Scheduling a code multiple

È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione diffusa è per esempio quella che si fa tra i processi che si eseguono in **primo piano** (*foreground*), o **interattivi**, e i processi che si eseguono in **sottofondo** (*background*), o a **lotti** (*batch*). Questi due tipi di processi hanno tempi di risposta diversi e possono quindi avere diverse necessità di scheduling. Inoltre, i processi che si eseguono in primo piano possono avere la priorità, definita esternamente, sui processi che si eseguono in sottofondo.

L'**algoritmo di scheduling a code multiple** (*multilevel queue scheduling algorithm*) suddivide la coda dei processi pronti in code distinte (Figura 5.6). I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità o il tipo. Ogni coda ha il proprio algoritmo di scheduling. Per esempio, per i processi in primo piano e i processi in sottofondo si possono usare code distinte. La coda dei processi in primo piano si può gestire con un algoritmo RR, mentre quella dei processi in sottofondo si può gestire con un algoritmo FCFS.



**Figura 5.6** Scheduling a code multiple.

In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunque di uno scheduling per priorità fissa e con prelazione. Per esempio, la coda dei processi in primo piano può avere la priorità assoluta sulla coda dei processi in sottofondo.

Si consideri il seguente algoritmo di scheduling a code multiple, in ordine di priorità:

1. processi di sistema;
2. processi interattivi;
3. processi interattivi di *editing*;
4. processi eseguiti in sottofondo;
5. processi degli studenti.

Ogni coda ha la priorità assoluta sulle code di priorità più bassa; nessun processo della coda dei processi in sottofondo può iniziare l'esecuzione finché le code per i processi di sistema, interattivi e interattivi di *editing* non siano tutte vuote. Se un processo interattivo di *editing* entrasse nella coda dei processi pronti durante l'esecuzione di un processo in sottofondo, si avrebbe la prelazione su quest'ultimo.

Esiste anche la possibilità di impostare i quanti di tempo per le code. Per ogni coda si stabilisce una parte del tempo d'elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono. Nell'esempio precedente, si può assegnare l'80 per cento del tempo d'elaborazione della CPU alla coda dei processi in primo piano, per lo scheduling RR tra i suoi processi; mentre per la coda dei processi in sottofondo si riserva il 20 per cento del tempo d'elaborazione della CPU, da assegnare ai suoi processi sulla base del criterio FCFS.

### 5.3.6 Scheduling a code multiple con retroazione

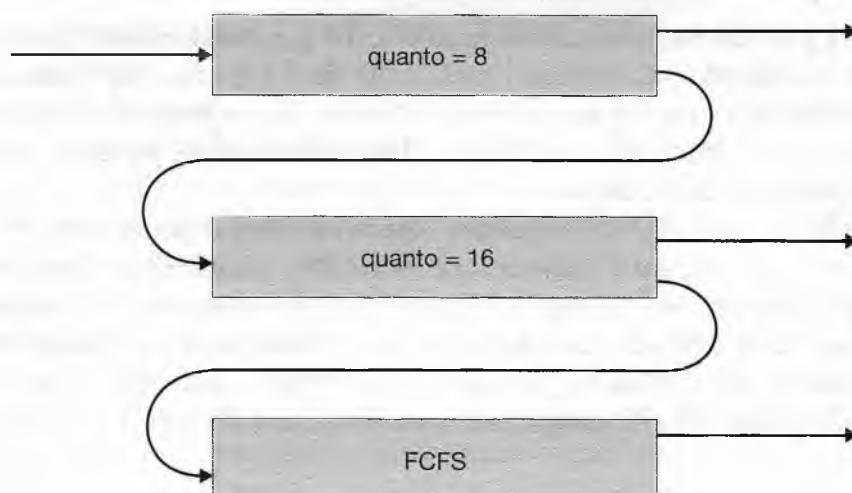
Di solito in un algoritmo di scheduling a code multiple i processi si assegnano in modo permanente a una coda all'entrata nel sistema, e non si possono spostare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in primo piano e quelli che si eseguono in sottofondo, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in primo piano o in sottofondo. Quest'impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

Lo scheduling a code multiple con retroazione (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. L'idea consiste nel separare i processi che hanno caratteristiche diverse nelle sequenze delle operazioni della CPU. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi con prevalenza di I/O e i processi interattivi nelle code con priorità più elevata. Analogamente, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo. In questo modo si attua una forma d'invecchiamento che impedisce il verificarsi di un'attesa indefinita.

Si consideri, per esempio, uno scheduler a code multiple con retroazione con tre code, numerate da 0 a 2, come nella Figura 5.7. Lo scheduler fa eseguire tutti i processi presenti nella coda 0; quando la coda 0 è vuota, si eseguono i processi nella coda 1; analogamente, i processi nella coda 2 si eseguono solo se le code 0 e 1 sono vuote. Un processo in ingresso nella coda 1 ha la prelazione sui processi della coda 2; un processo in ingresso nella coda 0, a sua volta, ha la prelazione sui processi della coda 1.

All'ingresso nella coda dei processi pronti, i processi vengono assegnati alla coda 0 e ottengono un quanto di tempo di 8 millisecondi; i processi che non terminano entro tale quanto di tempo, vengono spostati alla fine della coda 1. Se la coda 0 è vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio FCFS.

Questo algoritmo di scheduling dà la massima priorità ai processi con una sequenza di operazioni della CPU della durata di non più di 8 millisecondi. I processi di questo tipo ottengono rapidamente la CPU, terminano la propria sequenza di operazioni della CPU e passano alla successiva sequenza di operazioni di I/O; anche i processi che necessitano di più di 8 ma di non più di 24 millisecondi (coda 1) vengono serviti rapidamente. I processi più lun-



**Figura 5.7** Code multiple con retroazione.

ghi finiscono nella coda 2 e sono serviti secondo il criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code 0 e 1.

Generalmente uno scheduler a code multiple con retroazione è caratterizzato dai seguenti parametri:

- ◆ numero di code;
- ◆ algoritmo di scheduling per ciascuna coda;
- ◆ metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- ◆ metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- ◆ metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

La definizione di uno scheduler a code multiple con retroazione costituisce il più generale criterio di scheduling della CPU, che nella fase di progettazione si può adeguare a un sistema specifico. Sfortunatamente corrisponde anche all'algoritmo più complesso; la definizione dello scheduler migliore richiede infatti particolari metodi per la selezione dei valori dei diversi parametri.

## 5.4 Scheduling dei thread

Nel Capitolo 4 abbiamo arricchito il modello dei processi con i thread, distinguendo quelli *a livello utente* da quelli *a livello kernel*. Sui sistemi operativi che prevedono la loro presenza, il sistema pianifica l'esecuzione dei thread a livello kernel, non dei processi. I thread a livello utente sono gestiti da una libreria: il kernel non è consapevole della loro esistenza. Di conseguenza, per eseguire i thread a livello utente occorre associare loro dei thread a livello kernel. Tale associazione può essere indiretta, ossia realizzata con un processo leggero (LWP). Trattiamo adesso le questioni dello scheduling che riguardano i thread a livello utente e a livello kernel, offrendo esempi specifici dello scheduling per Pthreads.

### 5.4.1 Ambito della competizione

La prima distinzione fra thread a livello utente e a livello kernel riguarda il modo in cui è pianificata la loro esecuzione. Nei sistemi che impiegano il modello da molti a uno (Paragrafo 4.2.1) e il modello da molti a molti (Paragrafo 4.2.3), la libreria dei thread pianifica l'esecuzione dei thread a livello utente su un LWP libero; si parla allora di **ambito della competizione ristretto al processo** (*process-contention scope*, PCS), perché la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo. In realtà, affermando che la libreria dei thread *pianifica* l'esecuzione dei thread a livello utente associandoli agli LWP liberi, non si intende che il thread sia in esecuzione su una CPU; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di thread del kernel su un processore fisico. Per determinare quale thread a livello kernel debba essere eseguito da una CPU, il kernel esamina i thread di tutto il sistema; si parla allora di **ambito della competizione allargato al sistema** (*system-contention scope*, SCS). Quindi, nel caso di SCS, tutti i thread del sistema competono per l'uso della CPU. I sistemi caratterizzati dal modello da uno a uno (quali Windows XP, Solaris 9 e Linux) pianificano i thread unicamente sulla base di SCS.

Se l'ambito della competizione è ristretto al processo, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta. Le priorità dei thread a livello utente sono stabilite dal programmatore, e la libreria dei thread non le modifica; alcune librerie danno facoltà al programmatore di cambiare la priorità di un thread. Si noti che quando l'ambito della competizione è ristretto al processo si è soliti applicare la prelazione al thread in esecuzione, a vantaggio di thread con priorità più alta; tuttavia, se i thread sono dotati della medesima priorità, non vi è garanzia sulla ripartizione del tempo (Paragrafo 5.3.4).

### 5.4.2 Scheduling di Pthread

La generazione dei thread con POSIX Pthreads è stata introdotta nel Paragrafo 4.3.1, insieme a un programma esemplificativo. Ci accingiamo ora a esaminare la API Pthread del POSIX, che consente di specificare sia PCS che SCS per la generazione dei thread. Per specificare l'ambito della contesa Pthreads usa i valori seguenti:

- ◆ PTHREAD\_SCOPE\_PROCESS pianifica i thread con lo scheduling PCS
- ◆ PTHREAD\_SCOPE\_SYSTEM pianifica i thread tramite lo scheduling SCS

Nei sistemi che si avvalgono del modello da molti a molti (Paragrafo 4.2.3), la politica PTHREAD\_SCOPE\_PROCESS pianifica i thread a livello utente sugli LWP disponibili. Il numero di LWP viene stabilito dalla libreria dei thread, che in qualche caso si serve delle attivazioni dello scheduler (4.4.6). La seconda politica, PTHREAD\_SCOPE\_SYSTEM, crea, in corrispondenza di ciascun thread a livello utente, un LWP a esso vincolato, realizzando così una corrispondenza secondo il modello da molti a uno (Paragrafo 4.2.2).

Lo IPC di Pthread offre due funzioni per appurare e impostare l'ambito della contesa:

- ◆ `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- ◆ `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

Il primo parametro per entrambe le funzioni è un puntatore agli attributi del thread. Il secondo parametro della funzione `pthread_attr_setscope()` riceve uno dei valori PTHREAD\_SCOPE\_SYSTEM o PTHREAD\_SCOPE\_PROCESS, che stabiliscono l'ambito della contesa. Qualora si verifichi un errore, ambedue le funzioni restituiscono valori non nulli.

Nella Figura 5.8 presentiamo un programma Pthread che determina l'ambito della contesa in vigore e lo imposta a PTHREAD\_SCOPE\_PROCESS; quindi crea cinque thread distinti, che andranno in esecuzione secondo il modello di scheduling SCS. Si noti che, nel caso di alcuni sistemi, sono possibili solo determinati valori per l'area della disputa. I sistemi Linux e Mac OS X, per esempio, consentono soltanto PTHREAD\_SCOPE\_SYSTEM.

## 5.5 Scheduling per sistemi multiprocessore

Fin qui la trattazione ha riguardato i problemi inerenti lo scheduling della CPU in un sistema a processore singolo; se sono disponibili più unità d'elaborazione, anche il problema dello scheduling è proporzionalmente più complesso. Si sono sperimentate diverse possibilità e, come s'è visto nella trattazione dello scheduling di una sola CPU, "la soluzione migliore" non esiste. Nel seguito si analizzano brevemente alcuni argomenti attinenti lo scheduling per sistemi multiprocessore. Si considerano i sistemi in cui le unità d'elaborazione sono, in relazione alle loro funzioni, identiche (**sistemi omogenei**): si può usare qualunque unità d'elaborazione disponibile per eseguire qualsiasi processo presente nella coda. (Ma anche i

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* ottiene gli attributi di default */
    pthread_attr_init(&attr);

    /* per prima cosa appura l'ambito della contesa */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Impossibile appurare l'ambito della contesa\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Valore d'ambito della contesa non
                        ammesso.\n");
    }

    /* imposta l'algoritmo di scheduling a PCS o SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* genera i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* adesso aspetta la terminazione di tutti i thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* ogni thread inizia l'esecuzione da questa funzione */
void *runner(void *param)
{
    /* fai qualcosa ... */

    pthread_exit(0);
}

```

**Figura 5.8** API Pthread per lo scheduling.

multiprocessori omogenei, talvolta, possono incorrere in limitazioni. Si consideri un sistema con un dispositivo di I/O collegato a un processore mediante un bus privato; ogni processo che intenda avvalersi di tale dispositivo dovrà essere pianificato per l'esecuzione su quel processore.)

### 5.5.1 Soluzioni di scheduling per multiprocessori

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell'I/O e le altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice dell'utente. Si tratta della **multielaborazione asimmetrica**, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema.

Quando invece ciascun processore provvede al proprio scheduling, si parla di **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP). In questo caso i processi pronti per l'esecuzione sono situati tutti in una coda comune, oppure vi è un'apposita coda per ogni processore. In entrambi i casi, lo scheduler di ciascun processore esamina la coda appropriata, da cui seleziona un processo da eseguire. Come vedremo nel Capitolo 6, l'accesso concorrente di più processori a una struttura dati comune rende delicata la programmazione dello scheduler, al fine di evitare che due processori scelgano il medesimo processo o che un processo in coda non vada perso. La SMP è messa a disposizione da quasi tutti i sistemi operativi moderni, quali Windows XP, Windows 2000, Solaris, Linux e Mac OS X.

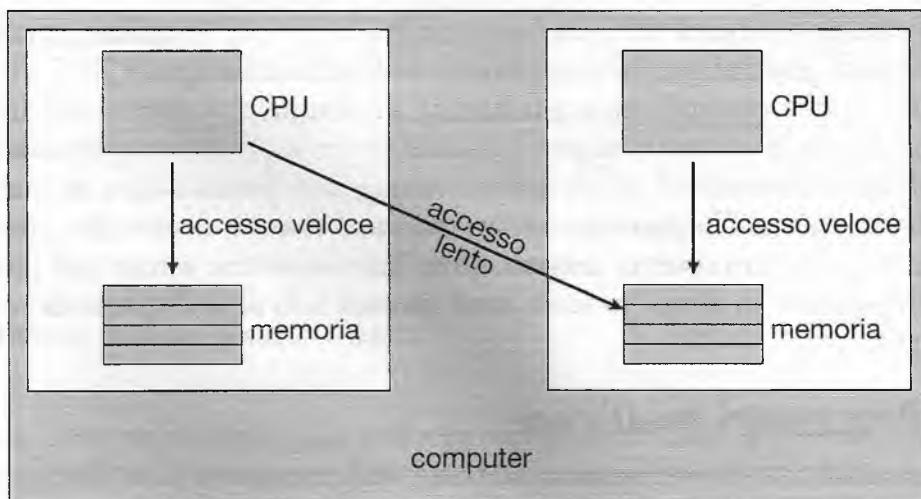
Nei paragrafi successivi discuteremo le questioni concernenti la SMP.

### 5.5.2 Predilezione per il processore

Si consideri che cosa accade alla memoria cache dopo che un processo sia stato eseguito da uno specifico processore: i dati che il processore ha trattato da ultimo permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la memoria cache. Ecco allora che cosa succede se un processo si sposta su un altro processore: i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita. A causa degli alti costi di svuotamento e riempimento della cache, molti sistemi SMP tentano di impedire il passaggio di processi da un processore all'altro, mirando a mantenere un processo sullo stesso processore che lo sta eseguendo. Si parla di **predilezione per il processore** (*processor affinity*), intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione.

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di **predilezione debole** (*soft affinity*). In questo caso è possibile che un processo migri da un processore all'altro. Alcuni sistemi, per esempio Linux, dispongono di chiamate di sistema con cui specificare che un processo non debba cambiare processore; in tal modo, si realizza la **predilezione forte** (*hard affinity*). Solaris permette che i processi siano assegnati a un determinato insieme di processori, definendo quali processi possono essere eseguiti da una data CPU. Solaris permette anche la predilezione debole.

L'architettura della memoria principale di un sistema può influenzare le questioni relative alla predilezione. La Figura 5.9 mostra un'architettura con accesso non uniforme alla memoria (NUMA) in cui la CPU ha un accesso più rapido ad alcune zone di memoria rispetto ad altre. Di solito una situazione di questo tipo si ha in sistemi dove sono presenti diverse schede, ognuna con CPU e memoria proprie. Le CPU su una scheda possono accedere alla memoria sulla stessa scheda con meno ritardo rispetto a quella su schede diverse. Se lo sche-



**Figura 5.9** NUMA e lo scheduling della CPU.

duler della CPU di un sistema operativo e gli algoritmi di allocazione della memoria lavorano insieme, allora un processo con predilezione per una determinata CPU può essere allocato nella memoria residente sulla stessa scheda in cui è montata la CPU. Questo esempio mostra anche come i sistemi operativi non siano definiti e implementati in maniera tanto nittida quanto è descritto nei libri di testo. Le linee di demarcazione tra le componenti di un sistema operativo, lungi dall'essere nette, sono sfumate; opportuni algoritmi instaurano poi connessioni fra le varie parti allo scopo di ottimizzare le prestazioni e l'affidabilità.

### 5.5.3 Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutti i processori per sfruttare appieno i vantaggi della multielaborazione. Se ciò non avviene, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati con una coda di processi in attesa. Il bilanciamento del carico tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema SMP. Bisogna notare che è necessario, di norma, solo nei sistemi in cui ciascun processore detiene una coda privata di processi passibili di esecuzione. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è sovente superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune eseguibile dei processi. Va ricordato, tuttavia, che in quasi tutti i sistemi operativi attuali predisposti per la SMP, ogni processore è effettivamente dotato di una propria coda di processi eseguibili.

Il bilanciamento del carico può seguire due approcci: la **migrazione guidata** (*push migration*) e la **migrazione spontanea** (*pull migration*). La prima prevede che un processo apposito controlli periodicamente il carico di ogni processore: nell'ipotesi di una sproporzione, riporterà il carico in equilibrio, spostando i processi dal processore saturo ad altri più liberi, o inattivi. La migrazione spontanea, invece, si ha quando un processore inattivo sottrae ad un processore sovraccarico un processo in attesa. I due tipi di migrazione non sono mutuamente esclusivi, e trovano spesso applicazione contemporanea nei sistemi con bilanciamento del carico. Lo scheduler Linux, per esempio, che vedremo nel Paragrafo 5.6.3, e lo scheduler ULE dei sistemi FreeBSD, si avvalgono di entrambe le tecniche. Linux esegue il proprio algoritmo di bilanciamento del carico a intervalli di 200 millisecondi (migrazione guidata), e ogniqualvolta si svuoti la coda di esecuzione di un processore (migrazione spontanea).

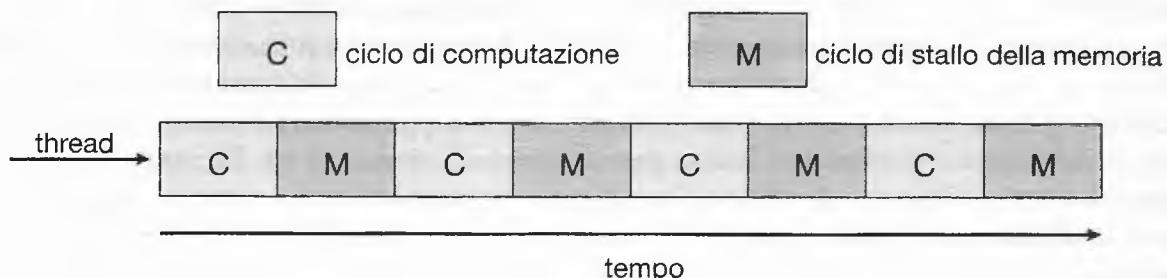
Una singolare proprietà del bilanciamento del carico è di annullare, spesso, il vantaggio derivante dalla predilezione del processore, analizzata nel Paragrafo 5.5.2. Il vantaggio di poter eseguire un processo dall'inizio alla fine sul medesimo processore è che in tal modo la memoria cache contiene i dati necessari per quel processo. Con la migrazione dei processi necessaria al bilanciamento del carico, questo vantaggio si perde. Anche in questo frangente, come di consueto nell'ingegneria dei sistemi, non vi sono dogmi che prescrivano una strategia ottima: in alcuni sistemi, un processore inattivo sottrae sempre un processo da un processore impegnato; in altri, i processi sono spostati solo se la disparità del carico supera una data soglia.

### 5.5.4 Processori multicore

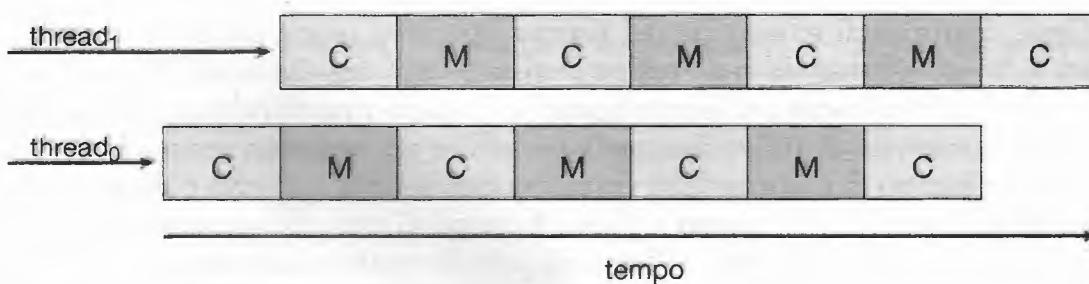
Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la tendenza recente nel progetto di processori è di inserire più unità di calcolo in un unico chip fisico, dando origine a un **processore multicore**. Ogni unità di calcolo ha i registri che le servono per conservare informazioni sul suo stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling. Proviamo a vedere che cosa può succedere. Le ricerche hanno permesso di scoprire che quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come **stallo della memoria**, può verificarsi per varie ragioni, come ad esempio la mancanza dei dati richiesti nella cache. La Figura 5.10 mostra uno stallo della memoria. In questo scenario, il processore può trascorrere fino al 5 per cento del suo tempo attendendo che i dati siano disponibili in memoria. Per rimediare a questa situazione, molti dei progetti hardware recenti implementano delle unità di calcolo multithread in cui due o più thread hardware sono assegnati a una singola unità di calcolo. In questo modo, se un thread è in situazione di stallo in attesa della memoria, l'unità di calcolo può passare il controllo a un altro thread. La Figura 5.11 mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono interfoliate (*interleaved*). Dal punto di vista del sistema operativo, ogni thread hardware appare come un processore logico in grado di eseguire un thread software. In un sistema a due thread con due unità di calcolo il sistema operativo vede dunque quattro processori logici. La CPU UltraSPARC T1 monta otto unità di calcolo per singolo chip, e quattro thread hardware per ogni unità di calcolo; dalla prospettiva del sistema operativo si vedono 32 processori logici.

In generale, ci sono due modi per rendere un processore multithread: attraverso il **multithreading grezzo** (*coarse-grained*) o il **multithreading fine** (*fine-grained*). Nel multithreading grezzo un thread resta in esecuzione su un processore fino al verificarsi di un evento a



**Figura 5.10** Stallo della memoria.



**Figura 5.11** Sistema multicore e multithread.

lunga latenza, come ad esempio uno stallo di memoria. A causa dell'attesa introdotta dall'evento a lunga latenza, il processore deve passare a un altro thread e iniziare a eseguirlo. Tuttavia, il costo per cambiare il thread in esecuzione è alto, perché occorre ripulire la pipeline delle istruzioni prima che il nuovo thread possa iniziare a essere eseguito sull'unità di calcolo. Quando il nuovo thread è in esecuzione inizia a riempire la pipeline con le sue istruzioni. Il multithreading fine (anche detto *multithreading interfogliato*) passa da un thread a un altro con un livello molto più fine di granularità (tipicamente al termine di un ciclo di istruzione). Tuttavia, il progetto di sistemi a multithreading fine include una logica dedicata al cambio di thread. Ne risulta così che il costo del passaggio da un thread a un altro è piuttosto basso.

Va osservato che un processore multithread e multicore richiede due diversi livelli di scheduling. A un primo livello vi sono le decisioni di scheduling che devono essere intraprese dal sistema operativo per stabilire quale thread software mandare in esecuzione su ciascun thread hardware (processore logico). Per realizzare questo tipo di scheduling il processore può scegliere qualunque algoritmo, ad esempio quelli descritti nel Paragrafo 5.3. Un secondo livello di scheduling specifica come ogni unità di calcolo decida quale thread hardware eseguire. Ci sono diverse strategie che si possono adottare in questa situazione. Il processore UltraSPARC T1, menzionato poche righe fa, usa un semplice algoritmo circolare (round-robin) per lo scheduling dei quattro thread hardware su ogni unità di calcolo. Un altro esempio ci è dato da Intel Itanium, una CPU dual-core con due thread hardware per ogni unità di calcolo. Un valore dinamico di *urgency*, compreso tra 0 e 7, dove 0 rappresenta l'urgenza minore e 7 la più alta, viene assegnato a ogni thread hardware del processore. L'Itanium identifica cinque diversi eventi che possono far scattare un cambio di thread. Quando uno di questi eventi si verifica, la logica preposta al cambio di thread confronta l'urgenza dei thread che dovrebbero scambiarsi e sceglie di mandare in esecuzione sull'unità di calcolo il thread con il valore più alto.

### 5.5.5 Virtualizzazione e scheduling

Un sistema dotato di virtualizzazione, anche se a singola CPU, agisce spesso come un sistema multiprocessore. La virtualizzazione software offre una o più CPU virtuali a ogni macchina virtuale in esecuzione sul sistema, e quindi pianifica l'utilizzo della CPU fisica condivisa dalle macchine virtuali. Le sostanziali differenze tra le varie tecnologie di virtualizzazione rendono difficile sintetizzare gli effetti della virtualizzazione sullo scheduling (si veda il Paragrafo 2.8). In generale, comunque, la maggior parte degli ambienti di virtualizzazione ha un sistema operativo ospitante e diversi sistemi ospiti. Il sistema operativo ospitante crea e gestisce le macchine virtuali, e ogni macchina virtuale ha un sistema operativo ospite installato con applicazioni in esecuzione su di esso. Ogni sistema operativo ospite può essere messo a punto per usi specifici, per particolari applicazioni e utenti e anche per il tempo ripartito (*time sharing*) o le operazioni real-time.

Ogni algoritmo di scheduling del sistema operativo ospite che fa assunzioni sulla quantità di lavoro effettuabile in un tempo prefissato verrà negativamente influenzato dalla virtualizzazione. Si consideri un sistema operativo a tempo ripartito che prova a suddividere il tempo in intervalli di 100 millisecondi, per offrire agli utenti un tempo di risposta ragionevole. All'interno di una macchina virtuale questo sistema operativo è alla mercé del sistema di virtualizzazione per quanto riguarda il tempo di CPU che gli viene assegnato. Un intervallo di tempo fissato in 100 millisecondi può diventare nella realtà molto più lungo dei 100 millisecondi di tempo di CPU virtuale. A seconda di quanto è occupato il sistema, i 100 millisecondi possono anche diventare un secondo di tempo reale o più, con il risultato che il tempo di risposta per gli utenti che lavorano sulla macchina virtuale sarà insoddisfacente. Gli effetti su un sistema real-time, poi, sarebbero catastrofici.

Il risultato finale di questi livelli di scheduling è che i singoli sistemi operativi virtualizzati sfruttano a loro insaputa solo una parte dei cicli di CPU disponibili, mentre effettuano lo scheduling come se sfruttassero tutti i cicli fisicamente disponibili. Tipicamente, l'orologio all'interno di una macchina virtuale fornisce valori sbagliati, perché i contatori impiegano più tempo a scattare di quanto farebbero su una CPU dedicata. La virtualizzazione può quindi vanificare i benefici di un buon algoritmo di scheduling del sistema operativo ospitato sulla macchina virtuale.

## 5.6 Esempi di sistemi operativi

---

Procediamo ora nella descrizione dei criteri di scheduling per i sistemi operativi Solaris, Windows XP e Linux. Sarà bene tenere presente che, per Solaris e Linux, lo scheduling in questione è relativo ai thread a livello kernel. Si rammenti inoltre che Linux non distingue tra processi e thread; di conseguenza, relativamente allo scheduler di Linux, useremo il termine *task (compito)*.

### 5.6.1 Esempio: scheduling di Solaris

Solaris utilizza uno scheduling dei thread basato sulle priorità in cui ogni thread appartiene a una delle sei seguenti classi.

1. Tempo ripartito (TS).
2. Interattivo (IA).
3. Real-time (RT).
4. Sistema (SYS).
5. Ripartizione equa (FSS, per *fair share*).
6. Priorità fissa (FP).

All'interno di ciascuna classe vi sono priorità e algoritmi di scheduling differenti.

La classe di scheduling predefinita per i processi è quella a tempo ripartito. È basata su un criterio di scheduling che modifica dinamicamente le priorità, assegnando porzioni di tempo variabili, grazie a una coda multipla con retroazione. Per default, tra le priorità e le frazioni di tempo sussiste una relazione inversa: più alta è la priorità, minore la frazione di tempo associata; più bassa è la priorità, maggiore sarà la frazione di tempo. Di solito, i processi interattivi hanno priorità alta, mentre i processi con prevalenza d'elaborazione hanno priorità bassa. Questo criterio di scheduling offre un buon tempo di risposta per i processi interattivi e una buona produttività per i processi con prevalenza d'elaborazione. La classe

priorità	quanto di tempo	quanto di tempo esaurito	ripresa dell'attività
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

**Figura 5.12** Tabella di dispatch di Solaris per i thread interattivi e a tempo ripartito.

interattiva e quella a tempo ripartito utilizzano gli stessi criteri di scheduling, ma la prima privilegia le applicazioni dotate di interfacce a finestre, a cui attribuisce priorità più elevate per ottenere prestazioni migliori.

La Figura 5.12 mostra la tabella di dispatch per lo scheduling dei thread interattivi e a tempo ripartito. Queste due classi contemplano 60 livelli di priorità; ne elenchiamo solo alcuni. La tabella di dispatch della Figura 5.12 contiene i seguenti campi.

- ◆ **Priorità.** È la priorità dipendente dalle classi, nel caso di quelle interattiva e a tempo ripartito. Il valore cresce al crescere della priorità.
- ◆ **Quanto di tempo.** È il quanto di tempo della relativa priorità. Come si può notare, priorità e frazioni di tempo sono inversamente correlate: alla priorità 0, infatti, spetta il quanto di tempo più lungo (200 millisecondi), mentre alla priorità 59, cioè la più alta, corrisponde il quanto minimo (20 millisecondi).
- ◆ **Quanto di tempo esaurito.** È la nuova priorità dei thread che abbiano consumato l'intero quanto di tempo loro assegnato senza sospendersi. Tali thread sono considerati a prevalenza d'elaborazione; le loro priorità, come evidenziato dalla tabella, subiscono una diminuzione.
- ◆ **Ripresa dell'attività.** È la priorità di un thread che ritorni in attività dopo un periodo di attesa (dell'I/O, per esempio). Come si può vedere nella tabella, quando è disponibile l'I/O per un thread in attesa, la priorità del thread assume un valore compreso tra 50 e 59. Si implementa così il criterio di scheduling che consiste nel fornire risposte sollecite ai processi interattivi.

La classe dei thread real-time ha la priorità maggiore. Ciò fa sì che i processi real-time abbiano la garanzia di ottenere una risposta dal sistema entro limiti di tempo prefissati. Un

processo real-time sarà eseguito prima di un processo appartenente a qualsiasi altra classe. In generale, tuttavia, pochi processi appartengono alla classe real-time.

Solaris sfrutta la classe sistema per eseguire i processi del kernel, quali lo scheduler e il demone per la paginazione. La priorità di un processo del sistema, una volta fissata, non cambia. La classe sistema è riservata all'uso da parte del kernel (i processi utenti eseguiti in modalità di sistema non appartengono alla classe sistema).

Le classi a priorità fissa e a ripartizione equa sono state introdotte in Solaris 9. I thread appartenenti alla classe a priorità fissa hanno lo stesso livello di priorità di quelli della classe a tempo ripartito, ma le loro priorità non vengono modificate dinamicamente. Per la classe a ripartizione equa le decisioni di scheduling vengono prese sulla base delle *quote* (*shares*) di CPU, e non sulla base delle proprietà. Le quote di CPU sono assegnate a un insieme di processi, chiamato **progetto**, e indicano in che misura il progetto ha diritto all'uso delle risorse disponibili.

Ogni classe di scheduling include una scala di priorità. Tuttavia, lo scheduler converte le priorità specifiche della classe in priorità globali e sceglie per l'esecuzione il thread con la priorità globale più elevata. La CPU esegue il thread prescelto finché (1) si blocca, (2) esaurisce la propria frazione di tempo, o (3) è soggetto a prelazione da un thread con priorità più alta. Se vi sono più thread con la stessa priorità, lo scheduler utilizza una coda circolare RR. La Figura 5.13 mostra le relazioni fra le sei classi di scheduling, e quali sono le priorità globali a loro assegnate. Va notato che il kernel utilizza 10 thread per servire le interruzioni. Questi thread non appartengono ad alcuna classe e sono eseguiti con massima priorità (160-169). Come già ricordato, tradizionalmente Solaris utilizzava il modello da molti a molti (Paragrafo 4.2.3), ma a partire da Solaris 9 è passato al modello da uno a uno (Paragrafo 4.2.2).

## 5.6.2 Esempio: scheduling di Windows XP

Il sistema operativo Windows XP compie lo scheduling dei thread servendosi di un algoritmo basato su priorità e prelazione. Lo scheduler assicura che si eseguano sempre i thread a più alta priorità. La porzione del kernel che si occupa dello scheduling si chiama *dispatcher*. Una volta selezionato dal *dispatcher*, un thread viene eseguito finché non sia sottoposto a prelazione da un altro thread a priorità più alta oppure termini, esaurisca il suo quanto di tempo o esegua una chiamata di sistema bloccante, per esempio un'operazione di I/O. Se un thread d'elaborazione in tempo reale, ad alta priorità, entra nella coda dei processi pronti per l'esecuzione mentre è in esecuzione un thread a bassa priorità, quest'ultimo viene sottoposto a prelazione. Ciò realizza un accesso preferenziale alla CPU per i thread d'elaborazione in tempo reale che ne hanno necessità.

Per determinare l'ordine d'esecuzione dei thread il *dispatcher* impiega uno schema di priorità a 32 livelli. Le priorità sono suddivise in due classi: la classe **variable** raccoglie i thread con priorità da 1 a 15, mentre la classe **real-time** raccoglie i thread con priorità tra 16 e 31 (esiste anche un thread, per la gestione della memoria, che si esegue con priorità 0). Il *dispatcher* adopera una coda per ciascuna priorità di scheduling e percorre l'insieme delle code da quella a priorità più alta a quella a priorità più bassa, finché trova un thread pronto per l'esecuzione. In assenza di tali thread, il *dispatcher* fa eseguire un thread speciale detto **idle thread**.

C'è una relazione tra le priorità numeriche del kernel del sistema operativo Windows XP e quelle dell'API Win32. Secondo l'API Win32 un processo può appartenere a diverse classi di priorità, tra cui le seguenti:

- ◆ REALTIME\_PRIORITY\_CLASS
- ◆ HIGH\_PRIORITY\_CLASS
- ◆ ABOVE\_NORMAL\_PRIORITY\_CLASS

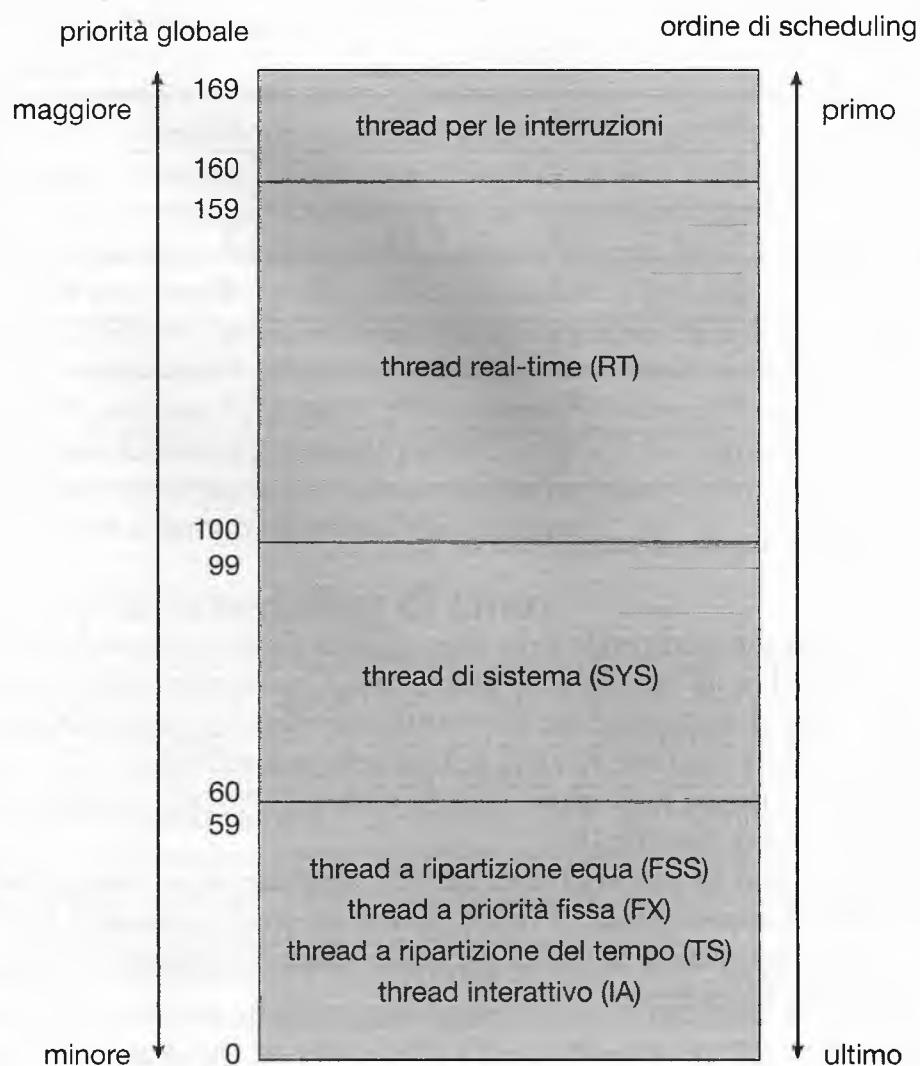


Figura 5.13 Scheduling di Solaris.

- ◆ NORMAL\_PRIORITY\_CLASS
- ◆ BELOW\_NORMAL\_PRIORITY\_CLASS
- ◆ IDLE\_PRIORITY\_CLASS.

Le priorità di ciascuna classe eccetto REALTIME\_PRIORITY\_CLASS sono priorità di classe variabile, quindi la priorità di un thread appartenente a queste classi può cambiare.

Ciascuna di queste classi prevede delle priorità relative, i cui valori comprendono i seguenti:

- ◆ TIME\_CRITICAL
- ◆ HIGHEST
- ◆ ABOVE\_NORMAL
- ◆ NORMAL
- ◆ BELOW\_NORMAL
- ◆ LOWEST
- ◆ IDLE.

	realtime	high	above_normal	normal	below_normal	idle_priority
time_critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above_normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below_normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

**Figura 5.14** Priorità nel sistema operativo Windows XP.

La priorità di ciascun thread dipende dalla priorità della classe cui appartiene e dalla priorità relativa che il thread ha all'interno della stessa classe. Questa relazione è rappresentata nella Figura 5.14. I valori di ciascuna classe di priorità sono riportati nella prima riga in alto. La prima colonna a sinistra contiene i valori delle diverse priorità relative. Per esempio, se la priorità relativa di un thread nella classe ABOVE\_NORMAL\_PRIORITY\_CLASS è NORMAL, la priorità numerica di quel thread è 10.

Inoltre, ogni thread ha una priorità di base che rappresenta un valore nell'intervallo di priorità della classe di appartenenza. Il valore predefinito per la priorità di base in una classe è quello della priorità relativa NORMAL per quella classe. Le priorità di base per ciascuna classe di priorità sono le seguenti:

- ◆ REALTIME\_PRIORITY\_CLASS–24
- ◆ HIGH\_PRIORITY\_CLASS–13
- ◆ ABOVE\_NORMAL\_PRIORITY\_CLASS–10
- ◆ NORMAL\_PRIORITY\_CLASS–8
- ◆ BELOW\_NORMAL\_PRIORITY\_CLASS–6
- ◆ IDLE\_PRIORITY\_CLASS–4.

Di solito, i processi appartengono alla classe NORMAL\_PRIORITY\_CLASS, sempre che il processo genitore non appartenga alla classe IDLE\_PRIORITY\_CLASS, o sia stata specificata un'altra classe alla creazione del processo. Di solito la priorità iniziale di un thread è la priorità di base del processo a cui il thread appartiene.

Quando il quanto di tempo di un thread si esaurisce, il thread viene interrotto e se il thread fa parte della classe a priorità variabile, la sua priorità viene ridotta. Tuttavia, la priorità non si abbassa mai oltre la priorità di base. L'abbassamento della priorità tende a limitare l'uso della CPU da parte dei thread con prevalenza d'elaborazione. Se un thread a priorità variabile è rilasciato da un'operazione d'attesa, il *dispatcher* aumenta la sua priorità. L'entità di questo aumento dipende dal tipo d'evento che il thread attendeva: un thread che attendeva dati dalla tastiera riceve un forte aumento di priorità, uno che attendeva operazioni relative a un disco riceve un aumento più moderato. Questa strategia mira a fornire buoni tempi di risposta per i thread interattivi, con interfacce basate su mouse e finestre; permette

inoltre ai thread con prevalenza di I/O di tenere occupati i dispositivi di I/O, e rende nel contempo possibile l'utilizzo con esecuzione in sottofondo dei cicli di CPU inutilizzati da parte dei thread con prevalenza d'elaborazione. Questa strategia si segue in molti sistemi operativi a tempo ripartito d'elaborazione, compreso UNIX. Inoltre, per migliorare il tempo di risposta, la finestra attraverso cui l'utente sta interagendo ottiene un incremento di priorità.

Quando un utente richiede l'esecuzione di un programma interattivo, il sistema deve fornire al relativo processo prestazioni particolarmente elevate. Per questa ragione, il sistema Windows XP segue una regola specifica di scheduling per i processi della classe NORMAL\_PRIORITY\_CLASS. Il sistema operativo Windows XP distingue tra il *processo in primo piano*, correntemente selezionato sullo schermo e i *processi in sottofondo*, che non sono attualmente selezionati. Quando un processo passa in primo piano, Windows XP aumenta il suo quanto di tempo di un certo fattore, tipicamente pari a 3, ciò fa sì che il processo in primo piano possa continuare la propria esecuzione per un tempo tre volte più lungo, prima che si abbia una prelazione dovuta al tempo ripartito d'elaborazione.

### 5.6.3 Esempio: scheduling di Linux

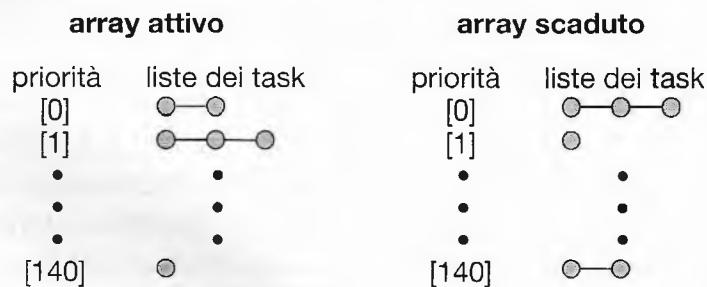
Prima della versione 2.5, il kernel di Linux impiegava, per lo scheduling, una variante dell'algoritmo tradizionale di UNIX. Quest'ultimo, però, comporta due problemi: non fornisce strumenti adeguati per i sistemi SMP, e risponde male al crescere dell'ordine di grandezza del numero dei task nel sistema. A partire dalla versione 2.5, il kernel offre un algoritmo di scheduling che gira in tempo costante – in simboli,  $O(1)$  – a prescindere dal numero di task nel sistema. Il nuovo scheduler è anche migliorato nei confronti della SMP: permette infatti la gestione della predilezione per il processore, rende possibile il bilanciamento del carico, e offre inoltre strumenti volti a garantire equità ai task interattivi.

Lo scheduler di Linux ricorre a un algoritmo di scheduling con prelazione, basato sulle priorità, con due gamme di priorità separate: un intervallo **real-time** che va da 0 a 99 e un intervallo detto **nice** compreso tra 100 e 140. Queste due gamme sono poi tradotte all'interno di una scala globale di priorità, in cui i valori numericamente più bassi rappresentano le priorità più alte.

A differenza degli scheduler di molti altri sistemi, come Solaris (Paragrafo 5.6.1) e Windows XP (Paragrafo 5.6.2), Linux assegna ai task con priorità più alta porzioni di tempo più cospicue e a quelle dalla priorità più bassa quanti di tempo più brevi. La relazione tra priorità e lunghezza del quanto di tempo è mostrata dalla Figura 5.15.



Figura 5.15 Relazione fra le priorità e la lunghezza del quanto di tempo.



**Figura 5.16** Liste dei task indicizzate sulla base della priorità.

Un task pronto per l'esecuzione è considerato eseguibile dalla CPU se non ha ancora consumato per intero il proprio quanto di tempo. Allorché esaurisce tale quanto, il task è considerato scaduto, e non può più essere posto in esecuzione finché tutti gli altri task non abbiano consumato la propria porzione di tempo. Il kernel elenca i task pronti per l'esecuzione in una struttura dati detta *runqueue* (*coda di esecuzione*). Per garantire la possibilità di SMP, ciascun processore mantiene la propria coda di esecuzione e opera una pianificazione autonoma. Ogni coda di esecuzione contiene due array di priorità: **attivo** e **scaduto**. Il primo contiene tutti i task che hanno ancora tempo da sfruttare, mentre il secondo elenca i task scaduti. Entrambi gli array contengono una lista di task, ordinati progressivamente secondo la priorità (Figura 5.16). Lo scheduler sceglie il task con la priorità più alta dall'array attivo affinché sia eseguito dalla CPU. Sulle macchine dotate di più processori, ogni processore mantiene una propria coda di esecuzione, e seleziona il task successivo da eseguire. Una volta che tutti i task nell'array attivo abbiano consumato il loro quanto di tempo (cioè, quando l'array attivo è vuoto), gli array delle priorità si scambiano i ruoli: quello scaduto diventa quello attivo e viceversa.

Linux implementa lo scheduling real-time come definito dallo standard POSIX.1b; si veda il Paragrafo 5.4.2 per un'analisi approfondita. I task real-time ricevono priorità statiche, mentre gli altri task hanno priorità dinamiche, basate sul loro valore *nice*, cui si aggiunge o si sottrae il valore 5. Il grado di interattività di un task determina se il valore 5 sarà sommato al valore nice o sottratto da esso. Tale grado di interattività è determinato dal tempo trascorso dal task in attesa prima che il suo I/O sia disponibile. I task maggiormente interattivi hanno, in genere, tempi di attesa più lunghi; poiché lo scheduler dà loro preferenza, la probabilità che essi ottengano un bonus vicino a -5 è alta. Questi adattamenti hanno l'effetto di attribuire priorità più alte per i task interattivi. Al contrario, i task con tempi di attesa più brevi sono spesso a prevalenza di elaborazione, e dunque vedranno diminuire la loro priorità.

La priorità dinamica di un task è ricalcolata nel momento in cui abbia esaurito la propria porzione di tempo e debba passare dall'array attivo a quello scaduto. Così, quando i due array si scambiano i ruoli, tutti i task del nuovo array attivo avranno ricevuto nuove priorità, con i relativi quanti di tempo.

## 5.7 Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della CPU per un sistema particolare. Come abbiamo visto nel Paragrafo 5.3, esistono molti algoritmi di scheduling, ciascuno dotato dei propri parametri; quindi, la scelta di un algoritmo può essere abbastanza difficile.

Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo. Nel Paragrafo 5.2 si spiega che i criteri si definiscono spesso nei termini dell'utilizzo della CPU, del tempo di risposta o della produttività. Per scegliere un algoritmo occorre innanzitutto stabilire l'importanza relativa di queste misure. Tra i criteri suggeriti si possono inserire diverse misure, per esempio le seguenti:

- ◆ rendere massimo l'utilizzo della CPU con il vincolo che il massimo tempo di risposta sia 1 secondo;
- ◆ rendere massima la produttività in modo che il tempo di completamento sia (in media) linearmente proporzionale al tempo d'esecuzione totale.

Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari metodi di valutazione.

### 5.7.1 Modelli deterministici

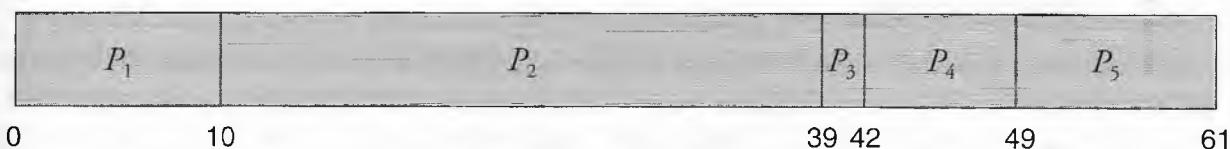
Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della valutazione analitica. La **valutazione analitica**, secondo l'algoritmo dato e il carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro.

La definizione e lo studio di un **modello deterministico** è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

Si supponga, per esempio, di avere il carico di lavoro illustrato di seguito; i cinque processi si presentano al tempo 0, nell'ordine dato, e la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

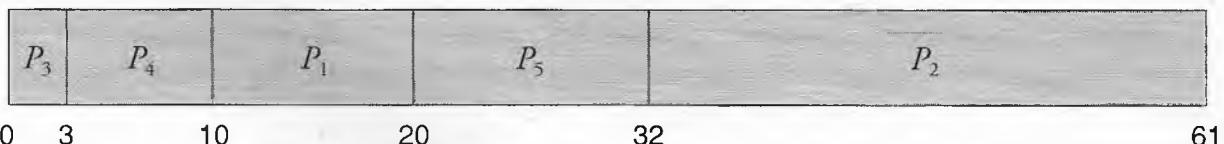
Processo	Durata della sequenza
$P_1$	10
$P_2$	29
$P_3$	3
$P_4$	7
$P_5$	12

Si può stabilire con quale fra gli algoritmi di scheduling FCFS, SJF e RR (quanto di tempo = 10 millisecondi) per questo insieme di processi si ottenga il minimo tempo medio d'attesa. Con l'algoritmo FCFS i processi si eseguono secondo lo schema seguente.



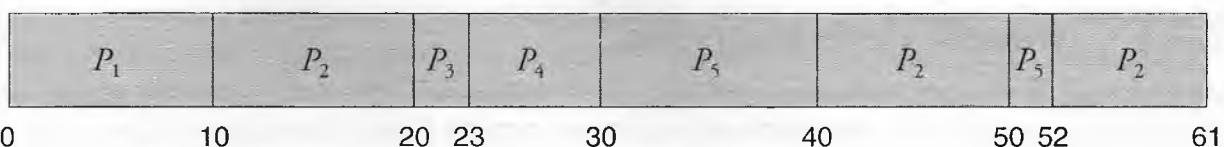
Il tempo d'attesa è di 0 millisecondi per il processo  $P_1$ , di 10 millisecondi per il processo  $P_2$ , di 39 millisecondi per il processo  $P_3$ , di 42 millisecondi per il processo  $P_4$  e di 49 millisecondi per il processo  $P_5$ . Quindi, il tempo d'attesa medio è di  $(0 + 10 + 39 + 42 + 49)/5 = 28$  millisecondi.

Con l'algoritmo SJF senza prelazione i processi si eseguono come segue.



Il tempo d'attesa è di 10 millisecondi per il processo  $P_1$ , di 32 millisecondi per il processo  $P_2$ , di 0 millisecondi per il processo  $P_3$ , di 3 millisecondi per il processo  $P_4$  e di 20 millisecondi per il processo  $P_5$ . Quindi, il tempo d'attesa medio è di  $(10 + 32 + 0 + 3 + 20)/5 = 13$  millisecondi.

Con l'algoritmo RR i processi si eseguono come segue.



Il tempo d'attesa è di 0 millisecondi per il processo  $P_1$ , di 32 millisecondi per il processo  $P_2$ , di 20 millisecondi per il processo  $P_3$ , di 23 millisecondi per il processo  $P_4$  e di 40 millisecondi per il processo  $P_5$ . Quindi, il tempo d'attesa medio è di  $(0 + 32 + 20 + 23 + 40)/5 = 23$  millisecondi.

È importante notare come, *in questo caso*, il criterio SJF fornisca come risultato un tempo medio d'attesa minore della metà del tempo corrispondente ottenuto con lo scheduling FCFS; l'algoritmo RR fornisce un risultato intermedio tra i precedenti.

La definizione e lo studio di un modello deterministico è semplice e rapida; i risultati sono numeri esatti che consentono il confronto tra gli algoritmi. Nondimeno, anche i parametri devono essere numeri esatti e i risultati sono applicabili solo a quei casi. Il suo impiego principale consiste nella descrizione degli algoritmi di scheduling e nella presentazione d'esempi. Nei casi in cui si possono eseguire ripetutamente gli stessi programmi e si possono misurare con precisione i requisiti d'elaborazione dei programmi, i modelli deterministici sono utilizzabili per scegliere un algoritmo di scheduling. Lo studio dei modelli deterministici rispetto a un insieme d'esempi può indicare tendenze che si possono poi analizzare e verificare separatamente. Si può per esempio mostrare che per l'ambiente descritto, vale a dire tutti i processi e i relativi tempi disponibili al tempo 0, con il criterio SJF si ottiene sempre il tempo d'attesa minimo.

## 5.7.2 Reti di code

In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi (e di tempi) da usare nei modelli deterministici. Si possono però determinare le distribuzioni delle sequenze di operazioni della CPU e delle sequenze di operazioni di I/O, poiché si possono misurare e quindi approssimare, o più semplicemente stimare. Si ottiene una formula matematica che indica la probabilità di una determinata sequenza di operazioni della CPU. Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media. Analogamente, è necessario fornire anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare la produttività media, l'utilizzo o il tempo d'attesa medi, e così via, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La CPU è un server con la propria coda dei processi pronti, e il sistema di I/O ha le sue code dei dispositivi. Se sono noti l'andamento degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via. Questo tipo di studio si chiama **analisi delle reti di code** (*queueing-network analysis*).

Si consideri il seguente esempio: sia  $n$  la lunghezza media di una coda, escluso il processo correntemente servito, detti  $W$  il tempo medio d'attesa nella coda e  $\lambda$  l'andamento medio d'arrivo dei nuovi processi nella coda (per esempio, 3 processi al secondo); si prevede che, nel tempo  $W$  durante il quale un processo attende nella coda, raggiungano la coda  $\lambda \times W$  nuovi processi. Se il sistema è stabile, il numero dei processi che lasciano la coda deve essere uguale al numero dei processi che vi arrivano; quindi,

$$n = \lambda \times W$$

Quest'equazione è nota come **formula di Little** ed è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione d'arrivi.

La formula di Little è utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due. Per esempio, sapendo che ogni secondo arrivano 7 processi (in media), e che normalmente nella coda ne sono presenti 14, si può calcolare che il tempo medio d'attesa per ogni processo è di 2 secondi.

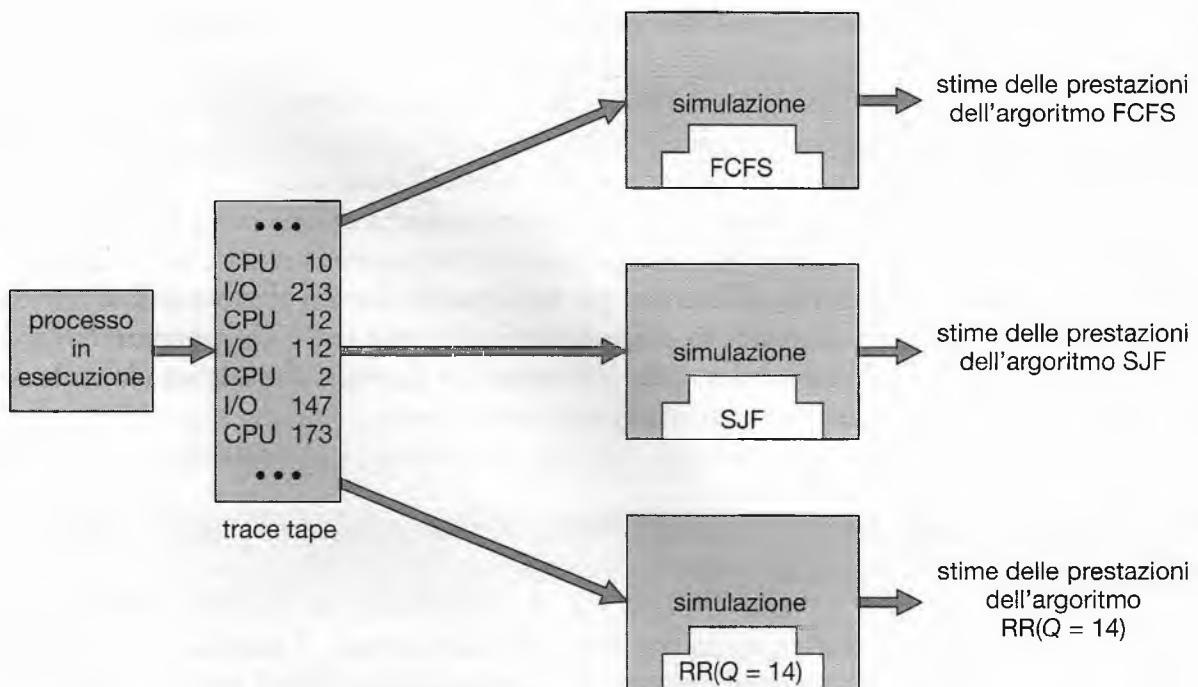
L'analisi delle reti di code può essere utile per il confronto degli algoritmi di scheduling, ma presenta alcuni limiti. Attualmente le classi di algoritmi e distribuzioni trattabili sono piuttosto limitate. Inoltre, poiché può essere difficile lavorare con la matematica di distribuzioni e algoritmi complicati, spesso, affinché siano trattabili matematicamente, si definiscono in modo irrealistico le distribuzioni d'arrivo e servizio. Generalmente è necessario stabilire anche un numero di presupposti indipendenti che possono non essere precisi. Per poter calcolare una risposta, le reti di code spesso si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

### 5.7.3 Simulazioni

Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di **simulazioni**. Le simulazioni implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema. Il simulatore dispone di una variabile che rappresenta un *clock*; con l'aumentare del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

I dati necessari per condurre la simulazione si possono ottenere in vari modi. Il metodo più diffuso impiega un generatore di numeri casuali, programmato per creazione di processi, durata delle sequenze di operazioni della CPU, arrivi, conclusioni e così via; in modo conforme alle rispettive distribuzioni di probabilità, definibili matematicamente (esponenziali, uniformi, di Poisson) oppure in modo empirico. Se la distribuzione deve essere definita in modo empirico, si fanno misure sul sistema reale in esame, e si usano i risultati per definire la distribuzione effettiva degli eventi nel sistema reale.

Tuttavia, una simulazione condotta secondo la distribuzione può non essere precisa, a causa delle relazioni esistenti tra eventi successivi nel sistema reale. La distribuzione relativa alle frequenze, infatti, si limita a indicare quanti eventi di una data categoria si verificano, senza fornire informazioni sul loro ordine. Per rimediare a questo problema si può sottoporre il sistema reale a un controllo continuo, con la registrazione della sequenza degli even-



**Figura 5.17** Valutazione di algoritmi di scheduling della CPU tramite una simulazione.

ti effettivi – in questo modo si ottiene un cosiddetto **trace tape** – (Figura 5.17), che poi si usa per condurre la simulazione. Si tratta di uno strumento eccellente che permette di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.

Poiché spesso richiedono diverse ore del tempo d'elaborazione, le simulazioni possono tuttavia essere molto onerose. Una simulazione dettagliata dà risultati molto precisi, ma richiede anche una gran quantità di tempo, e molto spazio di memoria per la registrazione degli eventi. Inoltre, la progettazione, la codifica e la messa a punto di un simulatore possono essere un compito assai impegnativo.

### 5.7.4 Realizzazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di funzionamento del sistema.

Il problema principale di questo metodo è il suo costo: le spese non sono dovute solo alla codifica dell'algoritmo e alle modifiche da fare al sistema operativo affinché possa gestire l'algoritmo con le sue strutture dati, ma anche alle reazioni degli utenti a fronte di costanti modifiche del sistema operativo. Alla maggior parte degli utenti non interessa la realizzazione di un sistema operativo migliore, ma soltanto eseguire i processi e usare i risultati. Un sistema operativo che si trovi costantemente in fasi di modifica e messa a punto non aiuta gli utenti a svolgere il loro lavoro.

Un'altra difficoltà da affrontare per fare qualsiasi valutazione di un algoritmo è il cambiamento dell'ambiente in cui lo si usa. L'ambiente non cambia solo nel modo consueto, cioè per la scrittura di nuovi programmi e nuovi problemi che si possono riscontrare, ma si modifica anche in seguito alle prestazioni dello scheduler. Se si dà la priorità ai processi brevi, gli utenti possono suddividere i processi più lunghi in gruppi di processi brevi. Se si dà la

priorità ai processi interattivi rispetto ai processi non interattivi, gli utenti possono passare all'uso interattivo.

Per esempio, alcuni ricercatori progettarono un sistema che classificava automaticamente i processi nelle categorie interattiva e non interattiva sulla base della quantità di I/O eseguita dal o verso il terminale. I processi che non leggessero o scrivessero sul terminale per un intero secondo erano classificati come non interattivi, e conseguentemente spostati in una coda a bassa priorità. Un programmatore reagì a questa strategia modificando i suoi programmi di modo che scrivessero sul terminale un carattere a intervalli regolari di meno di un secondo. Il risultato fu che le applicazioni del programmatore ricevettero alta priorità dal sistema, sebbene il loro output fosse del tutto privo di senso.

Gli algoritmi di scheduling più flessibili sono quelli che possono essere tarati dagli amministratori del sistema o dai suoi utenti in modo da adattarsi a una specifica gamma di applicazioni. Per esempio, le macchine impegnate in applicazioni grafiche all'avanguardia avranno necessità di scheduling diverse da quelle di un server web o di un file server. Alcuni sistemi operativi, e in particolare diverse versioni di UNIX, danno all'amministratore la possibilità di calibrare con grande precisione i parametri di scheduling in vista di particolari configurazioni del sistema. Solaris, per esempio, offre il comando `dispadmin` per la modifica dei parametri che regolano le classi di scheduling discusse nel Paragrafo 5.6.1.

Un altro approccio è di usare delle API appropriate per modificare la priorità di processi e thread. Le API Java, POSIX e Windows offrono tali funzionalità. Questa tecnica ha però lo svantaggio che tarare un sistema o un'applicazione per migliorarne le prestazioni spesso non ha lo stesso riscontro in situazioni più generali.

## 5.8 Sommario

Lo scheduling della CPU consiste nella scelta di un processo dalla coda dei processi pronti cui assegnare la CPU. L'effettiva assegnazione della CPU al processo prescelto è eseguita dal dispatcher.

L'algoritmo di scheduling in ordine d'arrivo (FCFS) è il più semplice, ma può far sì che brevi processi attendano processi molto lunghi. Si dimostra che lo scheduling ottimale, che determina il minimo tempo medio d'attesa, è lo scheduling per brevità (SJF). Realizzare lo scheduling SJF è complicato, poiché è difficile prevedere la lunghezza della successiva sequenza di operazioni della CPU. L'algoritmo SJF è un caso particolare dell'algoritmo generale di scheduling per priorità, che si limita ad assegnare la CPU al processo con priorità più elevata. Sia lo scheduling per priorità sia lo scheduling SJF possono condurre a situazioni d'attesa indefinita. L'invecchiamento (*aging*) è una tecnica che si usa per impedire che avvengano tali situazioni.

Lo scheduling circolare (RR) è il più appropriato per un sistema a tempo ripartito: si assegna la CPU al primo processo della coda dei processi pronti per  $q$  unità di tempo (quanto di tempo); dopodiché si ha la prelazione della CPU e si mette il processo in fondo alla coda dei processi pronti. Il problema principale è la scelta della durata del quanto di tempo; se è troppo lungo, lo scheduling RR si riduce a uno scheduling FCFS; se è troppo breve, il carico di scheduling dovuto al tempo dei cambi di contesto diventa eccessivo.

L'algoritmo FCFS è senza prelazione; l'algoritmo RR è con prelazione; gli algoritmi SJF e con priorità possono essere sia con prelazione sia senza prelazione.

Lo scheduling a code multiple permette l'uso di diversi algoritmi per diverse classi di processi. Le più comuni sono la coda dei processi interattivi, eseguiti in primo piano, con

scheduling RR, e la coda per processi a lotti, eseguiti in sottofondo, con scheduling FCFS. Le code multiple con retroazione permettono ai processi di spostarsi da una coda all'altra.

Molti dei sistemi attuali supportano processori multipli permettendo a ciascun processore una pianificazione indipendente. In genere ogni processore mantiene una propria coda di processi pronti (o di thread), tutti disponibili all'esecuzione. Tra gli altri aspetti relativi allo scheduling in sistemi multiprocessore vi sono la predilezione, il bilanciamento del carico e lo scheduling con processori multicore e in sistemi di virtualizzazione.

I sistemi operativi che gestiscono i thread a livello kernel devono occuparsi dello scheduling dei thread, e non di quello dei processi. Tra questi vi sono il Solaris e il Windows XP; entrambi gestiscono lo scheduling dei thread impiegando un algoritmo di scheduling con diritto di prelazione, basato su priorità e che comprende i thread in tempo reale. Anche lo scheduler dei processi di Linux impiega un algoritmo basato su priorità e che prevede la gestione dei processi in tempo reale. Gli algoritmi di scheduling di questi tre sistemi operativi favoriscono generalmente i processi interattivi rispetto ai processi a lotti o con prevalenza d'elaborazione.

La vasta gamma di algoritmi di scheduling esistenti rende imperativa la disponibilità di metodi per la loro selezione. I metodi analitici sfruttano strumenti matematici per valutare le prestazioni degli algoritmi. Le simulazioni determinano le prestazioni eseguendo gli algoritmi in presenza di un insieme "rappresentativo" di processi. Va detto, però, che la simulazione può tutt'al più dare un'approssimazione delle effettive prestazioni dei sistemi. L'unica tecnica affidabile per la valutazione delle prestazioni di un algoritmo di scheduling consiste nell'implementarlo su un vero sistema, e nel misurarne le prestazioni in un contesto reale.

## Esercizi pratici

- 5.1 Un algoritmo di scheduling della CPU stabilisce un ordine per l'esecuzione dei processi pianificati. Dati  $n$  processi da mandare in esecuzione su di un singolo processore, quanti differenti scheduling sono possibili? Date una formula in funzione di  $n$ .
- 5.2 Spiegate la differenza tra lo scheduling con e senza prelazione.
- 5.3 Supponete che i seguenti processi siano pronti per l'esecuzione agli istanti di tempo indicati nella tabella che segue. Nella tabella è anche indicata la durata della sequenza di operazioni per ogni processo. Nel rispondere alle domande seguenti utilizzate uno scheduling senza prelazione e basate le vostre decisioni sulle informazioni che avete a disposizione nel momento in cui la decisione deve essere presa.

Processo	Istante di arrivo	Durata della sequenza
$P_1$	0,0	8
$P_2$	0,4	4
$P_3$	1,0	1

- a. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling FCFS?
- b. Qual è il tempo di completamento medio di questi processi se si utilizza un algoritmo di scheduling SJF?
- c. L'algoritmo SJF dovrebbe, in teoria, migliorare le prestazioni, ma si noti che al tempo 0 la scelta ricadrà sul processo  $P_1$ , perché non si sa ancora che presto arriveranno due processi più piccoli. Calcolate il tempo di completamento medio

ipotizzando che la CPU venga lasciata inattiva per il primo istante di tempo e che successivamente venga utilizzato l'algoritmo SJF. Ricordate che i processi  $P_1$  e  $P_2$  restano in attesa durante il periodo di inattività, e quindi il loro tempo di attesa può aumentare. Questo algoritmo è conosciuto come scheduling a futuro conosciuto.

5.4 Quali vantaggi si hanno nell'avere quanti di tempo di dimensioni differenti a differenti livelli in un sistema di code multiple?

5.5 Molti algoritmi di scheduling della CPU sono parametrici. L'algoritmo RR, ad esempio, richiede un parametro che specifichi l'intervallo di tempo. Le code multiple con retroazione richiedono parametri per specificare il numero di code, l'algoritmo di scheduling da utilizzare per ogni coda, il criterio usato per muovere i processi tra le code, e così via.

Questi algoritmi sono dunque classi di algoritmi (per esempio, la classe di algoritmi RR per tutti gli intervalli di tempo, e così via). Una classe di algoritmi ne può includere un'altra (per esempio, l'algoritmo FCFS è un algoritmo RR con intervallo di tempo infinito). Quale relazione intercorre (se esiste una relazione) tra le seguenti coppie di classi di algoritmi?

- a. Priorità e SJF.
- b. Code mutiple con retroazione e FCFS.
- c. Priorità ed FCFS.
- d. RR e SJF.

5.6 Supponete che un algoritmo di scheduling (a livello dello scheduling della CPU a breve termine) favorisca quei processi che hanno usato meno CPU in un passato recente. Spiegate perché questo algoritmo favorirà programmi con prevalenza di I/O senza bloccare permanentemente i programmi con prevalenza di elaborazione.

5.7 Individuate le differenze tra gli scheduling PCS e SCS.

## Esercizi

5.8 Perché è importante, per lo scheduler, distinguere i programmi con prevalenza di I/O da quelli con prevalenza di elaborazione?

5.9 Considerate come le seguenti coppie di criteri per lo scheduling entrino in conflitto in certe situazioni:

- a. utilizzo della CPU e tempo di risposta;
- b. tempo di completamento medio e tempo di attesa massimo;
- c. utilizzo dei dispositivi di I/O e utilizzo della CPU.

5.10 Considerate la formula della media esponenziale atta a predire la durata della sequenza successiva della CPU. Quali implicazioni scaturiscono dall'assegnazione dei seguenti valori ai parametri usati dall'algoritmo?

- a.  $\alpha = 0$  e  $\tau_0 = 100$  millisecondi
- b.  $\alpha = 0,99$  e  $\tau_0 = 10$  millisecondi

- 5.11 Considerate il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza	Priorità
$P_1$	10	3
$P_2$	1	1
$P_3$	2	3
$P_4$	1	4
$P_5$	5	2

Presumiamo che i processi siano arrivati nell'ordine  $P_1, P_2, P_3, P_4, P_5$ , e siano tutti presenti al tempo 0.

- Disegnate quattro diagrammi di Gantt che illustrino l'esecuzione di questi processi con gli algoritmi di scheduling FCFS, SJF, con priorità senza prelazione (un numero di priorità più basso indica una priorità maggiore) e RR (quanto = 1).
- Calcolate il tempo di completamento di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- Calcolate il tempo d'attesa di ciascun processo per ciascun algoritmo di scheduling di cui al punto a).
- Dite quale, fra le esecuzioni di cui al punto a), ha il minimo tempo medio d'attesa (per tutti i processi).

- 5.12 Quale tra i seguenti algoritmi di scheduling potrebbe generare un'attesa indefinita?

- In ordine di arrivo (FCFS).
- Per brevità (SJF).
- Circolare (RR).
- Per priorità.

- 5.13 Data una variante dell'algoritmo di scheduling RR in cui gli elementi della coda dei processi pronti sono puntatori ai PCB:

- descrivete l'effetto dell'inserimento di due puntatori allo stesso processo nella coda dei processi pronti;
- descrivete principali vantaggi e svantaggi di questo schema;
- ipotizzate una modifica all'algoritmo RR ordinario che consenta di ottenere lo stesso effetto senza duplicare i puntatori.

- 5.14 Considerate un sistema su cui vengano eseguiti dieci processi con prevalenza di I/O e un processo con prevalenza di elaborazione. Supponiamo che i primi processi richiedano un'operazione di I/O ogni millisecondo di elaborazione della CPU, e che ciascuna di tali operazioni sia completata in 10 millisecondi. Ipotizzate, inoltre, che il tempo necessario per il cambio di contesto sia di 0,1 millisecondi e che tutti i processi siano operazioni a lungo termine. Calcolate l'utilizzo della CPU in presenza di scheduler circolare RR se:

- il quanto di tempo è pari a 1 millisecondo;
- il quanto di tempo è pari a 10 millisecondi.

- 5.15 Considerate un sistema che applichi un algoritmo di scheduling a code multiple. A quale strategia può ricorrere l'utente che voglia ottenere per un suo processo la massima quantità di tempo dalla CPU?
- 5.16 Considerate il seguente algoritmo di scheduling con diritto di prelazione, e basato su priorità variabili dinamicamente. I numeri di priorità maggiori indicano una priorità più alta. Quando un processo attende la CPU (nella coda dei processi pronti), la sua priorità varia a un tasso  $\alpha$ ; quando è in esecuzione, la sua priorità varia a un tasso  $\beta$ . All'ingresso nella coda dei processi pronti, si attribuisce la priorità 0 a tutti i processi. I parametri  $\alpha$  e  $\beta$  si possono impostare in modo da fornire algoritmi di scheduling diversi.
- Descrivete l'algoritmo risultante da  $\beta > \alpha > 0$ .
  - Descrivete l'algoritmo risultante da  $\alpha < \beta < 0$ .
- 5.17 Spiegate a che cosa sia dovuta la maggiore preferenza che i seguenti algoritmi di scheduling accordano ai processi di breve durata:
- in ordine di arrivo (FCFS);
  - circolare (RR);
  - a code multiple con retroazione.
- 5.18 Usando l'algoritmo di scheduling di Windows XP, quale priorità numerica è assegnata, nei casi che seguono, a:
- Un thread appartenente alla `REALTIME_PRIORITY_CLASS` con una priorità relativa `HIGHEST`.
  - Un thread appartenente alla `NORMAL_PRIORITY_CLASS` con una priorità relativa `NORMAL`.
  - Un thread appartenente alla `HIGH_PRIORITY_CLASS` con una priorità relativa `ABOVE_NORMAL`.
- 5.19 Considerate l'algoritmo di scheduling del sistema operativo Solaris per i thread a tempo ripartito.
- Qual è il quanto di tempo (in millisecondi) per un thread con priorità 10? E per uno con priorità 55?
  - Ipotizzate che un thread con priorità 35 abbia consumato l'intera porzione di tempo riservatagli senza bloccarsi. Quale sarà la nuova priorità assegnata dallo scheduler a questo thread?
  - Poniamo che un thread con priorità 35 si blocchi per l'I/O prima che la propria porzione di tempo sia finita. Qual è la nuova priorità che lo scheduler assegnerà a questo thread?
- 5.20 Lo scheduler tradizionale di UNIX stabilisce una relazione inversa fra numeri e priorità: più alto è il numero, più bassa la priorità. Lo scheduler calcola *ex novo* le priorità dei processi una volta ogni secondo, usando la funzione seguente:
- $$\text{Priorità} = (\text{uso recente della CPU}/2) + \text{base}$$
- dove  $\text{base} = 60$  e *uso recente della CPU* è un valore che esprime la frequenza con cui un processo si è servito della CPU dall'ultima determinazione delle priorità.
- Ipotizziamo che l'uso recente della CPU per il processo  $P_1$  sia 40, per il processo  $P_2$  sia 18, e per il processo  $P_3$  sia 10. Quali saranno le nuove priorità di questi tre processi? Alla luce di quanto ottenuto, lo scheduler tradizionale di UNIX aumenta o diminuisce la priorità relativa di un processo a prevalenza di elaborazione?

## 5.9 Note bibliografiche

---

Le code con retroazione furono originariamente implementate sul sistema CTSS descritto in [Corbato et al. 1962]. Questa tecnica di scheduling è stata analizzata in [Schrage 1967]. L'algoritmo basato sulle priorità con prelazione dell'Esercizio 5.16 è stato suggerito da [Kleinrock 1975].

[Anderson et al. 1989], [Lewis e Berg 1998] e [Philbin et al. 1996] analizzano lo scheduling dei thread. Lo scheduling di processori multicore è trattato da McNairy e Bhatia [2005] e Kongetira et al. [2005].

Lo scheduling basato sulle quote (*fair-share*) è trattato da [Henry 1984], [Woodside 1986], e [Kay e Lauder 1988].

Le strategie di scheduling di UNIX V sono descritte da [Bach 1987]; quelle di UNIX FreeBSD 5.2 da [McKusick et al. 2005] e Neville-Neil [2005]; quelle di Mach da [Black 1990]. Love [2005] tratta lo scheduling in Linux. I dettagli dello scheduler ULE si trovano in Roberson [2003]. Lo scheduling di Solaris è trattato in [Mauro e McDougall 2007]. [Solomon 1998] e [Solomon e Russinovich 2000] e [Russinovich e Solomon 2005] trattano dello scheduling di Windows, rispettivamente. [Butenhof 1997] e [Lewis e Berg 1998] si occupano dello scheduling in Pthread. Siddha et al. [2007] discutono gli obiettivi futuri dello scheduling in sistemi multicore.

## Capitolo 6

# Sincronizzazione dei processi



### OBIETTIVI

- Introduzione al problema della sezione critica, le cui soluzioni – sia hardware sia software – sono utilizzabili per assicurare la coerenza dei dati condivisi.
- Introduzione al concetto di transazione atomica e descrizione dei meccanismi atti ad assicurare l'atomicità.

Un processo cooperante è un processo che può influenzarne un altro in esecuzione nel sistema o anche subirne l'influenza. I processi cooperanti possono condividere direttamente uno spazio logico di indirizzi (cioè, codice e dati) oppure condividere dati soltanto attraverso i file. Nel primo caso si fa uso dei thread, presentati nel Capitolo 4. L'accesso concorrente a dati condivisi può tuttavia causare situazioni di incoerenza degli stessi dati. In questo capitolo si trattano vari meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti, che condividono uno spazio logico di indirizzi, così da mantenere la coerenza dei dati.

## 6.1 Introduzione

Nel Capitolo 3 è stato descritto un modello di sistema costituito da un certo numero di processi sequenziali cooperanti o thread, tutti in esecuzione asincrona e con la possibilità di condividere dati. Tale modello è illustrato attraverso l'esempio del produttore/consumatore, che ben rappresenta molte situazioni che riguardano i sistemi operativi. Nel Paragrafo 3.4.1, in particolare, si è descritto come un buffer limitato sia utilizzabile per permettere ai processi la condivisione della memoria.

Torniamo al concetto di buffer limitato. Come è stato sottolineato, la nostra soluzione consente la presenza contemporanea di non più di `DIM_BUFFER - 1` elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera, `contatore`, inizializzata a 0, che si incrementa ogniqualvolta s'inscrive un nuovo elemento nel buffer e si decrementa ogniqualvolta si preleva un elemento dal buffer. Il codice per il processo produttore si può modificare come segue:

```
while (true)
{
    /* produce un elemento in appena_Prodotto */
```

```

        while (contatore == DIM_BUFFER)
            ; /* non fa niente */
        buffer[inserisci] = appena_Prodotto;
        inserisci = (inserisci + 1) % DIM_BUFFER;
        contatore++;
    }

```

Il codice per il processo consumatore si può modificare come segue:

```

while (true)
{
    while (contatore == 0)
        ; /* non fa niente */
    da_Consumare = buffer[preleva];
    preleva = (preleva + 1) % DIM_BUFFER;
    contatore--;
    /* consuma un elemento in da_Consumare */
}

```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile `contatore` sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni `contatore++` e `contatore-` in modo concorrente. Terminata l'esecuzione delle due istruzioni, il valore della variabile `contatore` potrebbe essere 4, 5 o 6! Il solo risultato corretto è `contatore == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Si può dimostrare che il valore di `contatore` può essere scorretto: l'istruzione `contatore++` si può codificare in un tipico linguaggio macchina, come

```

registro1 := contatore
registro1 := registro1 + 1
contatore := registro1

```

dove `registro1` è un registro locale della CPU. Analogamente, l'istruzione `contatore-` si può codificare come

```

registro2 := contatore
registro2 := registro2 - 1
contatore := registro2

```

dove `registro2` è un registro locale della CPU. Anche se `registro1` e `registro2` possono essere lo stesso registro fisico, per esempio un accumulatore, occorre ricordare che il contenuto di questo registro viene salvato e recuperato dal gestore dei segnali d'interruzione (Paragrafo 1.2.3).

L'esecuzione concorrente delle istruzioni `contatore++` e `contatore-` equivale a un'esecuzione sequenziale delle istruzioni del linguaggio macchina introdotte precedentemente, intercalate (*interleaved*) in una qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione di alto livello. Una di queste sequenze è

---

```

 $T_0:$  produttore esegue registro1 := contatore {registro1 = 5}
 $T_1:$  produttore esegue registro1 := registro1 + 1 {registro1 = 6}
 $T_2:$  consumatore esegue registro2 := contatore {registro2 = 5}
 $T_3:$  consumatore esegue registro2 := registro2 - 1 {registro2 = 4}
 $T_4:$  produttore esegue contatore := registro1 {contatore = 6}
 $T_5:$  consumatore esegue contatore := registro2 {contatore = 4}

```

e conduce al risultato errato in cui **contatore == 4**; si registra la presenza di 4 elementi nel buffer, mentre in realtà gli elementi sono 5. Se si invertisse l'ordine delle istruzioni in  $T_4$  e  $T_5$  si giungerebbe allo stato errato in cui **contatore == 6**.

Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette **race condition**) occorre assicurare che un solo processo alla volta possa modificare la variabile **contatore**. Questa condizione richiede una forma di sincronizzazione dei processi.

Tali situazioni si verificano spesso nei sistemi operativi, nei quali diversi componenti compiono operazioni su risorse condivise. Inoltre, con la diffusione dei sistemi multicore si dà sempre più importanza allo sviluppo di applicazioni multithread in cui diversi thread, che possono anche condividere dei dati, sono in esecuzione in parallelo su unità di calcolo distinte. Ovviamente tali operazioni non devono interferire reciprocamente in modi indesiderati. Data l'importanza della questione, la maggior parte di questo capitolo è dedicata ai problemi della sincronizzazione e coordinazione dei processi.

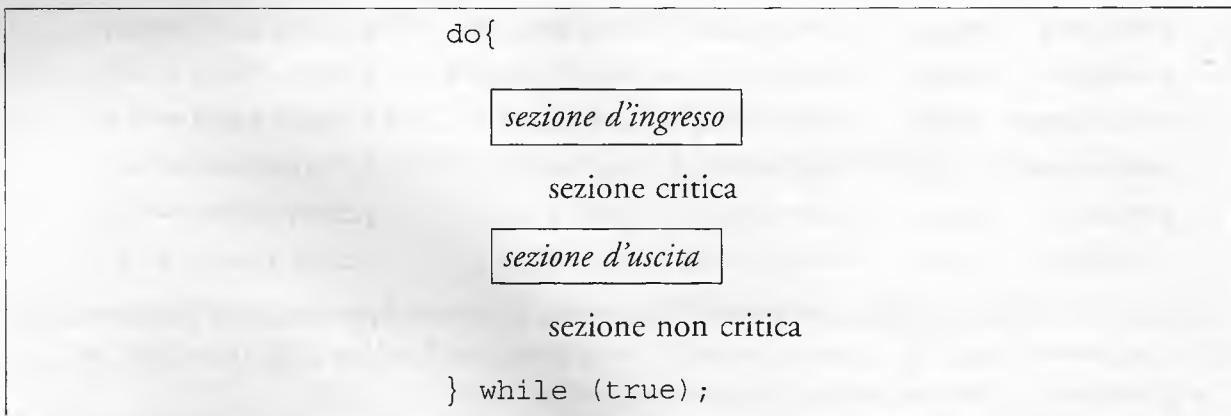
## 6.2 Problema della sezione critica

---

Si consideri un sistema composto di  $n$  processi  $\{P_0, P_1, \dots, P_{n-1}\}$  ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica. Quindi, l'esecuzione delle sezioni critiche da parte dei processi è *mutuamente esclusiva* nel tempo. Il problema della **sezione critica** si affronta progettando un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**. La Figura 6.1 mostra la struttura generale di un tipico processo  $P_i$ . La sezione d'ingresso e quella d'uscita sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti.

1. **Mutua esclusione.** Se il processo  $P_i$  è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
2. **Progresso.** Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.



**Figura 6.1** Struttura generale di un tipico processo  $P_r$

3. **Attesa limitata.** Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla **velocità relativa** degli  $n$  processi.

In un dato momento, numerosi processi in modalità utente possono essere attivi nel sistema operativo. Se ciò si verifica, il codice del kernel, che implementa il sistema operativo, si trova a dover regolare gli accessi ai dati condivisi. Si consideri per esempio una struttura dati del kernel che mantenga una lista di tutti i file aperti nel sistema. Tale lista deve essere modificata quando un nuovo file è aperto, e quindi aggiunto all'elenco, oppure chiuso, e quindi tolto dall'elenco. Due o più processi che dovessero aprire dei file, ognuno per proprio conto e simultaneamente, potrebbero ingenerare nel sistema una cosiddetta *race condition* legata ai necessari aggiornamenti della lista dei file aperti. Altre strutture dati del kernel soggette a problemi analoghi sono quelle per l'allocazione della memoria, per la gestione delle interruzioni e le liste dei processi. La responsabilità di preservare il *sistema operativo* da simili problemi relativi all'ordine degli accessi compete a chi sviluppa il kernel.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l'impiego di: (1) **kernel con diritto di prelazione** e (2) **kernel senza diritto di prelazione**. Un kernel con diritto di prelazione consente che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. Un kernel senza diritto di prelazione non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU. In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegnava il kernel. Altrettanto non si può dire dei kernel con diritto di prelazione, motivo per cui bisogna avere cura, nella progettazione, di mantenerli al riparo dai problemi insiti nell'ordine degli accessi. I kernel con diritto di prelazione rivelano particolari difficoltà di progettazione quando sono destinati ad architetture SMP, poiché in tali ambienti due processi nella modalità di sistema possono essere eseguiti in contemporanea su processori differenti.

Perché, allora, i kernel con diritto di prelazione dovrebbero essere preferiti a quelli senza diritto di prelazione? I kernel con diritto di prelazione sono più adatti alla programmazione real-time, dal momento che permette ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel. Inoltre, i kernel con dirit-

to di prelazione possono vantare una maggiore prontezza nelle risposte, data la loro scarsa propensione a eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa. Naturalmente, il fatto che questo effetto sia notevole o trascurabile dipende dai dettagli del codice del kernel. Vedremo più avanti come diversi sistemi operativi usino la prelazione all'interno del kernel.

## 6.3 Soluzione di Peterson

Illustriamo adesso una classica soluzione software al problema della sezione critica, nota come **soluzione di Peterson**. A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali `load` e `store`, non è affatto certo che la soluzione di Peterson funzioni correttamente su tali sistemi. Tuttavia si è scelto di presentarla ugualmente perché rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune insidie legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

La soluzione di Peterson si applica a due processi,  $P_0$  e  $P_1$ , ognuno dei quali esegue alternativamente la propria sezione critica e la sezione rimanente. Per il seguito, è utile convenire che se  $P_i$  denota uno dei due processi,  $P_j$  denoti l'altro; ossia, che  $j = 1 - i$ .

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

```
int turno;
boolean flag[2];
```

La variabile `turno` segnala, per l'appunto, di chi sia il turno d'accesso alla sezione critica; quindi, se `turno == i`, il processo  $P_i$  è autorizzato a eseguire la propria sezione critica. L'array `flag`, invece, indica se un processo *sia pronto* a entrare nella propria sezione critica. Per esempio, se `flag[i]` è `true`,  $P_i$  lo è. In base a queste delucidazioni, possiamo ora analizzare l'algoritmo descritto nella Figura 6.2.

Per accedere alla sezione critica, il processo  $P_i$  assegna innanzitutto a `flag[i]` il valore `true`; quindi attribuisce a `turno` il valore  $j$ , conferendo così all'altro processo la facoltà di entrare nella sezione critica. Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a `turno` sia il valore  $i$  sia il valore  $j$ . Soltanto uno dei due permane: l'altro sarà immediatamente sovrascritto. Il valore definitivo

```
do {
    flag[i] = true;
    turno = j;
    while (flag[j] && turno == j);

    sezione critica

    flag[i] = false;

    sezione non critica
} while (true);
```

**Figura 6.2** Struttura del processo  $P_i$  nella soluzione di Peterson.

di turno stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica.

Dimostriamo ora la correttezza di questa soluzione. Dobbiamo provare che:

1. la mutua esclusione è preservata;
2. il requisito del progresso è soddisfatto;
3. il requisito dell'attesa limitata è rispettato.

Per dimostrare la proprietà 1, si osservi come ogni  $P_i$  acceda alla propria sezione critica solo se  $\text{flag}[j] == \text{false}$  oppure  $\text{turno} == i$ . Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche, allora  $\text{flag}[0] == \text{flag}[1] == \text{true}$ . Si desume da queste due osservazioni che  $P_0$  e  $P_1$  sono impossibilitati a eseguire con successo le rispettive istruzioni `while` approssimativamente nello stesso momento: `turno`, infatti, può valere 0 o 1, ma non entrambi. Pertanto, uno dei processi – poniamo  $P_j$  – deve aver eseguito con successo l'istruzione `while`, mentre  $P_i$  aveva da eseguire almeno un'istruzione aggiuntiva (“`turno == j`”). Tuttavia, poiché da quel momento, e fino al termine della permanenza di  $P_j$  nella propria sezione critica, restano valide le assicurazioni  $\text{flag}[j] == \text{true}$  e  $\text{turno} == j$ , ne consegue che la mutua esclusione è preservata.

Per dimostrare le proprietà 2 e 3, osserviamo come l'ingresso di un processo  $P_i$  nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione `while`, con le condizioni  $\text{flag}[j] == \text{true}$  e  $\text{turno} == j$ ; questa è l'unica possibilità. Qualora  $P_j$  non sia pronto a entrare nella sezione critica,  $\text{flag}[j] == \text{false}$ , e  $P_i$  può accedere alla propria sezione critica. Se  $P_j$  ha impostato  $\text{flag}[j]$  a `true` e sta eseguendo il proprio ciclo `while`,  $\text{turno} == i$ , oppure  $\text{turno} == j$ . Se  $\text{turno} == i$ ,  $P_i$  entrerà nella propria sezione critica. Se  $\text{turno} == j$ ,  $P_j$  entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica,  $P_j$  reimposta  $\text{flag}[j]$  a `false`, consentendo a  $P_i$  di entrarvi. Se  $P_j$  imposta  $\text{flag}[j]$  a `true`, deve anche attribuire alla variabile `turno` il valore `i`. Poiché tuttavia  $P_i$  non modifica il valore della variabile `turno` durante l'esecuzione dell'istruzione `while`,  $P_i$  entrerà nella sezione critica (progresso) dopo che  $P_j$  abbia effettuato non più di un ingresso (attesa limitata).

## 6.4 Hardware per la sincronizzazione

Abbiamo appena descritto una soluzione software al problema della sezione critica. In generale, si può affermare che qualunque soluzione al problema richiede l'uso di un semplice strumento detto *lock* (*lucchetto*). Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che le sezioni critiche sono protette da lock. In altri termini, per accedere alla propria sezione critica un processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita. Si veda in proposito la Figura 6.3.

Nelle pagine seguenti esploreremo altre soluzioni al problema della sezione critica che sfruttano tecniche diverse, dai meccanismi hardware alle API disponibili per i programmatori. Tali soluzioni si basano tutte sul concetto di *lock*; come si vedrà, la progettazione di un lock può divenire complessa.

Iniziamo presentando alcune semplici istruzioni hardware disponibili in molti sistemi e mostrando come queste possano essere efficacemente utilizzate per risolvere il problema della sezione critica. Le funzionalità hardware possono rendere più facile il compito del programmatore e migliorare l'efficienza del sistema.

In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si potessero interdire le interruzioni mentre si modificano le variabili condivise. In

```

do{
    acquisisce il lock
    sezione critica
    restituisce il lock
    sezione non critica
} while (true);

```

**Figura 6.3** Soluzione al problema della sezione critica tramite lock.

questo modo si assicurerrebbe un'esecuzione ordinata e senza possibilità di prelazione della corrente sequenza di istruzioni; non si potrebbe eseguire nessun'altra istruzione, quindi non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. È questo l'approccio seguito dai kernel senza diritto di prelazione.

Sfortunatamente questa soluzione non è sempre praticabile; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione delle interruzioni a tutte le unità d'elaborazione. Tale trasmissione ritarda l'accesso a ogni sezione critica determinando una diminuzione dell'efficienza. Si considerino, inoltre, gli effetti su un orologio di sistema aggiornato tramite le interruzioni.

Per questo motivo molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo **atomico** – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice. Anziché discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali che stanno alla base di queste istruzioni.

L'istruzione **TestAndSet()** si può definire com'è illustrato nella Figura 6.4. Questa istruzione è eseguita **atomicamente**, cioè come un'unità non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni **TestAndSet()**, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. Se si dispone dell'istruzione **TestAndSet()**, si può realizzare la mutua esclusione dichiarando una variabile booleana globale **lock**, inizializzata a **false**. La struttura del processo  $P_i$  è illustrata nella Figura 6.5.

L'istruzione **Swap()**, definita nella Figura 6.6, agisce sul contenuto di due parole di memoria; come l'istruzione **TestAndSet()**, è anch'essa eseguita atomicamente. Se si dispo-

```

boolean TestAndSet(boolean *obiettivo) {
    boolean valore = *obiettivo;
    *obiettivo = true;
    return valore;
}

```

**Figura 6.4** Definizione dell'istruzione **TestAndSet()**.

```

do {
    while (TestAndSet (&lock));
        sezione critica
        lock = false;
        sezione non critica
    } while (true);

```

**Figura 6.5** Realizzazione di mutua esclusione con `TestAndSet()`.

ne dell'istruzione `Swap()`, la mutua esclusione si garantisce dichiarando e inizializzando al valore `false` una variabile booleana globale `lock`. Inoltre, ogni processo possiede anche una variabile booleana locale `chiave`. La struttura del processo  $P_i$  è illustrata nella Figura 6.7.

Questi algoritmi soddisfano il requisito della mutua esclusione, ma non quello dell'attesa limitata. La Figura 6.8 mostra un altro algoritmo che sfrutta l'istruzione `TestAndSet()` per soddisfare tutti e tre i requisiti desiderati. Le strutture dati condivise sono:

```

boolean attesa[n];
boolean lock;

```

e sono inizializzate al valore `false`. Per dimostrare che l'algoritmo soddisfa il requisito di mutua esclusione, si considera che il processo  $P_i$  possa entrare nella propria sezione critica solo se `attesa[i] == false` oppure `chiave == false`. Il valore di `chiave` può diventare `false` solo se si esegue `TestAndSet()`. Il primo processo che esegue `TestAndSet()` trova `chiave == false`; tutti gli altri devono attendere. La variabile `attesa[i]` può diventare `false` solo se un altro processo esce dalla propria sezione critica; solo una variabile `attesa[i]` vale `false`, il che consente di rispettare il requisito di mutua esclusione.

Per dimostrare che l'algoritmo soddisfa il requisito di progresso, basta osservare che le argomentazioni fatte per la mutua esclusione valgono anche in questo caso; infatti un processo che esce dalla sezione critica imposta `lock` al valore `false` oppure `attesa[j]` al valore `false`; entrambe consentono a un processo in attesa l'ingresso nella propria sezione critica.

Per dimostrare che l'algoritmo soddisfa il requisito di attesa limitata occorre osservare che un processo, quando lascia la propria sezione critica, scandisce il vettore `attesa` nell'ordinamento circolare ( $i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1$ ) e designa il primo processo in questo ordinamento presente nella sezione d'ingresso (`attesa[j] == true`) come il primo

```

void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

```

**Figura 6.6** Definizione dell'istruzione `Swap()`.

```

do {

    chiave = true;
    while (chiave == true)
        Swap(&lock, &chiave);

    sezione critica

    lock = false;

    sezione non critica

} while (true);

```

**Figura 6.7** Realizzazione di mutua esclusione con Swap().

processo che deve entrare nella propria sezione critica. Qualsiasi processo che attende l'ingresso nella propria sezione critica può farlo entro  $n - 1$  turni.

Per gli sviluppatori hardware la progettazione delle istruzioni atomiche TestAndSet() per sistemi multiprocessore non è certo un compito banale. Quest'argomento è trattato nei testi di architetture dei calcolatori.

```

do {

    attesa[i] = true;
    chiave = true;
    while (attesa[i] && chiave)
        chiave = TestAndSet(&lock);
    attesa[i] = false;

    sezione critica

    j = (i + 1) % n;
    while ((j != i) && !attesa[i])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        attesa[j] = false;

    sezione non critica

} while (true);

```

**Figura 6.8** Mutua esclusione con attesa limitata con TestAndSet().

## 6.5 Semafori

Le varie soluzioni hardware al problema della sezione critica presentate nel Paragrafo 6.4, basate su istruzioni quali `TestAndSet()` e `Swap()`, complicano l'attività dei programmati di applicazioni. Per superare questa difficoltà si può usare uno strumento di sincronizzazione chiamato semaforo.

Un **semaforo** `S` è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`. Queste operazioni erano originariamente chiamate `P` (per `wait()`; dall'olandese *proberen*, verificare) e `V` (per `signal()`; da *verhogen*, incrementare). La definizione classica di `wait()` in pseudocodice è la seguente:

```
wait(S) {
    while(S <= 0)
        ;//non-op
    S--;
}
```

La definizione classica di `signal()` in pseudocodice è la seguente:

```
signal(S) {
    S++;
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Inoltre, nel caso della `wait(S)` si devono eseguire senza interruzione anche la verifica del valore intero di `S` ( $S \leq 0$ ) e la sua possibile modifica (`S--`). Nel Paragrafo 6.5.2 si spiega come si possono realizzare queste operazioni.

### 6.5.1 Uso dei semafori

Si usa distinguere tra **semafori contatore**, il cui valore numerico è illimitato, e i **semafori binari**, il cui valore è 0 o 1. In relazione a certi sistemi i semafori binari sono anche detti **lock mutex** (*mutex locks*), perché fungono da “lock” che garantiscono la mutua esclusione (dall'inglese *mutual exclusion*).

I semafori sono utilizzabili per risolvere il problema della sezione critica con  $n$  processi. Gli  $n$  processi condividono un semaforo comune, **mutex**, inizializzato a 1. Ogni processo  $P_i$  è strutturato com'è illustrato nella Figura 6.9.

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderino utilizzare un esemplare della risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono un esemplare della risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, vengono allocati tutti gli esemplari della risorsa, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente:  $P_1$  con un'istruzione  $S_1$  e  $P_2$  con un'istruzione  $S_2$ . Si supponga di voler eseguire  $S_2$  solo dopo che  $S_1$  è terminata. Que-

```

do {
    wait(mutex);
    sezione critica
    signal(mutex);
    sezione non critica
} while (true);

```

**Figura 6.9** Realizzazione di mutua esclusione con semafori.

sto schema si può prontamente realizzare facendo condividere a  $P_1$  e  $P_2$  un semaforo comune, `sincronizzazione`, inizializzato a 0, e inserendo nel processo  $P_1$  le istruzioni

```

S1;
signal(sincronizzazione);

```

e nel processo  $P_2$  le istruzioni

```

wait(sincronizzazione);
S2;

```

Poiché `sincronizzazione` è inizializzato a 0,  $P_2$  esegue  $S_2$  solo dopo che  $P_1$  ha eseguito `signal(sincronizzazione)`, che si trova dopo  $S_1$ .

## 6.5.2 Realizzazione

Il principale svantaggio della definizione di semaforo è che richiede una condizione di **attesa attiva** (*busy waiting*). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della sezione d'ingresso. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione d'attesa attiva spreca cicli della CPU che un altro processo potrebbe sfruttare in modo produttivo. Questo tipo di semaforo è anche detto **spinlock**, perché i processi “girano” (*spin*) mentre attendono al semaforo. (I semafori spinlock hanno però il vantaggio di non richiedere cambio di contesto nel caso in cui un processo sia fermo in attesa. Tali cambi di contesto possono essere piuttosto costosi, in termini di tempo. Ne consegue che i semafori spinlock sono utili quando i lock sono applicati per brevi intervalli di tempo: infatti, trovano frequente applicazione nei sistemi multiprocessore, dove un processo gira su un processore mentre un altro thread esegue la propria sezione critica su un altro processore.)

Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni `wait()` e `signal()`: quando un processo invoca l'operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può **bloccare** se stesso. L'operazione di **bloccaggio** pone il processo in una coda d'attesa associata al semaforo e cambia lo stato del processo nello stato d'attesa. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione.

Un processo bloccato, che attende a un semaforo  $S$ , sarà riavviato in seguito all'esecuzione di un'operazione `signal()` su  $S$  da parte di qualche altro processo. Il processo si riav-

via tramite un'operazione `wakeup()`, che modifica lo stato del processo da attesa a pronto. Il processo entra nella coda dei processi pronti. (L'uso della CPU può essere o non essere commutato dal processo in esecuzione al processo appena divenuto pronto, a seconda del criterio di scheduling.)

Per realizzare i semafori secondo quel che s'è detto si può definire il semaforo come una struttura del linguaggio C:

```
typedef struct {
    int valore;
    struct processo *lista;
} semaforo;
```

A ogni semaforo sono associati un `valore` intero e una `lista` di processi, contenente i processi in attesa a un semaforo; l'operazione `signal()` preleva un processo da tale lista e lo attiva.

L'operazione `wait()` del semaforo si può definire come segue:

```
wait(semaforo *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi questo processo a S->lista;
        block();
    }
}
```

L'operazione `signal()` del semaforo si può definire come segue:

```
signal(semaforo *S) {
    S->valore++;
    if (S->valore <= 0) {
        togli un processo P da S->lista;
        wakeup(P);
    }
}
```

L'operazione `block()` sospende il processo che la invoca; l'operazione `wakeup(P)` pone in stato di pronto per l'esecuzione un processo P bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema di base.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, tale definizione può condurre a valori negativi. Se il valore del semaforo è negativo, la sua dimensione è data dal numero dei processi che attendono a quel semaforo. Ciò avviene a causa dell'inversione dell'ordine del decremento e della verifica nel codice dell'operazione `wait()`.

La lista dei processi che attendono a un semaforo si può facilmente realizzare inserendo un campo puntatore in ciascun blocco di controllo del processo (PCB). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Per aggiungere e togliere processi dalla lista assicurando un'attesa limitata si può usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale si può usare *qualsiasi* criterio d'accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

I semafori devono essere eseguiti in modo atomico. Si deve garantire che nessuno dei due processi possa eseguire operazioni `wait()` e `signal()` contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocessoresso lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di `signal()` e `wait()`. Nei sistemi con una sola CPU, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interruzioni, dando la possibilità allo scheduler di riprendere il controllo della CPU, il processo corrente continua indisturbato la sua esecuzione.

Nei sistemi multiprocessoresso è necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, disabilitare le interruzioni di tutti i processori può non essere cosa semplice, e causare un notevole calo delle prestazioni. È per questo che – per garantire l'esecuzione atomica di `wait()` e `signal()` – i sistemi SMP devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, gli spinlock).

È importante rilevare che questa definizione delle operazioni `wait()` e `signal()` non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi. Inoltre, l'attesa attiva si limita alle sezioni critiche delle operazioni `wait()` e `signal()`, che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l'attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l'attesa attiva è assai inefficiente.

### 6.5.3 Stallo e attesa indefinita

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento – l'esecuzione di un'operazione `signal()` – che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo si dice che i processi sono in **stallo** (*deadlocked*).

Per illustrare questo fenomeno si consideri un insieme di due processi,  $P_0$  e  $P_1$ , ciascuno dei quali ha accesso a due semafori, S e Q, impostati al valore 1:

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
.	.
.	.
.	.
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Si supponga che  $P_0$  esegua `wait(S)` e quindi  $P_1$  esegua `wait(Q)`; eseguita `wait(Q)`,  $P_0$  deve attendere che  $P_1$  esegua `signal(Q)`; analogamente, quando  $P_1$  esegue `wait(S)`, deve attendere che  $P_0$  esegua `signal(S)`. Poiché queste operazioni `signal()` non si possono eseguire,  $P_0$  e  $P_1$  sono in stallo.

## INVERSIONE DI PRIORITÀ E IL MARS PATHFINDER

L'inversione di priorità può essere più di un semplice inconveniente nello scheduling. Su sistemi con vincoli temporali molto restrittivi (come i sistemi real-time, si veda il Capitolo 19) l'inversione di priorità può far sì che un processo non venga eseguito nei tempi richiesti. Quando ciò succede possono innescarsi errori a cascata in grado di provocare un guasto del sistema.

Si consideri il caso del Mars Pathfinder, una sonda spaziale della NASA che nel 1997 portò il robot Sojourner rover su Marte per condurre un esperimento. Poco dopo che il Sojourner ebbe iniziato il suo lavoro, cominciarono ad aver luogo numerosi reset del sistema. Ciascun reset inizializzava di nuovo sia hardware che software, inclusi gli strumenti preposti alla comunicazione. Se il problema non fosse stato risolto, il Sojourner avrebbe fallito la sua missione.

Il problema era causato dal fatto che un processo ad alta priorità, di nome "bc\_dist", impiegava più tempo del dovuto a portare a termine il suo compito. Questo processo era forzatamente in attesa di una risorsa condivisa utilizzata da un processo a priorità inferiore, denominato "ASI/MET", a sua volta prelazionato da diversi processi di priorità media. Il processo "bc\_dist" andava quindi in stallo in attesa della risorsa condivisa, e il processo "bc\_sched", rilevando il problema, eseguiva il reset. Il Sojourner soffriva dunque di un tipico caso di inversione di priorità.

Il sistema operativo installato sul Sojourner era VxWorks (si veda il Paragrafo 19.6), che disponeva di una variabile globale per abilitare l'ereditarietà delle priorità su tutti i semafori. Dopo alcuni test, il valore della variabile del Sojourner (su Marte!) fu impostato correttamente, e il problema fu risolto.

Un resoconto completo del problema, della sua scoperta, e della sua soluzione è stato scritto dal responsabile del team software ed è disponibile all'indirizzo  
[http://research.microsoft.com/enus/um/people/mbj/Mars\\_Pathfinder/Authoritative\\_Account.html](http://research.microsoft.com/enus/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html).

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli *eventi di acquisizione e rilascio di risorse*, tuttavia anche altri tipi di eventi possono produrre situazioni di stallo (si veda il Capitolo 7, che descrive anche i meccanismi che servono ad affrontare questo tipo di problema).

Un'altra questione connessa alle situazioni di stallo è quella dell'**attesa indefinita** (nota anche col termine *starvation*). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO (last-in, first-out).

### 6.5.4 Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ognqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa ha dovere di prelazione su un processo a priorità più alta. Assumiamo, ad esempio, che vi siano tre processi *L*, *M* e *H*, le cui priorità seguono l'ordine *L* < *M* < *H*. Assumiamo che il processo *H* richieda la risorsa *R* alla quale sta accedendo il processo *L*. Usualmente il processo *H* resterebbe in attesa che *L* liberi la risorsa *R*. Supponiamo però che *M* diventi eseguibile, con prelazione sul processo *L*. Avviene indi-

rettamente che un processo con priorità più bassa, il processo  $M$ , influenzi il tempo che  $H$  attenderà in attesa della risorsa  $R$ .

Questo problema è noto come **inversione della priorità**. Dato che l'inversione di priorità si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un **protocollo di ereditarietà delle priorità**, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale. Nell'esempio discusso in precedenza, un protocollo di ereditarietà delle priorità avrebbe permesso al processo  $L$  di ereditare temporaneamente la priorità di  $H$ , impedendo così al processo  $M$  di prelazionare la sua esecuzione. In un tale caso, una volta che il processo  $H$  avrà terminato con la risorsa  $R$ , rinuncerà alla priorità ereditata da  $H$  assumendo di nuovo la priorità originale. Poiché  $R$  sarà a questo punto disponibile, il processo  $H$ , e non il processo  $M$ , sarà il successivo processo eseguito.

## 6.6 Problemi tipici di sincronizzazione

In questo paragrafo s'illustrano diversi problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. Questi problemi sono utili per verificare quasi tutte le nuove proposte di schemi di sincronizzazione. Nelle soluzioni che proponiamo ai problemi s'impiegano i semafori.

### 6.6.1 Produttori e consumatori con memoria limitata

Il problema dei *produttori e consumatori con memoria limitata*, trattato anche nel Paragrafo 6.1, si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento a nessuna realizzazione particolare. In conclusione del capitolo proponiamo al riguardo un progetto di programmazione.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con  $n$  posizioni, ciascuna capace di contenere un elemento. Il semaforo `mutex` garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1. I semafori `vuote` e `piene` conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo `vuote` si inizializza al valore  $n$ ; il semaforo `piene` si inizializza al valore 0.

La Figura 6.10 riporta la struttura generale del processo produttore, la Figura 6.11 quella del processo consumatore. È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

### 6.6.2 Problema dei lettori-scrittori

Si supponga che una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto dell'oggetto condiviso, mentre altri possono richiedere un aggiornamento, vale a dire una lettura e una scrittura, dello stesso oggetto. Questi due processi sono distinti, e si indicano chiamando **lettori** quelli interessati alla sola lettura e **scrittori** gli altri. Naturalmente, se due lettori accedono nello stesso mo-

```

do {
    . . .
    produce un elemento in appena_Prodotto
    . . .
    wait(vuote);
    wait(mutex);
    . . .
    inserisci in buffer l'elemento in appena_Prodotto
    . . .
    signal(mutex);
    signal(piene);
} while (true);

```

**Figura 6.10** Struttura generale del processo produttore.

mento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori**. Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità; la più semplice, cui si fa riferimento come al *primo* problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati. Il *secondo* problema dei lettori-scrittori si fonda sul presupposto che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema e quella del secondo possono condurre a uno stato d'attesa indefinita (*starvation*), degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. In questo paragrafo si presenta una solu-

```

do {
    wait(piene);
    wait(mutex);
    . . .
    rimuovi un elemento da buffer e mettilo in da_Consumare
    . . .
    signal(mutex);
    signal(vuote);
    . . .
    consuma l'elemento contenuto in da_Consumare
    . . .
} while (true);

```

**Figura 6.11** Struttura generale del processo consumatore.

zione del primo problema dei lettori-scrittori; indicazioni attinenti a soluzioni immuni all'attesa indefinita si trovano nelle Note bibliografiche.

La soluzione del primo problema dei lettori-scrittori prevede dunque la condivisione da parte dei processi lettori delle seguenti strutture dati:

```
semaforo mutex, scrittura;
int numlettori;
```

I semafori **mutex** e **scrittura** sono inizializzati a 1; **numlettori** è inizializzato a 0. Il semaforo **scrittura** è comune a entrambi i tipi di processi (lettura e scrittura). Il semaforo **mutex** si usa per assicurare la mutua esclusione al momento dell'aggiornamento di **numlettori**. La variabile **numlettori** contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati. Il semaforo **scrittura** funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che entra o esce dalla sezione critica. Non serve, invece, ai lettori che entrano o escono mentre altri lettori si trovano nelle rispettive sezioni critiche.

La Figura 6.12 illustra la struttura generale di un processo scrittore; la Figura 6.13 presenta la struttura generale di un processo lettore. Occorre notare che se uno scrittore si trova nella sezione critica e  $n$  lettori attendono di entrarvi, si accoda un lettore a **scrittura** e  $n - 1$  lettori a **mutex**. Inoltre, se uno scrittore esegue **signal (scrittura)** si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa. La scelta è fatta dallo scheduler.

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**. Per acquisire un tale lock, è necessario specificare la modalità di scrittura o di lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. È permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono massimamente utili nelle situazioni seguenti.

- ◆ Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi.
- ◆ Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di lettura-scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock **mutex**, compensato però dalla possibilità di eseguire molti lettori in concorrenza.

```
do {
    wait (scrittura);

    . . .
    esegui l'operazione di scrittura
    . . .
    signal (scrittura);
} while (true);
```

**Figura 6.12** Struttura generale di un processo scrittore.

```

do {
    wait(mutex);
    numlettori++;
    if (numlettori == 1)
        wait(scrrittura);
    signal(mutex);

    . . .
    esegui l'operazione di lettura
    . . .

    wait(mutex);
    numlettori--;
    if (numlettori == 0)
        signal(scrrittura);
    signal(mutex);
}while (true);

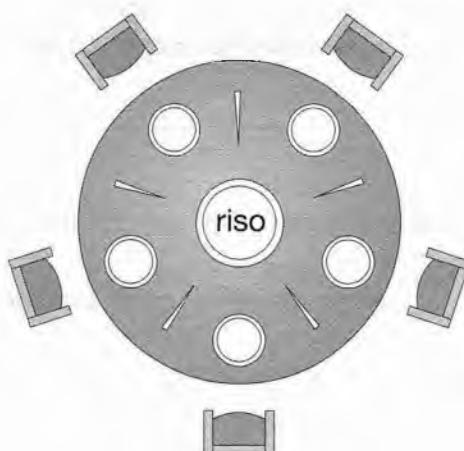
```

**Figura 6.13** Struttura generale di un processo lettore.

### 6.6.3 Problema dei cinque filosofi

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette (Figura 6.14). Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

Il problema dei cinque filosofi è considerato un classico problema di sincronizzazione, non certo per la sua importanza pratica, e neanche per antipatia verso i filosofi da parte degli informatici, ma perché rappresenta una vasta classe di problemi di controllo della con-



**Figura 6.14** Situazione dei cinque filosofi.

```

do {
    wait(bacchetta[i]);
    wait(bacchetta[(i + 1) % 5]);
    . . .
    mangia
    . . .
    signal(bacchetta[i]);
    signal(bacchetta[(i + 1) % 5]);
    . . .
    pensa
    . . .
} while (true);

```

**Figura 6.15** Struttura del filosofo *i*.

correnza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di stallo e d'attesa indefinita.

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati. Quindi, i dati condivisi sono

```
semaforo bacchetta[5];
```

dove tutti gli elementi `bacchetta` sono inizializzati a 1. La struttura del filosofo *i* è illustrata nella Figura 6.15.

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno tenti di afferrare la bacchetta di sinistra; tutti gli elementi di `bacchetta` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Di seguito sono elencate diverse possibili soluzioni per tali situazioni di stallo:

- ◆ solo quattro filosofi possono stare contemporaneamente a tavola;
- ◆ un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (occorre notare che quest'operazione si deve eseguire in una sezione critica);
- ◆ si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

Nel Paragrafo 6.7 presentiamo una soluzione al problema dei cinque filosofi che assicura l'assenza di situazioni di stallo. Si noti che qualsiasi soluzione soddisfacente per il problema dei cinque filosofi deve escludere la possibilità di situazioni d'attesa indefinita, in altre parole che uno dei filosofi muoia di fame (da qui il termine *starvation*) – una soluzione immune alle situazioni di stallo non esclude necessariamente la possibilità di situazioni d'attesa indefinita.

## 6.7 Monitor

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre.

L'impiego di contatori nell'ambito della soluzione al problema dei produttori e consumatori rappresenta un esempio di tali errori (Paragrafo 6.1). In quella circostanza, il problema di sincronizzazione appariva solo sporadicamente, e anche il valore del contatore si manteneva entro limiti ragionevoli, essendo sfasato tutt'al più di 1. Ciononostante, le soluzioni di questo tipo restano inaccettabili, ed è per ottenere soluzioni soddisfacenti che sono stati inventati i semafori.

Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo `mutex`, inizializzata a 1. Ogni processo deve eseguire `wait(mutex)` prima di entrare nella sezione critica e `signal(mutex)` al momento di uscirne. Se questa sequenza non è rispettata, può accadere che due processi occupino simultaneamente le rispettive sezioni critiche. Esaminiamo le difficoltà che possono insorgere. (Si noti che tali difficoltà possono insorgere anche nel caso che *un solo* processo abbia delle pecche. L'inconveniente può nascere da un involontario errore di programmazione o essere causato dalla negligenza del programmatore.)

- ◆ Supponiamo che un processo capovolga l'ordine in cui sono eseguite le istruzioni `wait()` e `signal()`, in questo modo:

```
signal(mutex);
. . .
sezione critica
. . .
wait(mutex);
```

In questa situazione, numerosi processi possono eseguire le proprie sezioni critiche allo stesso tempo, infrangendo il requisito della mutua esclusione. Questo errore può essere scoperto solo qualora diversi processi siano attivi simultaneamente nelle rispettive sezioni critiche. Si osservi che tale situazione potrebbe non essere sempre riproducibile.

- ◆ Ipotizziamo che un processo sostituisca `signal(mutex)` con `wait(mutex)`, cioè che esegua

```
wait(mutex);
. . .
sezione critica
. . .
wait(mutex);
```

Si genera, in questo caso, uno stallo (*deadlock*).

- ◆ Si supponga che un processo ometta `wait(mutex)`, `signal(mutex)`, o entrambi. In questo caso si viola la mutua esclusione oppure si genera uno stallo.

Questi esempi chiariscono come sia facile incorrere in errori allorché i programmatori utilizzino i semafori in maniera scorretta, nel tentativo di risolvere il problema delle sezioni critiche. Problemi di natura simile possono insorgere negli altri modelli di sincronizzazione, esaminati nel Paragrafo 6.6.

Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti con un linguaggio ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello – il tipo **monitor** – è descritto nel paragrafo successivo.

### 6.7.1 Uso del costrutto monitor

Un tipo, o tipo di dato astratto, incapsula i dati privati mettendo a disposizione dei metodi pubblici per operare su tali dati. Il tipo monitor presenta un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre ai corpi delle procedure o funzioni che operano su tali variabili. La sintassi di un monitor è mostrata nella Figura 6.16. La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una procedura definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione (Figura 6.17). Tale definizione di monitor non è abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi che, in questo caso, sono forniti dal costrutto **condition**.

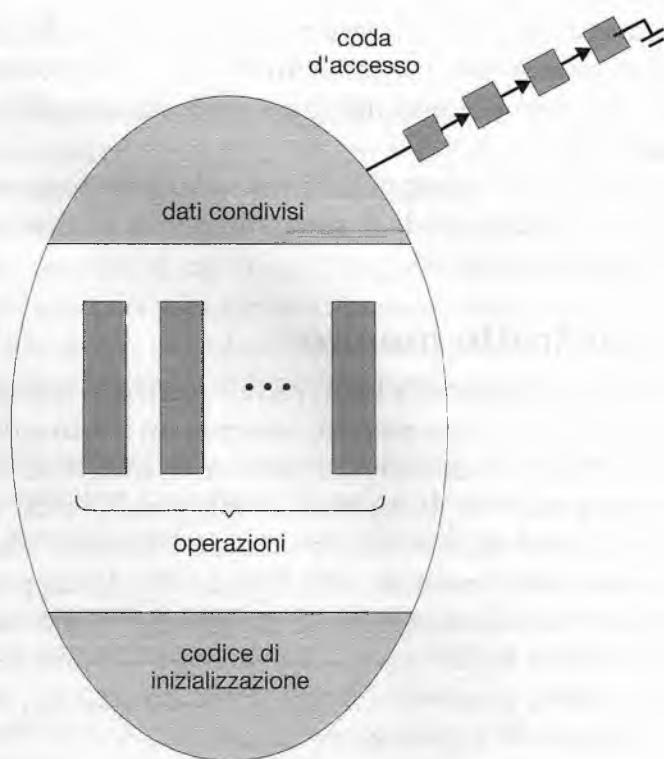
```

monitor nome_monitor
{
    dichiarazioni di variabili condivise

    procedure P1 ( . . . ) {
        .
    }
    procedure P2 ( . . . ) {
        .
    }
    .
    .
    .
    procedure Pn ( . . . ) {
        .
    }
    codice d'inizializzazione ( . . . ) {
        .
    }
}

```

**Figura 6.16** Sintassi di un monitor.



**Figura 6.17** Schema di un monitor.

Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali:

```
condition x, y;
```

Le uniche operazioni eseguibili su una variabile **condition** sono **wait()** e **signal()**. L'operazione

```
x.wait();
```

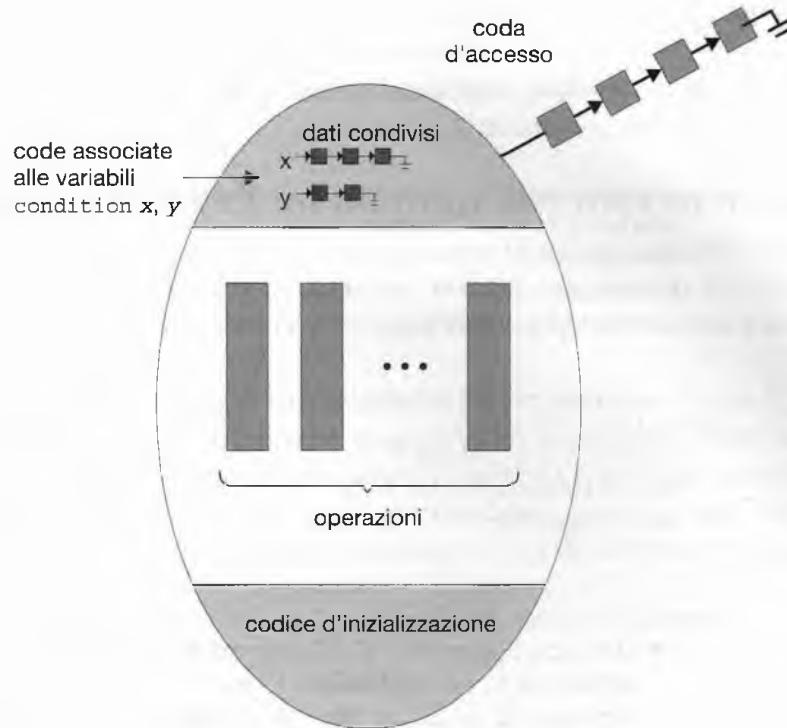
implica che il processo che la invoca rimanga sospeso finché un altro processo non invochi l'operazione

```
x.signal();
```

che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione **signal()** non ha alcun effetto, vale a dire che lo stato di **x** resta immutato, come se l'operazione non fosse stata eseguita; la situazione è descritta nella Figura 6.18. Tutto ciò contrasta con l'operazione **signal()** associata ai semafori, poiché questa influenza sempre sullo stato del semaforo.

Si supponga, per esempio, che quando un processo *P* invoca l'operazione **x.signal()**, esista un processo sospeso *Q* associato alla variabile **x** di tipo **condition**. Chiaramente, se al processo sospeso *Q* si permette di riprendere l'esecuzione, il processo segnalante *P* è costretto ad attendere, altrimenti *P* e *Q* sarebbero contemporaneamente attivi all'interno del monitor. Occorre in ogni modo notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono quindi due possibilità:

1. **segnalare e attendere.** *P* attende che *Q* lasci il monitor o attenda su un'altra variabile **condition**;
2. **segnalare e proseguire.** *Q* attende che *P* lasci il monitor o attenda su un'altra variabile **condition**.



**Figura 6.18** Monitor con variabili condition.

Si possono fornire argomenti ragionevoli a favore dell'uno o dell'altra opzione. Da un lato, visto che  $P$  era già in esecuzione all'interno del monitor, il secondo metodo appare più ragionevole. D'altro canto, se si lascia proseguire il thread  $P$ , la condizione attesa da  $Q$  potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione. Il linguaggio Concurrent Pascal ha scelto un compromesso: quando il thread  $P$  esegue l'operazione `signal()`, lascia subito il monitor; pertanto,  $Q$  riprende immediatamente l'esecuzione.

Molti linguaggi di programmazione incorporano l'idea di monitor descritta in questo paragrafo. Tra questi, il Concurrent Pascal, Mesa, C# (da leggersi "C-sharp") e Java. Altri linguaggi, come Erlang, forniscono alcune tipologie di supporto alla concorrenza usando un meccanismo simile.

### 6.7.2 Soluzione al problema dei cinque filosofi per mezzo di monitor

Illustriamo quindi i concetti relativi al costrutto monitor presentando una soluzione esente da stallo del problema dei cinque filosofi. La soluzione impone però il vincolo che un filosofo possa prendere le sue bacchette solo quando siano entrambe disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

```
enum {pensa, affamato, mangia} stato[5];
```

Il filosofo  $i$  può impostare la variabile `stato[i] = mangia` solo se i suoi due vicini non stanno mangiando:

```
((stato[(i + 4) % 5] != mangia) && (stato[(i + 1) % 5] != mangia)).
```

```

monitor fc
{
    enum {pensa, affamato, mangia} stato[5];
    condition auto[5];

    prende(int i) {
        stato[i] = affamato;
        verifica(i);
        if (stato[i] != mangia)
            auto[i].wait();
    }

    posa(int i) {
        stato[i] = pensa;
        verifica((i + 4) % 5);
        verifica((i + 1) % 5);
    }

    verifica(int i) {
        if ((stato[(i + 4) % 5] != mangia) &&
            (stato[i] == affamato) &&
            (stato[(i + 1) % 5] != mangia)) {
            stato[i] = mangia;
            auto[i].signal();
        }
    }

    codice di inizializzazione() {
        for (int i = 0; i < 5; i++)
            stato[i] = pensa;
    }
}

```

**Figura 6.19** Una soluzione con monitor al problema dei cinque filosofi.

Inoltre, occorre impiegare la seguente struttura dati:

```
condition auto[5];
```

dove il filosofo  $i$  può ritardare se stesso quando ha fame, ma non riesce a ottenere le bacchette di cui ha bisogno.

A questo punto si può descrivere la soluzione al problema dei cinque filosofi. La distribuzione delle bacchette è controllata dal monitor `fc` (Figura 6.19). Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione `prende()`; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione `posa()` e comincia a pensare. Il filosofo  $i$  deve quindi chiamare le operazioni `prende()` e `posa()` nella seguente sequenza:

```
fc.prende(i);
```

```
...
```

È facile dimostrare che questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di stallo. Occorre però notare che un filosofo può attendere indefinitamente. La soluzione di questo problema è lasciata come esercizio per il lettore.

### 6.7.3 Realizzazione di un monitor per mezzo di semafori

A questo punto si considera la possibilità di realizzare il meccanismo del monitor usando i semafori. A ogni monitor si associa un semaforo `mutex`, inizializzato a 1; un processo deve eseguire `wait(mutex)` prima di entrare nel monitor, e `signal(mutex)` dopo aver lasciato il monitor.

Poiché un processo che esegue una `signal()` deve attendere finché il processo risvegliato si metta in attesa o lasci il monitor, si introduce un altro semaforo, `prossimo`, inizializzato a 0, a cui i processi che eseguono una `signal()` possono autosospendersi. Per contare i processi sospesi al semaforo `prossimo`, si usa una variabile intera `prossimo_contatore`. Quindi, ogni procedura esterna di monitor `F` si sostituisce col seguente codice:

```
wait(mutex);
...
corpo di F
...
if (prossimo_contatore > 0)
    signal(prossimo);
else
    signal(mutex);
```

In questo modo si assicura la mutua esclusione all'interno del monitor.

A questo punto si può descrivere la realizzazione delle variabili `condition`. Per ogni variabile `x` di tipo `condition` si introducono un semaforo `x_sem` e una variabile intera `x_contatore`, entrambi inizializzati a 0. L'operazione `x.wait()` si può realizzare come segue:

```
x_contatore++;
if (prossimo_contatore > 0)
    signal(prossimo);
else
    signal(mutex);
wait(x_sem);
x_contatore--;
```

L'operazione `x.signal()` si può realizzare come segue:

```
if (x_contatore > 0) {
    prossimo_contatore++;
    signal(x_sem);
    wait(prossimo);
    prossimo_contatore--;
}
```

Questa soluzione è applicabile alle definizioni di monitor date da Hoare e Brinch-Hansen. In alcuni casi, tuttavia, questo livello di generalità della codifica non è necessario, e si possono apportare notevoli miglioramenti all'efficienza. La soluzione a questo problema è lasciata al lettore nell'Esercizio 6.27.

### 6.7.4 Ripresa dei processi all'interno di un monitor

A questo punto si discute il problema dell'ordine di ripresa dei processi all'interno di un monitor. Se più processi sono sospesi alla condizione *x*, e se qualche processo esegue l'operazione *x.signal()*, è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell'usare un ordinamento FCFS, secondo cui il processo che attende da più tempo viene ripreso per primo. Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di **attesa condizionale** della forma

```
x.wait(c);
```

dove con *c* si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione *wait()*. Il valore di *c*, chiamato **numero di priorità**, viene poi memorizzato col nome del processo sospeso. Quando si esegue *x.signal()*, si riprende il processo cui è associato il numero di priorità più basso.

Per comprendere questo nuovo meccanismo, si consideri il monitor illustrato nella Figura 6.20; tale monitor ha il compito di assegnare una particolare risorsa a processi in competizione. Quando richiede l'assegnazione di una delle sue risorse, ogni processo specifica il tempo massimo per il quale prevede di usare la risorsa. Il monitor assegna la risorsa al processo con la richiesta di assegnazione più breve.

```
monitor assegnazione_risorse
{
    boolean occupato;
    condition x;

    acquisizione(int tempo) {
        if (occupato)
            x.wait(tempo);
        occupato = true;
    }

    rilascio() {
        occupato = false;
        x.signal();
    }

    codice di inizializzazione() {
        occupato = false;
    }
}
```

**Figura 6.20** Un monitor per l'assegnazione di una singola risorsa.

Per accedere alla risorsa in questione il processo deve rispettare la sequenza

```
R.acquisizione(t);
...
accesso alla risorsa;
...
R.rilascio();
```

dove R è un'istanza di tipo `assegnazione_risorse`.

Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza d'accesso sia rispettata. In particolare, può accadere quanto segue:

- ◆ un processo può accedere alla risorsa senza prima ottenere il permesso d'accesso;
- ◆ una volta che ne ha ottenuto l'accesso, un processo può non rilasciare più la risorsa;
- ◆ un processo può tentare di rilasciare una risorsa che non ha mai richiesto;
- ◆ un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta.

Le stesse difficoltà si sono incontrate col costrutto della sezione critica e sono, in realtà, simili alle difficoltà che condussero allo sviluppo dei costrutti di sezione critica e di monitor. In precedenza ci si è preoccupati del corretto uso dei semafori, ora ci si deve preoccupare del corretto uso delle operazioni ad alto livello definite dal programmatore, senza poter avere, a questo livello, l'assistenza del compilatore.

Una possibile soluzione del problema precedente prevede l'inclusione delle operazioni d'accesso alle risorse all'interno del monitor `assegnazione_risorse`. Tuttavia, adottando questa soluzione, per lo scheduling delle risorse si userebbe l'algoritmo di scheduling del monitor anziché quello desiderato.

Per garantire che i processi rispettino le sequenze appropriate, è necessario controllare tutti i programmi che usano il monitor `assegnazione_risorse` e la risorsa da esso gestita. Per stabilire la correttezza del sistema è necessario verificare le seguenti due condizioni: la prima, che i processi utenti devono sempre impiegare il monitor secondo una sequenza corretta; la seconda, che è necessario assicurare che un processo non cooperante non cerchi di aggirare la mutua esclusione offerta dal monitor, e tenti di accedere direttamente alla risorsa condivisa senza usare i protocolli d'accesso. Soltanto se si assicurano queste due condizioni, si può garantire l'assenza di errori di sincronizzazione e che l'algoritmo di scheduling sia rispettato.

Questo controllo è possibile per sistemi statici di piccole dimensioni, mentre non è ragionevolmente applicabile a sistemi di grandi dimensioni o a sistemi dinamici. Questo problema di controllo dell'accesso si può risolvere solo introducendo ulteriori meccanismi, descritti nel Capitolo 14. Molti linguaggi di programmazione hanno accolto il concetto di monitor descritto in questo paragrafo, tra cui Concurrent Pascal, Mesa, C# (si pronuncia *C-sharp*) e Java, mentre altri, come Erlang, forniscono meccanismi simili di supporto alla concorrenza.

## 6.8 Esempi di sincronizzazione

Si descrivono ora i meccanismi di sincronizzazione forniti dai sistemi operativi Solaris, Windows XP e Linux, e le API Pthreads. Abbiamo prescelto questi tre sistemi perché offrono esempi validi di approcci differenti rispetto alla sincronizzazione del kernel; le API Pthreads

## MONITOR IN JAVA

Per la sincronizzazione dei thread Java fornisce un meccanismo affine a quello del monitor. Ciascun oggetto, in Java, ha associato un singolo lock. Quando si dichiara un metodo **synchronized**, per invocare il metodo su un oggetto occorre possedere il lock dell'oggetto.

Si dichiara **synchronized** un metodo inserendo la parola chiave nella definizione del metodo. Il codice che segue, per esempio, dichiara **metodoSicuro()** come **synchronized**:

```
public class SempliceClasse {
    ...
    public synchronized void metodoSicuro() {
        /* Implementazione di metodoSicuro() */
        ...
    }
}
```

Si supponga di creare un'istanza di **SempliceClasse()**, nel modo seguente:

```
SempliceClasse sc = new SempliceClasse();
```

Per richiamare il metodo **sc.metodoSicuro()** è necessario possedere il lock dell'oggetto istanza **sc**. Se il lock è già proprietà di un thread diverso, il thread che invoca il metodo dichiarato **synchronized** si blocca ed è collocato nella lista d'attesa per il lock. La lista d'attesa è formata dall'insieme di thread che attendono la disponibilità del lock. Se, al momento dell'invocazione di un metodo **synchronized** dell'istanza, il lock è disponibile, il thread chiamante diviene il proprietario del lock dell'oggetto e può accedere al metodo. Il lock ritorna disponibile quando il thread termina l'esecuzione del metodo; un thread in lista d'attesa, quindi, è selezionato come nuovo proprietario del lock.

Java offre inoltre i metodi **wait()** e **notify()**, che funzionano analogamente alle istruzioni **wait()** e **signal()** per i monitor. Nella versione 1.5 la macchina virtuale Java comprende (tra gli altri meccanismi per la concorrenza) la API del package **java.util.concurrent** che mette a disposizione semafori, variabili condizionali e semafori mutex.

sono state incluse nella trattazione perché ampiamente utilizzate dagli sviluppatori per la creazione e la sincronizzazione dei thread su piattaforme UNIX e Linux. Come si vedrà nel corso del paragrafo i metodi di sincronizzazione messi a disposizione dai tre differenti sistemi variano in modo sottile ma significativo.

### 6.8.1 Sincronizzazione in Solaris

Per regolare l'accesso alle sezioni critiche, Solaris mette a disposizione semafori mutex adattivi, variabili condizionali, semafori, lock di lettura-scrittura e i cosiddetti tornelli (*turnstiles*). Si vedano i Paragrafi 6.5 e 6.6 per la descrizione di questi argomenti.

Un **mutex adattivo** (*adaptive mutex*) protegge l'accesso a ogni elemento critico di dati; in un sistema multiprocessore si attiva come un semaforo ordinario realizzato come uno spinlock. Se i dati sono soggetti a lock e quindi già in uso, nel mutex adattivo si possono verificare due situazioni: se i dati sono posseduti da un thread correntemente in esecuzione in un'altra unità d'elaborazione, il thread che ha fatto la nuova richiesta d'accesso entra in uno stato d'attesa attiva (*spinlock*), mentre aspetta la rimozione del lock, poiché è probabile che il thread in possesso dei dati termini la propria elaborazione in breve tempo; viceversa, se quest'ultimo non si trova nello stato d'esecuzione, il thread richiedente si sospende nello sta-

to d'attesa fino alla rimozione del lock. In questo modo si evita il ciclo d'attesa attiva, poiché probabilmente i dati non saranno rilasciati in un tempo ragionevolmente breve. Si ha questa situazione, per esempio, quando il thread che ha bloccato l'accesso ai dati è a sua volta nello stato d'attesa. Poiché un sistema dotato di una singola CPU può eseguire un solo thread alla volta, il thread che possiede il lock non è mai in esecuzione nell'istante in cui un altro thread verifica la presenza del lock. Quindi, in un sistema con una CPU, un thread che incontra un lock sospende la propria esecuzione anziché persistere nel ciclo d'attesa attiva.

Solaris adotta il metodo del mutex adattivo per proteggere soltanto i dati cui si accede da segmenti di codice molto corti; in pratica si usa solo se un lock permane per meno di qualche centinaio di istruzioni. Se il segmento di codice fosse più lungo, l'uso dei cicli d'attesa sarebbe inefficiente. Per segmenti di codice più lunghi il sistema ricorre all'impiego di semafori o variabili condizionali. Se l'oggetto richiesto è già posseduto da altri thread, il thread richiedente invoca una `wait()` e si sospende. Quando il thread che possiede il dato ne rilascia il controllo, invia un segnale al successivo thread presente nella coda d'attesa per quel dato. L'ulteriore costo richiesto dalla sospensione e dalla successiva riattivazione del thread, compresi i relativi cambi di contesto, è sicuramente minore di quello dovuto alle parecchie centinaia di istruzioni che si sprecherebbero nel ciclo d'attesa del semaforo ad attesa attiva.

I lock di lettura-scrittura si usano per proteggere i dati cui si accede spesso, e di solito per la sola lettura. In tali circostanze essi sono più efficienti dei semafori, poiché più thread possono leggere i dati in modo concorrente, mentre un semaforo avrebbe imposto la serializzazione di questi accessi. Poiché la sua realizzazione introduce un costo aggiuntivo, anche i lock di lettura-scrittura si applicano solo alle sezioni di codice lunghe.

Solaris utilizza i cosiddetti tornelli per ordinare la lista dei thread che attendono di ottenere un mutex adattivo o un lock di lettura-scrittura. Un **tornello** (*turnstile*) è una struttura a coda contenente i thread che attendono il rilascio di un lock. Ad esempio, se un thread possiede il lock di un oggetto sincronizzato, tutti gli altri thread che cercano di acquisirlo si bloccano ed entrano nel tornello relativo a quel lock. Quando il lock è rimosso, il kernel seleziona un thread dal tornello per concedergli la proprietà del lock. Ogni oggetto sincronizzato con almeno un thread che attende di acquisire il lock richiede un tornello separato. Tuttavia, anziché associare un tornello a ciascun oggetto sincronizzato, il sistema operativo assegna un tornello a ogni thread a livello kernel.

Il tornello del primo thread che si blocca su un oggetto sincronizzato diventa il tornello per l'oggetto stesso, i thread successivi si aggiungono allo stesso tornello. Quando il thread iniziale infine rilascia il lock, acquisisce un nuovo tornello da una lista di tornelli liberi mantenuta dal kernel. Per prevenire un'inversione delle priorità, i tornelli sono organizzati secondo un **protocollo di ereditarietà delle priorità**. Ciò significa che, se un thread detiene il lock di un oggetto di cui è in attesa un thread a priorità maggiore, il thread a priorità minore eredita temporaneamente la priorità del thread a priorità maggiore. Al rilascio del lock, il thread riassume la priorità originaria.

Si noti che i meccanismi di gestione dei lock usati dal kernel sono disponibili anche per i thread a livello utente, sicché gli stessi tipi di lock sono disponibili sia all'interno sia all'esterno del kernel. Una differenza cruciale nella loro realizzazione è il protocollo di ereditarietà delle priorità: i lock del kernel adottano i metodi di ereditarietà delle priorità del kernel usati dallo scheduler (Paragrafo 19.4); quelli dei thread a livello utente non offrono questa funzionalità.

Per ottimizzare le prestazioni di Solaris, gli sviluppatori hanno via via perfezionato e calibrato l'implementazione dei lock. Poiché i lock si usano frequentemente, e spesso per funzioni cruciali del kernel, la loro ottimizzazione può condurre a notevoli incrementi delle prestazioni.

## 6.8.2 Sincronizzazione in Windows XP

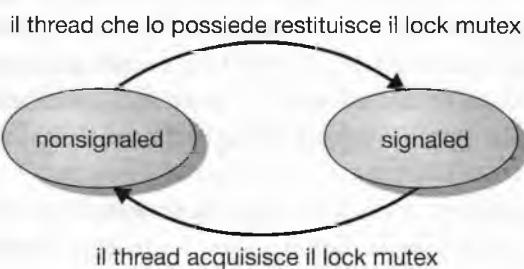
Il sistema operativo Windows XP ha un kernel multithread che offre anche la gestione di applicazioni per le elaborazioni in tempo reale e di architetture multiprocessore. Quando il kernel di Windows XP accede a una risorsa globale in un sistema con singola CPU, disabilita temporaneamente le interruzioni aventi procedure di gestione che potrebbero accedere alla stessa risorsa globale. In un sistema multiprocessore, si protegge l'accesso alle risorse globali con i semafori ad attesa attiva (*spinlock*). Proprio come nel caso di Solaris, il kernel usa i semafori ad attesa attiva solo per proteggere segmenti di codice brevi. Inoltre, per ragioni di efficienza, il kernel impedisce che un thread sia sottoposto a prelazione mentre detiene un semaforo ad attesa attiva.

Per la sincronizzazione fuori dal kernel, il sistema operativo offre gli **oggetti dispatcher**, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi mutex, semafori, eventi e timer. I dati condivisi si possono proteggere richiedendo che un thread entri in possesso di un mutex prima di potervi accedere, e rilasci il mutex al completamento dell'elaborazione di quei dati. Il comportamento dei semafori è illustrato nel Paragrafo 6.5. Gli eventi sono un meccanismo di sincronizzazione utilizzabile in modo simile alle variabili condizionali; cioè, possono notificare il verificarsi di una determinata condizione a un thread che l'attendeva. Infine i timer sono usati per informare un thread (o più di uno) della scadenza di uno specifico periodo di tempo.

Gli oggetti dispatcher possono essere nello stato *signaled* o nello stato *nonsignaled*. Uno **stato signaled** indica che l'oggetto è disponibile e che un thread che tentasse di accedere all'oggetto non sarebbe bloccato; uno **stato nonsignaled** indica che l'oggetto non è disponibile e che qualsiasi thread che tentasse di accedervi sarebbe bloccato. La Figura 6.21 illustra le transizioni di stato di un oggetto dispatcher di tipo lock mutex.

C'è una relazione tra lo stato di un *oggetto dispatcher* e lo stato di un thread. Se un thread si blocca a un *oggetto dispatcher* nello stato *nonsignaled*, il suo stato cambia da pronto per l'esecuzione ad attesa, e il thread viene messo nella coda d'attesa per quell'oggetto. Quando lo stato dell'*oggetto dispatcher* diventa *signaled*, il kernel verifica che non ci sia alcun thread che attende l'oggetto, altrimenti, ne fa passare uno, o più d'uno, dallo stato di attesa allo stato di pronto per l'esecuzione, dal quale può riprendere l'esecuzione. Il numero dei thread che il kernel seleziona dalla coda d'attesa dipende dal tipo di *oggetto dispatcher* al quale attendono. Il kernel selezionerebbe un solo thread dalla coda d'attesa nel caso di un mutex, poiché un oggetto mutex può essere posseduto da un solo thread. Nel caso di un oggetto evento, il kernel selezionerebbe tutti i thread che attendono l'evento stesso.

Consideriamo un lock mutex come esempio per illustrare gli *oggetti dispatcher* e gli stati dei thread. Se cercasse di acquisire un *oggetto dispatcher* di tipo mutex che sia nello stato *nonsignaled*, un thread sarebbe sospeso e messo in una coda d'attesa per l'oggetto mutex. Se il mutex passasse allo stato *signaled* (il risultato del rilascio del lock mutex da parte di un al-



**Figura 6.21** Oggetto dispatcher di tipo mutex.

tro thread), il thread in attesa in testa alla coda del mutex passerebbe dallo stato d'attesa allo stato di pronto per l'esecuzione e acquisirebbe il mutex.

Il progetto di programmazione rispetto al problema del produttore/consumatore alla fine di questo capitolo è espressamente dedicato ai lock mutex e ai semafori nella API di Win32.

### 6.8.3 Sincronizzazione dei processi in Linux

Prima della versione 2.6, Linux adoperava un kernel senza prelazione; ciò significa che non consentiva di applicare la prelazione ai processi eseguiti in modalità di sistema – neppure nel caso che processi con priorità più alta fossero pronti per l'esecuzione. Ora, per contro, il kernel di Linux ha adottato compiutamente il procedimento della prelazione, cosicché i task attivi nel kernel possono essere sottoposti a prelazione.

Il kernel di Linux si serve di spinlock e semafori (nonché della variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel. Sulle macchine SMP, il meccanismo fondamentale è lo spinlock; il kernel è progettato in modo da mantenere attivi gli spinlock solo per brevi periodi di tempo. Sulle macchine monoprocesso, gli spinlock sono inadatti, e si ricorre all'abilitazione e inibizione del diritto di prelazione nel kernel. Su tali macchine, in pratica, anziché attivare uno spinlock, il kernel inibisce la prelazione; anziché rimuovere lo spinlock, abilita la prelazione. In sintesi:

monoprocesso	multiprocesso
Inibisce la prelazione a livello kernel.	Attiva spinlock.
Abilita la prelazione a livello kernel.	Rimuove spinlock.

Il modo impiegato da Linux per abilitare e inibire il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`. Va detto che non è possibile sottoporre il kernel a prelazione se un task attivo nella modalità di sistema possiede un lock. Per aggirare l'ostacolo, ogni task nel sistema possiede una struttura, `thread_info`, in cui un contatore, `preempt_count`, indica il numero dei lock attivi nel sistema. Quando un lock entra in funzione, `preempt_count` aumenta di uno, mentre diminuisce di uno quando ne viene rimosso. Qualora il valore di `preempt_count` per il task in esecuzione sia maggiore di zero, sarebbe rischioso sottoporre a prelazione il kernel, dato che il task possiede un lock. Se il valore è zero, il kernel può subire l'interruzione (assumendo che non vi siano chiamate in sospeso a `preempt_disable()`).

Gli spinlock, insieme all'abilitazione e inibizione della prelazione, sono utilizzati nel kernel solo quando si ricorre per breve tempo a un lock (o all'inibizione della prelazione del kernel). Quando vi sia necessità di mantenere un lock attivo più a lungo, è opportuno utilizzare i semafori.

### 6.8.4 Sincronizzazione in Pthreads

La API Pthreads fornisce lock mutex, variabili condizionali e lock di lettura-scrittura per la sincronizzazione dei thread; è a disposizione dei programmati e non fa parte di questo o quel kernel. I lock mutex rappresentano, in ambiente Pthreads, la tecnica di sincronizzazione fondamentale. La loro finalità è di proteggere le sezioni critiche del codice: un thread che sia in procinto di entrare in una sezione critica si appropria del lock, quindi, al momento di uscirne, lo rimuove. Le variabili condizionali si comportano in Pthreads in maniera molto

simile a quanto descritto nel Paragrafo 6.7. I lock di lettura-scrittura hanno un funzionamento analogo al meccanismo descritto al Paragrafo 6.6.2. Molti sistemi predisposti per l'uso di Pthreads, inoltre, offrono semafori, sebbene non rientrino nello standard Pthreads; appartengono, invece, all'estensione POSIX SEM. Tra le altre estensioni della API Pthreads figurano gli spinlock, malgrado non tutte le estensioni siano ritenute portabili da un'implementazione a un'altra. Vedremo alla fine del capitolo, nel progetto di programmazione, come impiegare lock mutex e semafori in Pthreads.

## 6.9 Transazioni atomiche

---

La mutua esclusione nelle sezioni critiche assicura che siano eseguite in modo atomico. In altri termini, se due sezioni critiche sono eseguite in modo concorrente, il risultato che si ottiene coincide esattamente con quello che si otterrebbe dall'esecuzione sequenziale delle due sezioni critiche in un qualsiasi ordine. Pur essendo utile in molti ambiti, questa proprietà non è più sufficiente nei molti casi in cui si vuole avere la certezza che una sezione critica costituisca una singola unità logica di lavoro, caratterizzata dal fatto di essere eseguita nella sua totalità o non essere eseguita per niente. Un tipico esempio è costituito dal trasferimento di fondi; un'operazione che comporta l'addebito della valuta da un conto e il contemporaneo accredito su un altro. Chiaramente, affinché i dati mantengano la loro coerenza, o si portano a termine con successo entrambe le operazioni oppure non devono avvenire né l'addebito né l'accredito.

La discussione contenuta nel resto del paragrafo è strettamente correlata alla gestione delle **basi di dati**, a cui sono correlati i problemi di archiviazione, recupero e coerenza dei dati. Recentemente si è riscontrato un notevole aumento d'interesse circa l'uso di tecniche proprie delle basi di dati all'interno dei sistemi operativi. I sistemi operativi si possono considerare sistemi per la manipolazione di dati; in questa veste, possono sicuramente trarre vantaggio da tecniche e modelli ottenuti nella ricerca sulle basi di dati. Ad esempio, la flessibilità e la potenza di molti metodi per la gestione dei file usati nei sistemi operativi potrebbero migliorare se si sostituissero con metodi più formali propri delle basi di dati. Dal Paragrafo 6.9.2 al Paragrafo 6.9.4 saranno descritte alcune di queste tecniche e verrà illustrato il loro impiego da parte dei sistemi operativi. Prima di tutto, però, ci occupiamo dell'atomicità delle transazioni in senso generale, il concetto centrale per le tecniche di gestione delle basi di dati.

### 6.9.1 Modello di sistema

Un insieme di istruzioni (operazioni) che esegue una singola funzione logica prende il nome di **transazione**. Uno dei principali motivi per cui si fa ricorso alle transazioni è conservare l'atomicità malgrado la possibilità che si verifichino situazioni anomale all'interno del sistema.

Una transazione è un'unità di programma che accede a elementi contenuti in file residenti nella memoria secondaria ed eventualmente li aggiorna. Per gli scopi della presente trattazione è sufficiente considerare una transazione come una sequenza di operazioni **read** e **write**, terminate da un'operazione **commit** o da **abort**. L'operazione **commit** indica che la transazione è terminata con successo, mentre l'operazione **abort** significa che la transazione è fallita, a causa di qualche errore logico o di un guasto del sistema.

In entrambi i casi, poiché una transazione fallita può aver già alterato i dati ai quali ha avuto accesso, il loro stato potrebbe non coincidere con quello in cui si sarebbero trovati se la transazione fosse stata eseguita in modo atomico. Al fine di assicurare la proprietà di ato-

## MEMORIA TRANSAZIONALE

Con l'emergere dei sistemi multicore si è visto un incremento della pressione per lo sviluppo di applicazioni multithread che traggono vantaggio dalle unità di calcolo multiple. Tuttavia, le applicazioni multithread presentano un rischio maggiore di race condition e situazioni di stallo. Per risolvere questi problemi sono state usate tecniche tradizionali come lock, semafori e monitor. Le **memorie transazionali** forniscono una strategia alternativa per lo sviluppo di applicazioni concorrenti sicure.

Una **transazione di memoria** è una sequenza di operazioni atomiche di lettura-scrittura. Se tutte le operazioni di una transazione sono eseguite, la transazione di memoria viene completa, altrimenti le operazioni devono essere annullate e deve essere ripristinata la situazione precedente l'inizio della transazione. I vantaggi della memoria transazionale possono essere sfruttati mediante l'aggiunta di nuove funzionalità a un linguaggio di programmazione.

Si consideri il seguente esempio. Si supponga di avere a disposizione una funzione `update()` che modifica dati condivisi. Questa funzione potrebbe essere scritta in maniera tradizionale usando i lock:

```
update () {
    acquire();
    /* modifica dati condivisi */
    release();
}
```

Tuttavia, l'utilizzo di meccanismi di sincronizzazione come lock e semafori implica molti potenziali problemi, incluso lo stallo dei processi. Inoltre, in seguito alla crescita del numero di thread, i meccanismi tradizionali basati sui lock non si adattano al meglio.

Un'alternativa consiste nell'aggiungere ai linguaggi di programmazione nuove funzionalità che sfruttano il vantaggio dato dalla memoria transazionale. Nel nostro esempio, supponiamo di aggiungere il costrutto `atomic{S}` che assicura che le operazioni in S siano eseguite come transazione. Potremo riscrivere il metodo `update()` così:

```
update () {
    atomic {
        /* modifica dati condivisi */
    }
}
```

Il vantaggio di utilizzare tale meccanismo al posto dei lock sta nel fatto che è il sistema di memoria transazionale, e non il programmatore, a garantire l'atomicità. Inoltre, il sistema è in grado di identificare le istruzioni nei blocchi atomici che possono essere eseguite in concorrenza, come ad esempio accessi concorrenti in lettura a una variabile condivisa. È comunque certamente possibile per un programmatore identificare queste situazioni e utilizzare i lock di lettura-scrittura, ma il compito si complica al crescere del numero di thread in un'applicazione.

La memoria transazionale può essere implementata via software oppure via hardware. La memoria transazionale software (STM), come il nome suggerisce, implementa la memoria transazionale esclusivamente via software, senza la necessità di hardware particolare. In questo schema si inserisce del codice ausiliario nelle transazioni. Esso è prodotto e inserito da un compilatore e gestisce ciascuna transazione esaminando quali istruzioni possono essere eseguite in concorrenza, e quando sono necessari dei lock a basso livello. La memoria transazionale hardware (HTM piccola) utilizza gerarchie di cache hardware e protocolli di coerenza della cache per gestire e risolvere conflitti riguardanti dati condivisi residenti in memorie cache di processori distinti. HTM non richiede una particolare strumentazione software e ha quindi un minor overhead rispetto a STM. Tuttavia, richiede che le gerarchie di cache esistenti e i protocolli di coerenza della cache siano modificati per il supporto di memorie transazionali.

Le memorie transazionali esistono da diversi anni, ma per diverso tempo non hanno conosciuto una vasta diffusione. Soltanto di recente il crescente utilizzo dei sistemi multicore e la maggior enfasi sulla programmazione concorrente hanno focalizzato molta ricerca su questo settore, sia da parte delle istituzioni accademiche sia da parte dei produttori di hardware, inclusi Intel e Sun Microsystems.

micità, la terminazione anomala di una transazione non deve produrre alcun effetto sullo stato dei dati che questa ha già modificato.

Quindi, è necessario ripristinare lo stato dei dati adoperati dalla transazione fallita, riportandolo a quello che li caratterizzava appena prima dell'inizio della transazione (**roll back**). Il rispetto di questa proprietà deve essere garantito dal sistema.

Per stabilire il modo in cui un sistema deve garantire l'atomicità delle proprie transazioni, è necessario identificare le proprietà dei dispositivi che si usano per memorizzare i dati ai quali esse accedono. I diversi tipi di dispositivi di memorizzazione si possono caratterizzare secondo capacità, velocità e attitudine al recupero dai guasti.

- ◆ **Memorie volatili.** Le informazioni registrate nelle memorie volatili, per esempio la memoria centrale o le cache, di solito non sopravvivono ai crolli del sistema. L'accesso a questo tipo di dispositivi è molto rapido, grazie sia alla velocità intrinseca degli accessi alla memoria sia alla possibilità di accedere in modo diretto ai dati.
- ◆ **Memorie non volatili.** Le informazioni registrate in memorie non volatili, per esempio dischi e nastri magnetici, di solito sopravvivono ai crolli del sistema. I dischi sono più affidabili della memoria centrale, ma meno dei nastri magnetici. Sia i dischi sia i nastri sono soggetti a guasti che possono causare anche la perdita dei dati. Poiché dischi e nastri sono dispositivi elettromeccanici che richiedono movimenti fisici per accedere ai dati, attualmente i tempi d'accesso alle memorie non volatili superano di diversi ordini di grandezza quelli alle memorie volatili.
- ◆ **Memorie stabili.** Le informazioni contenute nelle memorie stabili per definizione non si perdono *mai* (data l'impossibilità teorica di garantire questa proprietà, è però indispensabile considerare quest'affermazione con una certa cautela). Per realizzare un'approssimazione di questi dispositivi è necessario duplicare le informazioni in più memorie non volatili (di solito dischi) con tipi di guasto indipendenti, e aggiornare i dati in maniera controllata (si veda in proposito il Paragrafo 12.8).

Ci limitiamo qui a descrivere come sia possibile assicurare l'atomicità di una transazione in un ambiente in cui eventuali guasti comporterebbero la perdita delle informazioni contenute in memorie volatili.

### 6.9.2 Ripristino basato sulla registrazione delle modifiche

Un modo per assicurare l'atomicità è registrare in memorie stabili le informazioni che descrivono tutte le modifiche che la transazione ha apportato ai dati a cui ha avuto accesso. In tal senso, il metodo più largamente usato è quello della **registrazione con scrittura anticipata** (*write-ahead logging*); il sistema mantiene, nella memoria stabile, una struttura dati chiamata **log**, in cui ciascun elemento descrive una singola operazione **write** eseguita dalla transazione ed è composto dei seguenti campi:

- ◆ **nome della transazione.** Il nome, unico, della transazione che ha richiesto l'operazione **write**;
- ◆ **nome del dato modificato.** Il nome, unico, del dato scritto dall'operazione **write**;
- ◆ **valore precedente.** Il valore posseduto dall'elemento prima dell'operazione **write**;
- ◆ **nuovo valore.** Il valore che l'elemento avrà una volta terminata l'operazione **write**.

Esistono altri elementi speciali del log, usati per registrare gli eventi significativi che si possono verificare durante l'elaborazione della transazione, per esempio avvio, successo o fallimento della transazione.

Prima dell'avvio di una transazione  $T_i$  si registra l'elemento  $\langle T_i, \text{start} \rangle$  nel log; durante l'esecuzione, ciascuna operazione `write` di  $T_i$  è *preceduta* dalla scrittura dell'apposito nuovo elemento nel log. Il successo di  $T_i$  è sancito dalla registrazione dell'elemento  $\langle T_i, \text{commit} \rangle$  nel log.

Poiché le informazioni contenute nel log servono per la ricostruzione dello stato delle strutture dati alle quali le diverse transazioni hanno avuto accesso, non è ammissibile che l'effettivo aggiornamento di un componente di queste strutture avvenga prima che il corrispondente elemento del log sia stato registrato nell'apposito dispositivo di memoria stabile. Quindi, il requisito fondamentale affinché questo metodo abbia successo è che l'elemento relativo a un componente  $x$  sia registrato nella memoria stabile prima dell'esecuzione dell'operazione `write(x)`.

Naturalmente, questo metodo determina una penalizzazione delle prestazioni, poiché ogni `write` logica richiede in realtà l'esecuzione di due `write` fisiche. Inoltre, aumenta la quantità di memoria secondaria usata poiché, oltre allo spazio necessario al contenimento dei dati, si deve prevedere lo spazio necessario al log. In situazioni in cui i dati sono molto importanti e si deve disporre di una rapida funzione di ripristino, tale onere merita di essere sostenuto.

Mediante l'uso dei log il sistema può gestire qualsiasi malfunzionamento, purché non sia una perdita delle informazioni contenute nella memoria non volatile. L'algoritmo di ripristino impiega le due seguenti procedure:

- ◆ `undo( $T_i$ )`, per ripristinare il precedente valore di tutti i dati modificati dalla transazione  $T_i$ ;
- ◆ `redo( $T_i$ )`, per assegnare il nuovo valore a tutti i dati modificati dalla transazione  $T_i$ .

L'insieme dei dati aggiornati da  $T_i$ , con i valori vecchi e nuovi associati, è reperibile nel log.

Per garantirne un corretto comportamento anche se si dovesse verificare un guasto durante il procedimento di ripristino, le operazioni `undo` (annullamento) e `redo` (ripetizione) devono essere idempotenti (in altri termini, le esecuzioni multiple di un'operazione devono produrre i medesimi risultati di una singola esecuzione).

Se una transazione  $T_i$  termina in modo anormale (fallisce), per ripristinare lo stato dei dati modificati è sufficiente eseguire l'operazione `undo( $T_i$ )`. Se si verifica un guasto nel sistema, il ripristino di uno stato corretto comporta la consultazione del log per determinare quali transazioni annullare e quali ripetere. Tale classificazione delle transazioni avviene secondo i seguenti parametri:

- ◆ la transazione  $T_i$  deve essere annullata se il log contiene l'elemento  $\langle T_i, \text{start} \rangle$  ma non l'elemento  $\langle T_i, \text{commit} \rangle$ ;
- ◆ la transazione  $T_i$  deve essere ripetuta se il log contiene sia l'elemento  $\langle T_i, \text{start} \rangle$  sia l'elemento  $\langle T_i, \text{commit} \rangle$ .

### 6.9.3 Punti di verifica

Quando si verifica un guasto nel sistema è necessario consultare il log per determinare quali transazioni annullare e quali ripetere. Questo metodo richiede la scansione dell'intero log e presenta principalmente due inconvenienti.

1. La ricerca può richiedere un tempo piuttosto lungo.
2. La maggior parte delle transazioni, che secondo questo algoritmo deve essere ripetuta, ha già aggiornato con successo dati che, secondo il log, si dovrebbero ancora modificare. Benché la ripetizione delle modifiche non causi alcun danno (per l'idempotenza dell'operazione `redo`), il processo di ripristino richiederà senza dubbio più tempo.

Per ridurre questo genere di sprechi si introduce il concetto di **punto di verifica** (*checkpoint*). Durante l'esecuzione il sistema esegue la registrazione con scrittura anticipata e registrazioni periodiche, che costituiscono ciascun punto di verifica, definite dall'esecuzione della seguente sequenza di azioni:

1. registrazione in una memoria stabile di tutti gli elementi del log correntemente residenti in memorie volatili (di solito la memoria centrale);
2. registrazione in una memoria stabile di tutti i dati modificati correntemente residenti in memorie volatili;
3. registrazione dell'elemento <checkpoint> nel log residente nella memoria stabile.

La presenza dell'elemento <checkpoint> consente al sistema di rendere più efficiente la propria procedura di ripristino. Si consideri una transazione  $T_i$  terminata con successo (*committed*) prima dell'ultimo punto di verifica; l'elemento < $T_i$ , commit> precede nel log l'elemento <checkpoint>; quindi, qualsiasi modifica apportata da  $T_i$  è stata sicuramente registrata nella memoria stabile prima del punto di verifica o come parte del punto di verifica stesso; quindi durante un eventuale ripristino sarebbe inutile eseguire l'operazione redo sulla transazione  $T_i$ .

Quest'osservazione consente di migliorare il precedente algoritmo di ripristino: in seguito a un malfunzionamento, la procedura di ripristino esamina il log per determinare la più recente transazione  $T_i$  che ha iniziato la propria esecuzione prima del più recente punto di verifica; scandisce a ritroso il log fino al primo elemento <checkpoint> e quindi fino al primo elemento < $T_i$ , start>.

Una volta identificata la transazione  $T_i$ , le operazioni redo e undo si devono applicare solamente a  $T_i$  e a tutte le transazioni  $T_j$  cominciate dopo  $T_i$ , ignorando il resto del log. Detto  $T$  l'insieme di queste transazioni, le operazioni necessarie al completamento del ripristino sono le seguenti:

- esecuzione dell'operazione redo( $T_k$ ) per tutte le transazioni  $T_k$  appartenenti a  $T$  tali che il log contiene l'elemento < $T_k$ , commit>;
- esecuzione dell'operazione undo( $T_k$ ) per tutte le transazioni  $T_k$  appartenenti a  $T$  tali che il log non contiene l'elemento < $T_k$ , commit>.

#### 6.9.4 Transazioni atomiche concorrenti

Abbiamo finora considerato un ambiente in cui sia eseguibile solo una transazione per volta. Passiamo ora al caso in cui transazioni multiple siano attive concorrentemente. Poiché le transazioni sono atomiche, il risultato dell'esecuzione concorrente di più transazioni deve essere equivalente a quello che si otterrebbe eseguendo le transazioni in una sequenza arbitraria. Questa caratteristica di **serializzabilità** si può rispettare semplicemente eseguendo ciascuna transazione all'interno di una sezione critica. In altre parole, tutte le transazioni condividono un semaforo **mutex** inizializzato a 1; la prima operazione che una transazione esegue è **wait(mutex)**, mentre l'ultima operazione che si esegue dopo il successo o il fallimento della transazione è **signal(mutex)**.

Questo schema assicura l'atomicità di tutte le transazioni che si eseguono in modo concorrente, ma è troppo restrittivo; in molte situazioni si può consentire la sovrapposizione dell'esecuzione di più transazioni, pur rispettando la proprietà di serializzabilità; nel seguito si descrivono alcuni **algoritmi di controllo della concorrenza** che assicurano la serializzabilità.

$T_0$	$T_1$
read( $A$ )	
write( $A$ )	
read( $B$ )	
write( $B$ )	
	read( $A$ )
	write( $A$ )
	read( $B$ )
	write( $B$ )

Figura 6.22 Sequenza d'esecuzione 1: sequenza d'esecuzione seriale in cui  $T_0$  è seguita da  $T_1$ .

#### 6.9.4.1 Serializzabilità

Si consideri un sistema contenente due oggetti  $A$  e  $B$ , entrambi letti e modificati da due transazioni  $T_0$  e  $T_1$ , e si supponga che le due transazioni siano eseguite in modo atomico, nell'ordine  $T_0$ ,  $T_1$ . Questa **sequenza d'esecuzione** (*schedule*) è rappresentata nella Figura 6.22. Nella sequenza d'esecuzione 1 le istruzioni rispettano un ordinamento cronologico dall'alto verso il basso, con le istruzioni di  $T_0$  disposte nella colonna di sinistra e quelle di  $T_1$  nella colonna di destra.

Una sequenza d'esecuzione in cui ciascuna transazione sia eseguita in modo atomico si chiama **sequenza d'esecuzione seriale**; è composta da una sequenza di istruzioni appartenenti a transazioni diverse, caratterizzata dal fatto che tutte le istruzioni appartenenti a una singola transazione sono raggruppate. Quindi, dato un insieme di  $n$  transazioni, esistono  $n!$  differenti sequenze d'esecuzione seriali valide. Ogni sequenza d'esecuzione seriale è valida, poiché equivale all'esecuzione atomica in un ordine qualsiasi delle transazioni in essa contenute.

Se si consente a due transazioni di sovrapporre le proprie esecuzioni, la sequenza d'esecuzione risultante non è più seriale. Il che non implica necessariamente che l'esecuzione risultante sia scorretta (cioè non equivalente a una sequenza d'esecuzione seriale). Per dimostrare quest'affermazione è necessario definire la nozione di **operazioni conflittuali**.

Si consideri una sequenza d'esecuzione  $S$  in cui compaiono in successione due operazioni  $O_i$  e  $O_j$  appartenenti rispettivamente alle transazioni  $T_i$  e  $T_j$ . Se tentano di accedere agli stessi dati e se almeno una tra le due è una `write`, le operazioni  $O_i$  e  $O_j$  sono *conflittuali*. Al fine di illustrare meglio il concetto di operazioni conflittuali, si consideri la sequenza d'esecuzione 2, non seriale, nella Figura 6.23. L'operazione `write(A)` di  $T_0$  è in conflitto con

$T_0$	$T_1$
read( $A$ )	
write( $A$ )	
	read( $A$ )
	write( $A$ )
read( $B$ )	
write( $B$ )	
	read( $B$ )
	write( $B$ )

Figura 6.23 Sequenza d'esecuzione 2: sequenza d'esecuzione concorrente serializzabile.

`read(A)` di  $T_1$ ; viceversa, l'operazione `write(A)` di  $T_1$  non è in conflitto con `read(B)` di  $T_0$ , poiché le due operazioni accedono a elementi diversi.

Si supponga che  $O_i$  e  $O_j$  siano due operazioni concatenate all'interno di una sequenza d'esecuzione  $S$ ; se  $O_i$  e  $O_j$  appartengono a transazioni diverse e non conflittuali si può invertirne l'ordine d'esecuzione, producendo una nuova sequenza d'esecuzione  $S'$ . Ci si aspetta che  $S$  ed  $S'$  siano equivalenti, poiché tutte le operazioni rispettano il medesimo ordine in entrambe le sequenze d'esecuzione, a eccezione delle sole  $O_i$  e  $O_j$  il cui ordine non è rilevante ai fini del risultato finale.

Per illustrare meglio il concetto di inversione dell'ordine delle operazioni si consideri la sequenza d'esecuzione 2 della Figura 6.23. Poiché l'operazione `write(A)` di  $T_1$  non è in conflitto con l'operazione `read(B)` di  $T_0$ , dallo scambio della loro posizione si ottiene una sequenza d'esecuzione equivalente: indipendentemente dallo stato iniziale, entrambe le sequenze d'esecuzione producono il medesimo risultato. Proseguendo nello scambio delle operazioni non conflittuali si ottiene ciò che segue:

- ◆ scambio dell'operazione `read(B)` di  $T_0$  con l'operazione `read(A)` di  $T_1$ ;
- ◆ scambio dell'operazione `write(B)` di  $T_0$  con l'operazione `write(A)` di  $T_1$ ;
- ◆ scambio dell'operazione `write(B)` di  $T_0$  con l'operazione `read(A)` di  $T_1$ .

Il risultato finale di questo procedimento è la sequenza d'esecuzione 1 della Figura 6.22, una sequenza d'esecuzione seriale. Questo ragionamento ha dimostrato come la sequenza d'esecuzione 2 sia equivalente a una sequenza d'esecuzione seriale e implica che, indipendentemente dallo stato iniziale del sistema, la sequenza d'esecuzione 2 produce il medesimo stato finale determinato da una qualsiasi sequenza d'esecuzione seriale equivalente.

Se una sequenza d'esecuzione  $S$  si può trasformare in una sequenza d'esecuzione seriale  $S'$  con una serie di scambi tra operazioni non conflittuali, si dice che  $S$  è in **conflitto serializzabile**. Quindi, la sequenza d'esecuzione 2 è in conflitto serializzabile, poiché si può trasformare nella sequenza d'esecuzione seriale 1.

#### 6.9.4.2 Protocolli per la gestione dei lock

Uno dei metodi che si usano per garantire la serializzabilità consiste nell'associare un lock a ciascun dato, e richiedere che ogni transazione rispetti il **protocollo per la gestione dei lock** (*locking protocol*), che governa l'acquisizione e il rilascio dei lock. Si può applicare un lock a un dato in diversi modi, due dei quali sono i seguenti.

- ◆ **Condiviso.** Se una transazione  $T_i$  ottiene il lock di  $Q$  in modo condiviso,  $T_i$  può leggere, ma non scrivere nell'elemento (tale forma si denota con  $S$ ).
- ◆ **Esclusivo.** Se una transazione  $T_i$  ottiene il lock di  $Q$  in modo esclusivo,  $T_i$  può leggere e scrivere nell'elemento (tale forma si denota con  $X$ ).

Ogni transazione deve richiedere il lock di un elemento  $Q$  nel modo adeguato al tipo di operazione che deve eseguire su di esso.

Per accedere a un elemento  $Q$ , una transazione  $T_i$  deve innanzitutto eseguire il lock di  $Q$  in modo adeguato. Se il lock di  $Q$  è attualmente disponibile, la sua acquisizione da parte di  $T_i$  è garantita; se però il lock di  $Q$  è posseduto da un'altra transazione,  $T_i$  dovrà attendere il rilascio. In particolare, se  $T_i$  richiede un lock esclusivo per  $Q$ , deve attendere che la transazione che ne è attualmente in possesso lo rilasci. Se, invece, il lock richiesto è di tipo condiviso,  $T_i$  deve aspettare il rilascio della risorsa solo se  $Q$  è attualmente soggetto a un lock esclusivo; altrimenti ottiene anch'essa il diritto di accedere a  $Q$ . Si noti la somiglianza tra questo schema e l'algoritmo dei lettori-scrittori trattato nel Paragrafo 6.6.2.

Una transazione può rilasciare il lock di un elemento precedentemente acquisito, anche se deve in ogni caso mantenerlo per tutto il periodo in cui accede all'elemento. Inoltre, non è sempre auspicabile che una transazione rilasci il lock di un elemento immediatamente dopo il suo ultimo accesso, poiché in questo modo la serializzabilità della transazione potrebbe non essere garantita.

Un protocollo che assicura la serializzabilità è il cosiddetto **protocollo per la gestione dei lock a due fasi**, che esige che ogni transazione richieda l'esecuzione delle operazioni di lock e di rilascio (*unlock*) in due fasi distinte:

- ◆ **fase di crescita.** Una transazione può ottenere nuovi lock sui dati, ma non rilasciarne alcuno in suo possesso;
- ◆ **fase di riduzione.** Una transazione può rilasciare lock sui dati di cui è in possesso, ma non ottenerne di nuovi.

Inizialmente, una transazione si trova nella fase di crescita e acquisisce i lock sui dati necessari; nel rilasciarne uno, la transazione entra nella fase di riduzione e non ne può richiedere di nuovi.

Il protocollo per la gestione dei lock a due fasi garantisce la serializzabilità dei conflitti (Esercizio 6.35), ma non elimina la possibilità di situazioni di stallo. Inoltre può accadere che, dato un insieme di transazioni, esistano sequenze d'esecuzione in conflitto serializzabile che tuttavia non si possono ottenere attraverso il protocollo per la gestione dei lock a due fasi. A ogni modo, per migliorare le prestazioni di questo protocollo è indispensabile disporre di ulteriori informazioni relative alle transazioni, oppure imporre una qualche struttura o un ordinamento dell'insieme dei dati.

#### 6.9.4.3 Protocolli basati sulla marcatura temporale

Nei precedenti protocolli per la gestione dei lock l'ordinamento seguito da ogni coppia di transazioni conflittuali è determinato nella fase d'esecuzione dal primo lock sui dati richiesto da entrambe e che comporta modi incompatibili. Un altro metodo per determinare l'ordine di serializzabilità consiste nella scelta anticipata di un ordinamento delle transazioni. Il metodo più comunemente adottato consiste nell'usare uno schema con **ordinamento a marche temporali** (*timestamp ordering*).

A ciascuna transazione  $T_i$  nel sistema si associa una marca temporale (*timestamp*) unica individuata da  $\text{TS}(T_i)$ ; il sistema assegna la marca temporale prima che la transazione  $T_i$  inizi la propria esecuzione. Se una transazione  $T_i$  riceve una marca temporale  $\text{TS}(T_i)$  e si presenta una nuova transazione  $T_j$ , allora  $\text{TS}(T_i) < \text{TS}(T_j)$ . Per realizzare questo schema esistono due semplici metodi, che sono i seguenti.

- ◆ Adoperare come marca temporale il valore dell'orologio di sistema; in altre parole, la marca temporale di una transazione equivale al valore dell'orologio nell'istante in cui si presenta nel sistema. Questo metodo non funziona per transazioni che avvengono in sistemi separati o con unità d'elaborazione che non condividono lo stesso orologio.
- ◆ Adoperare come marca temporale un contatore logico; in altri termini, la marca temporale di una transazione equivale al valore del contatore nell'istante in cui essa si presenta nel sistema. Il contatore s'incrementa dopo l'assegnazione di una nuova marca temporale.

Le marche temporali delle transazioni determinano l'ordine di serializzabilità. Quindi, se  $\text{TS}(T_i) < \text{TS}(T_j)$ , il sistema deve garantire che la sequenza d'esecuzione generata sia equivalente alla sequenza d'esecuzione seriale in cui la transazione  $T_i$  appare prima della transazione  $T_j$ .

Per realizzare questo schema a ogni elemento  $Q$  si associano due valori di marche temporali:

- ◆ **R-timestamp( $Q$ )**, che denota la maggiore tra le marche temporali di tutte le transazioni che hanno completato con successo un'operazione `read( $Q$ )`.
- ◆ **W-timestamp( $Q$ )**, che denota la maggiore tra le marche temporali di tutte le transazioni che hanno completato con successo un'operazione `write( $Q$ )`.

Queste marche temporali si aggiornano dopo l'esecuzione di ogni nuova istruzione `read( $Q$ )` o `write( $Q$ )`.

Il protocollo a ordinamento di marche temporali garantisce che l'esecuzione di tutte le operazioni conflittuali `read` e `write` rispetti l'ordinamento stabilito dalle marche temporali e funziona nel modo seguente.

- ◆ Si supponga che la transazione  $T_i$  sottoponga una `read( $Q$ )`.
  - ◊ Se  $\text{TS}(T_i) < \text{W-timestamp}(\mathcal{Q})$ , questo stato implica che  $T_i$  deve leggere un valore di  $Q$  che è già stato sovrascritto; quindi, si rifiuta l'operazione `read` e si annulla  $T_i$  ripristinando lo stato iniziale (*rollback*).
  - ◊ Se  $\text{TS}(T_i) \geq \text{W-timestamp}(\mathcal{Q})$ , si esegue l'operazione `read` e a R-timestamp( $Q$ ) si assegna il massimo tra R-timestamp( $Q$ ) e  $\text{TS}(T_i)$ .
- ◆ Si supponga che la transazione  $T_i$  sottoponga una `write( $Q$ )`.
  - ◊ Se  $\text{TS}(T_i) < \text{R-timestamp}(\mathcal{Q})$ , questo stato implica che il valore di  $Q$  che  $T_i$  sta producendo era necessario in precedenza e che  $T_i$  aveva supposto che tale valore non sarebbe mai stato prodotto. Quindi si rifiuta l'operazione `write` e si annulla  $T_i$  ripristinando lo stato iniziale.
  - ◊ Se  $\text{TS}(T_i) < \text{W-timestamp}(\mathcal{Q})$ , questo stato implica che  $T_i$  sta cercando di scrivere un valore di  $Q$  obsoleto. Quindi si rifiuta l'operazione `write` e si annulla  $T_i$  ripristinando lo stato iniziale.
  - ◊ Altrimenti, si esegue l'operazione `write`.

Ogni transazione  $T_i$  per cui lo schema di controllo della concorrenza ripristina lo stato iniziale come risultato della sottomissione di un'operazione `read` o `write`, riceve una nuova marca temporale e viene riavviata.

Per illustrare questo protocollo si consideri la sequenza d'esecuzione 3 di due transazioni  $T_2$  e  $T_3$  proposta nella Figura 6.24. Si assume che ciascuna transazione riceva la propria marca temporale immediatamente prima dell'esecuzione della sua prima istruzione. Quindi, nella sequenza d'esecuzione 3,  $\text{TS}(T_2) < \text{TS}(T_3)$ , e la presente sequenza d'esecuzione è consentita dal protocollo basato sulla marcatura temporale.

$T_2$	$T_3$
read ( $B$ )	
	read ( $B$ )
read ( $A$ )	write ( $B$ )
	read ( $A$ )
	write ( $A$ )

**Figura 6.24** Sequenza d'esecuzione 3: sequenza d'esecuzione possibile con il protocollo basato sulla marcatura temporale.

Questa sequenza d'esecuzione può essere generata anche dal protocollo per la gestione dei lock a due fasi; in ogni caso, certe sequenze d'esecuzione sono possibili con tale protocollo, ma non col protocollo basato sulla marcatura temporale, e viceversa.

Il protocollo a ordinamento di marche temporali garantisce la serializzabilità del conflitto. Questa proprietà deriva dal fatto che le operazioni conflittuali si elaborano secondo l'ordine stabilito dalle marche temporali. Il protocollo assicura inoltre l'assenza di situazioni di stallo, poiché nessuna transazione rimane in attesa.

## 6.10 Sommario

---

Dato un gruppo di processi sequenziali cooperanti che condividono dati, è necessario garantirne la mutua esclusione. Si tratta di assicurare che una sezione critica di codice sia utilizzabile da un solo processo o thread alla volta. Di solito l'hardware di un calcolatore fornisce diverse operazioni che assicurano la mutua esclusione, ma per la maggior parte dei programmatore queste soluzioni hardware sono troppo complicate da utilizzare. I semafori rappresentano una soluzione a questo problema. I semafori consentono di superare questa difficoltà; sono utilizzabili per risolvere diversi problemi di sincronizzazione e sono realizzabili in modo efficiente, soprattutto se è disponibile un'architettura che permette le operazioni atomiche.

Sono stati presentati diversi problemi di sincronizzazione – come il problema dei produttori e consumatori con memoria limitata, dei lettori-scrittori e dei cinque filosofi – che costituiscono esempi rappresentativi di una vasta classe di problemi di controllo della concorrenza. Questi problemi sono stati usati per verificare quasi tutti gli schemi di sincronizzazione proposti.

Il sistema operativo deve fornire mezzi di protezione contro gli errori di sincronizzazione. A tal fine sono stati proposti parecchi costrutti di linguaggio. I monitor offrono un meccanismo di sincronizzazione per la condivisione di tipi di dati astratti. Una variabile condizionale consente a una procedura di monitor di sospendere la propria esecuzione finché non riceve un segnale.

Solaris, Windows XP e Linux sono esempi di sistemi operativi moderni che offrono vari meccanismi come semafori, mutex, spinlock e variabili condizionali per il controllo dell'accesso ai dati condivisi. La API Pthreads fornisce supporto a mutex e variabili condizionali.

Una transazione è un'unità di programma da eseguire in modo atomico, cioè le operazioni a essa associate vanno eseguite nella loro totalità o non si devono eseguire per niente. Per assicurare la proprietà di atomicità anche nel caso di malfunzionamenti, si può usare la registrazione con scrittura anticipata. Si registrano tutti gli aggiornamenti nel log, che si mantiene in una memoria stabile. Se si verifica un crollo del sistema, si usano le informazioni conservate nel log per ripristinare lo stato dei dati aggiornati; tale ripristino si esegue usando le operazioni `undo` e `redo`. Per ridurre il carico dovuto alla ricerca nel log dopo un malfunzionamento del sistema è utilizzabile un metodo basato su punti di verifica.

Per assicurare una corretta esecuzione si deve usare uno schema di controllo della concorrenza che garantisca la serializzabilità. Esistono diversi schemi di controllo della concorrenza che assicurano la serializzabilità differendo un'operazione o arrestando la transazione che ha richiesto l'operazione. I più diffusi sono i protocolli per la gestione dei lock e gli schemi d'ordinamento a marche temporali.

## Esercizi pratici

- 6.1 Nel Paragrafo 6.4 si afferma che disabilitando le interruzioni si può influenzare l'orologio di sistema. Spiegate perché ciò può succedere e come tali effetti possono essere mitigati.
- 6.2 Il problema del fumatore di sigarette. Si consideri un sistema con tre processi *fumatore* e un processo *agente*. Ogni fumatore continua a ripetere le operazioni di arrotolarsi una sigaretta e fumarla. Per fare una sigaretta e per fumarla il fumatore ha bisogno di tre elementi: tabacco, cartina e fiammiferi. Uno dei processi fumatore ha le cartine, un altro ha il tabacco e il terzo ha i fiammiferi. L'agente ha una disponibilità infinita delle tre risorse. L'agente mette sul tavolo due dei tre ingredienti e il fumatore in possesso del terzo ingrediente si arrotola una sigaretta e la fuma, segnalando all'agente il completamento del suo incarico. L'agente mette quindi sul tavolo altri due elementi e il ciclo si ripete. Scrivete un programma per sincronizzare agente e fumatori usando la sincronizzazione in Java.
- 6.3 Spiegate perché Solaris, Windows XP e Linux implementano meccanismi di bloccaggio multipli. Descrivete le circostanze in cui questi sistemi operativi utilizzano spin-lock, mutex, semafori, mutex adattivi e variabili condition. Spiegate, in ciascun caso, perché occorre un tale meccanismo.
- 6.4 Descrivete come variano in termini di costo i dispositivi di memoria volatili, non volatili e stabili.
- 6.5 Illustrate il compito dei checkpoint. Quanto spesso i checkpoint devono essere eseguiti? Descrivete come la frequenza di checkpoint influisca su:
  - prestazioni del sistema, nel caso in cui non vi siano errori;
  - tempo necessario per il ripristino in seguito a un crash del sistema;
  - tempo necessario per il ripristino in seguito a un crash del disco.
- 6.6 Spiegate il concetto di atomicità di una transazione.
- 6.7 Mostrate che certe sequenze di esecuzione sono possibili con il protocollo per la gestione dei lock a due fasi, ma non con il protocollo basato su marcatura temporale, e viceversa.

## Esercizi

- 6.8 Le race condition sono possibili in diversi sistemi. Si consideri un sistema bancario con due funzioni: *deposit(amount)* e *withdraw(amount)*. Le due funzioni ricevono in ingresso l'importo (*amount*) che deve essere depositato (*deposit*) o prelevato (*withdraw*) da un conto corrente bancario. Assumete che un conto corrente sia condiviso tra marito e moglie, e che in maniera concorrente il marito chiami la funzione *withdraw()* e la moglie la funzione *deposit()*. Descrivete com'è possibile il verificarsi di una race condition e che cosa dovrebbe essere fatto per evitarla.
- 6.9 L'algoritmo seguente, concepito da Dekker, è la prima soluzione programmata conosciuta del problema della sezione critica per due processi. I due processi,  $P_0$  e  $P_1$ , condividono le seguenti variabili:

```
boolean flag[2]; /* inizialmente falsa* /
int turno;
```

```

do {
    flag[i] = true;

    while (flag[j]) {
        if (turno == j) {
            flag[i] = false;
            while (turno == j)
                ;// non fa niente
            flag[i] = true;
        }
    }
}

sezione critica

turno = j;
flag[i] = false;

sezione non critica

} while (true);

```

**Figura 6.25** Struttura del processo  $P_i$  nell'algoritmo di Dekker.

La struttura del processo  $P_i$  ( $i == 0$  oppure  $1$ ), dove  $P_j$  ( $j == 1$  oppure  $0$ ) è l'altro processo, è mostrata nella Figura 6.25. Dimostrate che l'algoritmo soddisfa tutti e tre i requisiti per il problema della sezione critica.

- 6.10 La prima soluzione programmata corretta del problema della sezione critica per  $n$  processi con un limite di  $n - 1$  turni d'attesa è stata proposta da Eisenberg e McGuire. I processi condividono le seguenti variabili:

```

enum statoP {inattivo, rich_in, in_sc};
statoP flag[n];
int turno;

```

Ciascun elemento di `pronto` è inizialmente `inattivo`; il valore iniziale di `turno` è irrilevante (compreso tra  $0$  e  $n - 1$ ). La struttura del processo  $P_i$  è illustrata nella Figura 6.26. Dimostrate che tale algoritmo soddisfa tutti e tre i requisiti del problema della sezione critica.

- 6.11 Qual è il significato della locuzione *attesa attiva*? Esistono attese di altro genere in un sistema operativo? È possibile evitare completamente l'attesa attiva? Corredate di motivazione le vostre risposte.
- 6.12 Spiegate perché gli spinlock non siano adatti ai sistemi monoprocesso, ma siano spesso usati nei sistemi multiprocessore.
- 6.13 Chiarite perché, per implementare le primitive di sincronizzazione, non è corretto disabilitare le interruzioni di un sistema monoprocesso se le primitive stesse sono destinate a programmi utenti.
- 6.14 Spiegate perché le interruzioni non costituiscono un metodo appropriato per implementare le primitive di sincronizzazione nei sistemi multiprocessore.

```

do {

    while (true) {
        flag[i] = rich_in;
        j = turno;

        while (j != i) {
            if (flag[j] != inattivo)
                j = turno;
            else
                j = (j + 1) % n;
        }
        flag[i] = in_sc;
        j = 0;

        while ((j < n) && (j == i) || flag[i] != in_sc))
            j++;
        if ((j >= n) && (turno == i || flag[turno] == inattivo))
            break;
    }
}

```

*sezione critica*

```

j = (turno + 1) % n;

while (flag[j] == inattivo)
    j = (j + 1) % n;

turno = j;
flag[i] = inattivo;

```

*sezione non critica*

```

} while (true);

```

**Figura 6.26** Struttura del processo  $P_i$  nell'algoritmo di Eisenberg e McGuire.

- 6.15 Descrivete due strutture dati di un kernel in cui possono verificarsi le cosiddette race condition. Assicuratevi di includere una descrizione della modalità in cui queste situazioni possono verificarsi.
- 6.16 Descrivete come l'istruzione `swap()` consenta di ottenere una mutua esclusione che soddisfa il requisito dell'attesa limitata.
- 6.17 I server possono essere progettati in modo da limitare il numero di connessioni aperte. In un dato momento, per esempio, un server può ammettere solo  $N$  connessioni socket per volta; dopo questo limite, il server non accetterà alcuna connessione entrante prima che una connessione esistente sia chiusa. Spiegate come il server possa usare i semafori per limitare il numero delle connessioni concorrenti.
- 6.18 Si dimostri che, se le operazioni `wait()` e `signal()` dei semafori non sono eseguite in modo atomico, la mutua esclusione rischia di essere violata.

- 6.19 Mostrate come implementare le operazioni `wait()` e `signal()` dei semafori negli ambienti multiprocessore tramite l'istruzione `TestAndSet()`. La soluzione dovrebbe comportare un'attesa attiva minima.
- 6.20 L'Esercizio 4.17 richiede che il thread padre attenda che il thread figlio termini la sua esecuzione prima di visualizzare i valori calcolati. Ipotizziamo di voler permettere che il thread padre acceda ai numeri di Fibonacci non appena questi vengono calcolati dal figlio, piuttosto che attendere che il figlio termini. Spiegate quali modifiche sono necessarie alla soluzione dell'Esercizio 4.17. Implementate la soluzione modificata.
- 6.21 Dimostrate che monitor, sezioni critiche e semafori sono equivalenti, e che, pertanto, consentono di risolvere gli stessi problemi di sincronizzazione.
- 6.22 Progettate un monitor con memoria limitata, in cui i buffer siano parte del monitor stesso.
- 6.23 All'interno di un monitor, la mutua esclusione fa sì che il monitor con memoria limitata dell'Esercizio 6.22 sia adatto soprattutto buffer piccoli.
- Spiegate perché questa affermazione è vera.
  - Progettate un nuovo schema idoneo a buffer grandi.
- 6.24 Argomentate il compromesso tra equità e produttività delle operazioni nel problema dei lettori-scrittori. Proponete un metodo per risolvere il problema dei lettori-scrittori senza che si determini attesa indefinita.
- 6.25 Come si differenzia l'operazione `signal()` dei monitor dalla corrispondente operazione dei semafori?
- 6.26 Ipotizziamo che l'istruzione `signal()` possa apparire solo per ultima in una procedura monitor. Proponete un modo per semplificare l'implementazione descritta nell'Paragrafo 6.7.
- 6.27 Considerate un sistema formato dai processi  $P_1, P_2, \dots, P_n$ , aventi priorità distinte, rappresentate da numeri interi. Scrivete un monitor grazie al quale tre identiche stampanti in linea siano assegnate a tali processi, stabilendo l'ordine di allocazione in base alle priorità.
- 6.28 Un file deve essere condiviso fra processi diversi, ognuno dei quali è identificato da un numero univoco. Al file possono accedere simultaneamente vari processi, a patto che osservino la seguente prescrizione: la somma degli identificatori di tutti i processi che accedono al file deve essere minore di  $n$ . Scrivete un monitor per coordinare l'accesso al file.
- 6.29 Se un processo invoca `signal()` al verificarsi di una condizione all'interno di un monitor, esso può continuare la propria esecuzione, oppure cedere il controllo al processo che ha ricevuto il segnale. Come cambia la soluzione al precedente esercizio in queste due circostanze?
- 6.30 Supponete di sostituire le operazioni `wait()` e `signal()` dei monitor con un solo costrutto, `await(B)`, dove  $B$  è un'espressione booleana generale, che obbliga il processo che lo esegue ad attendere finché  $B$  diventi vera.
- Si metta a punto, in base a questo modello, un monitor che affronti il problema dei lettori-scrittori.
  - Chiarite che cosa impedisce di implementare efficientemente questo costrutto.

- c. Quali restrizioni è necessario imporre all'istruzione `await(B)` perché possa essere implementata efficientemente? (Suggerimento: limitate la forma di `B`; si veda [Kessels 1977].)
- 6.31 Scrivete un monitor per realizzare una “sveglia” con cui un programma chiamante possa differire la propria esecuzione per il numero prescelto di unità di tempo (“batti o tic”). Si può ipotizzare l'esistenza di un orologio hardware che invochi una procedura `battito` sul vostro monitor a intervalli regolari.
- 6.32 Perché Solaris, Linux e Windows XP usano gli spinlock come meccanismo di sincronizzazione esclusivamente sui sistemi multiprocessore, e non su quelli a processore unico?
- 6.33 Nei sistemi dotati di un log che mettono a disposizione le transazioni è impossibile aggiornare i dati prima che siano registrati i relativi elementi. Perché è necessaria una tale restrizione?
- 6.34 Dimostrate che il protocollo per la gestione dei lock a due fasi assicuri la serializzabilità dei conflitti.
- 6.35 Quali conseguenze derivano dall'assegnazione di una nuova marca temporale a una transazione annullata per *rollback*? In che modo il sistema gestisce le transazioni richieste successivamente al *rollback* della transazione, e caratterizzate da marche temporali precedenti rispetto a essa?
- 6.36 Assumete di dover gestire un numero finito di risorse, appartenenti tutte allo stesso tipo. I processi possono richiedere una parte di queste risorse, per poi restituirlle quando hanno terminato. Un esempio proviene dai programmi commerciali, molti dei quali includono in un singolo pacchetto alcune licenze, che indicano il numero di applicazioni concorrentemente eseguibili. All'avvio di un'applicazione, il totale delle licenze a disposizione diminuisce. Se un'applicazione termina, il totale delle licenze aumenta. Quando tutte le licenze sono in uso, le nuove richieste per l'avvio di un'applicazione non avranno esito. Si ottempera a tali richieste solo nel momento in cui il proprietario di una licenza lascia libera un'applicazione e restituisce la relativa licenza.

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

Quando un processo mira a ottenere alcune risorse, invoca la funzione `decrease_count()`:

```
/* decrementa il numero di risorse disponibili */
/* di una quantità pari a count; restituisce 0 */
/* se vi sono risorse sufficienti, -1 altrimenti */
int decrease_count(int_count) {
    if (available_resources < count)
        return -1;
    else {
        available_resources -= count;
        return 0;
    }
}
```

Quando un processo intende restituire le risorse che possiede, chiama la funzione `increase_count()`:

```
/* incrementa il numero di risorse disponibili */
/* di una quantità pari a count */
int increase_count(int count) {
    available_resources += count;
    return 0;
}
```

Il programma precedente provoca un conflitto per l'accesso (ossia una *race condition*), per ovviare al quale dovete:

- a) individuare i dati interessati dal conflitto;
  - b) identificare, nel codice, il brano (o i brani) che danno luogo al conflitto;
  - c) adoperare un semaforo che regoli gli accessi, eliminando il conflitto.
- 6.37 La funzione `decrease_count()` dell'esercizio precedente restituisce 0 qualora vi siano sufficienti risorse a disposizione e -1 in caso contrario. Questo complica la programmazione di un processo che intenda sfruttare delle risorse:

```
while (decrease_count(count) == -1)
;
```

Riscrivete il segmento di codice dedicato alla gestione delle risorse, servendovi di un monitor e di variabili condizionali, in modo che la funzione `decrease_count()` sospenda il processo finché non siano disponibili risorse sufficienti. Ciò consentirà a un processo di invocare `decrease_count()` semplicemente così:

```
decrease_count(count);
```

Il processo rientra da questa chiamata solo quando vi siano risorse sufficienti disponibili.

## Problemi di programmazione

- 6.38 Considerate il problema del barbiere che dorme. Un negozio di barbiere ha una sala d'attesa con  $n$  sedie e una stanza del barbiere con la sedia da lavoro. Se non ci sono clienti da servire, il barbiere dorme; se un cliente entra nel negozio e trova tutte le sedie occupate, se ne va; se il barbiere è occupato, ma ci sono sedie disponibili, il cliente si siede; se il barbiere dorme, il cliente lo sveglia. Scrivete un programma che coordini barbiere e clienti.

## Progetti di programmazione

- 6.39 Il problema dei produttori e dei consumatori

Nel Paragrafo 6.6.1 abbiamo proposto una soluzione al problema dei produttori e consumatori che si basa su semafori e si avvale di un buffer limitato. In questo progetto daremo soluzione al problema del buffer limitato mediante i processi produttore e consumatore schematizzati nelle Figure 6.10 e 6.11, rispettivamente. La soluzione presentata nel Paragrafo 6.6.1 utilizza tre semafori: vuote e piene, che enumere-

rano le posizioni vuote e piene del buffer, e `mutex`, un semaforo binario, o a mutua esclusione, che protegge l'inserimento nel (o l'estrazione dal) buffer. In questo progetto i semafori vuote e piene saranno semafori contatori standard, mentre, per rappresentare `mutex`, si userà un lock mutex in luogo di un semaforo binario. Il produttore e il consumatore – eseguiti come thread separati – trasferiscono gli oggetti a un buffer, oppure li prelevano da esso; il buffer è sincronizzato con le strutture vuote, piene e `mutex`. Potete scegliere di risolvere il problema con Pthreads o con la API Win32.

## Buffer

Internamente, il buffer è costituito da un array di dimensione fissa avente tipo `buffer_item`, definito tramite `typedef`. L'array di oggetti `buffer_item` sarà trattato come una coda circolare. Definizione e dimensione relative possono essere poste in un file di intestazione come il seguente:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

Il buffer sarà manipolato da due funzioni, `insert_item()` e `remove_item()`, richiamate, rispettivamente, dal thread del produttore e dal thread del consumatore. Tali funzioni, in un primo abbozzo schematico, appaiono nella Figura 6.27.

Le funzioni `insert_item` e `remove_item` sincronizzeranno il produttore e il consumatore usando gli algoritmi delineati nelle Figure 6.10 e 6.11. Il buffer richiederà, inoltre, una funzione per inizializzare `mutex`, come pure i semafori vuote e piene.

La funzione `main()` inizializzerà il buffer, creando due thread distinti per il produttore e il consumatore. Dopo aver creato tali thread, la funzione `main()` rimarrà inattiva per un certo tempo e, alla ripresa dell'attività, terminerà l'applicazione. La funzione `main()` riceverà tre parametri dalla riga di comando, indicanti:

```
#include "buffer.h"
/* il buffer */
buffer_item buffer[BUFFER_SIZE];
int insert_item(buffer_item item) {
    /* inserisce un elemento nel buffer
       restituisce 0 se ha successo,
       altrimenti -1 per segnalare l'errore */
}
int remove_item(buffer_item *item) {
    /* estrae un elemento dal buffer
       e lo pone in item
       restituisce 0 se ha successo,
       altrimenti -1 per segnalare l'errore */
}
```

**Figura 6.27** Schema delle funzioni.

```
#include "buffer.h"
int main(int argc, char *argv[]) {
    /* 1. Ottiene i parametri argv[0], argv[1], argv[2]
        dalla riga di comando */
    /* 2. Inizializza il buffer */
    /* 3. Crea i thread di tipo produttore */
    /* 4. Crea i thread di tipo consumatore */
    /* 5. Attende */
    /* 6. Termina */
}
```

**Figura 6.28** Scheletro del programma.

- a. il tempo di inattività prima di terminare;
- b. il numero di thread produttori;
- c. il numero di thread consumatori.

La struttura di questa funzione è illustrata nella Figura 6.28.

### Thread produttori e consumatori

Il thread produttore alternerà l'inattività per periodi di tempo non prevedibili all'inserimento di un intero casuale nel buffer. I numeri casuali saranno generati tramite la funzione `rand()`, che dà luogo a valori interi casuali compresi tra 0 e `RAND_MAX`. Anche il consumatore rimarrà in stato di d'inattività per un intervallo di tempo casuale; tornato attivo, tenterà di estrarre un oggetto dal buffer. Lo schema dei thread produttori e consumatori è illustrato nella Figura 6.29.

Nei paragrafi successivi ci occuperemo dapprima dei dettagli relativi a Pthreads per poi passare alla API di Win32.

### Creazione dei thread con Pthreads

Per la creazione di thread con la API di Pthreads rimandiamo il lettore al Capitolo 4, contenente i dettagli necessari per generare il produttore e il consumatore.

### Lock mutex di Pthreads

Il brano di codice riportato nella Figura 6.30 illustra come si possano sfruttare i lock mutex forniti dalla API di Pthreads per proteggere una sezione critica.

Pthreads utilizza il tipo di dati `pthread_mutex_t` per i lock mutex. Un mutex è generato con la funzione `pthread_mutex_init(&mutex, NULL)`, in cui il primo parametro è un puntatore al mutex. Passando `NULL` come secondo parametro, il mutex ha gli attributi di default. Il mutex è attivato e disattivato con le funzioni `pthread_mutex_lock()` e `pthread_mutex_unlock()`. Se il lock mutex non è disponibile quando è invocata `pthread_mutex_lock()`, il thread chiamante rimane bloccato finché il proprietario del lock invoca `pthread_mutex_unlock()`. Tutte le funzioni mutex restituiscono il valore 0 in assenza d'errori e un valore non nullo altrimenti.

```

#include <stdlib.h> /* necessario per rand() */
#include "buffer.h"

void *producer(void *param) {
    buffer_item item;

    while (true) {
        /* resta inattivo per un intervallo di tempo casuale */
        sleep(...);
        /* genera un numero casuale */
        item = rand();
        if (insert_item(item))
            fprintf("report error condition");
        else
            printf("producer produced %d\n", item);
    }
}

void *consumer(void *param) {
    buffer_item item;

    while (true) {
        /* resta inattivo per un intervallo di tempo casuale */
        sleep(...);
        if (remove_item(&item))
            fprintf("report error condition");
        else
            printf("consumer consumed %d\n", item);
    }
}

```

**Figura 6.29** Schema dei thread dei produttori e consumatori.

```

#include <pthread.h>
pthread_mutex_t mutex;

/* crea il lock mutex */
pthread_mutex_init(&mutex,NULL);

/* acquisisce il lock mutex */
pthread_mutex_lock(&mutex);

/*** sezione critica ***/

/* rimuove il lock mutex */
pthread_mutex_unlock(&mutex);

```

**Figura 6.30** Esempio di codice.

```
#include <semaphore.h>
sem_t mutex;

/* crea il semaforo */
sem_init(&mutex, 0, 1);

/* acquisisce il semaforo */
sem_wait(&mutex);

/*** sezione critica ***/

/* restituisce il semaforo */
sem_post(&mutex);
```

**Figura 6.31** Esempio di codice.

## Semafori di Pthreads

Pthreads fornisce due tipi di semafori: con o senza nome. Per questo progetto, impiegheremo semafori senza nome. Il codice seguente mostra come si dà origine a un semaforo:

```
#include <semaphore.h>
sem_t sem;

/* Crea il semaforo e lo inizializza a 5 */
sem_init(&sem, 0, 5);
```

La funzione `sem_init()`, che crea e inizializza il semaforo, accetta tre parametri:

- un puntatore al semaforo;
- un flag indicante il livello di condivisione;
- il valore iniziale del semaforo.

Nell'esempio precedente, passare il flag 0 significa stabilire che questo semaforo è condivisibile unicamente dai thread appartenenti al medesimo processo che ha creato il semaforo. Un valore non nullo consentirebbe l'accesso al semaforo anche da parte di altri processi. In questo esempio, il valore iniziale del semaforo è inizializzato a 5.

Nel Paragrafo 6.5 abbiamo descritto le classiche operazioni sui semafori `wait()` e `signal()`. Pthreads richiama le operazioni `wait()` e `signal()` rispettivamente nell'ordine `sem_wait()` e `sem_post()`. L'esempio di codice illustrato nella Figura 6.31 genera un semaforo binario mutex con valore iniziale 1, e ne illustra l'uso nel proteggere una sezione critica.

## Lock mutex di Win32

I lock mutex sono oggetti dispatcher, come si è visto nel Paragrafo 6.8.2. Di seguito si può osservare come creare un lock mutex utilizzando la funzione `CreateMutex()`:

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

Il primo parametro si riferisce a un attributo di sicurezza per il lock mutex. Impostando a `NULL` questo attributo, i processi figli del creatore del lock mutex non ereditano il riferimento al mutex stesso. Il secondo parametro indica se il processo che ha creato il mutex sia il possessore iniziale del lock mutex: passando il valore `false`, si asserisce che il thread non è il possessore iniziale; vedremo fra breve come acquisire i lock mutex. Il terzo parametro consente di attribuire un nome al mutex. Tuttavia, poiché si è scelto un valore `NULL`, non gli attribuiremo un nome. Se `CreateMutex()` è stata eseguita senza errori, restituirà il riferimento `HANDLE` al lock; altrimenti, `NULL`.

Nel Paragrafo 6.8.2 abbiamo descritto i due possibili stati, detti *signaled* e *nonsignaled*, degli oggetti dispatcher. Quando un oggetto è nello stato *signaled* ne può entrare in possesso; una volta acquisito, l'oggetto dispatcher (per esempio, un lock mutex) passa allo stato *nonsignaled*. L'oggetto diviene *signaled* non appena risulta nuovamente disponibile.

I lock mutex si acquisiscono richiamando la funzione `WaitForSingleObject()` con due parametri: il riferimento `HANDLE` al lock, e un flag che indica la durata dell'attesa. Il codice mostra come si può acquisire il lock mutex creato in precedenza.

```
WaitForSingleObject(Mutex, INFINITE);
```

Il valore `INFINITE` significa che non vi è limite a quanto si è disposti ad attendere che il lock diventi disponibile. Con altri valori il thread chiamante potrebbe troncare l'operazione, qualora il lock non tornasse disponibile entro un tempo definito. Se il lock è in stato *signaled*, `WaitForSingleObject()` restituisce immediatamente il controllo, e il lock diviene *nonsignaled*. Per rendere disponibile un lock (ossia, per far sì che entri nello stato *signaled*) si invoca `ReleaseMutex()`:

```
ReleaseMutex(Mutex);
```

## Semafori di Win32

Nella API di Win32 anche i semafori sono oggetti dispatcher e pertanto adoperano lo stesso meccanismo di segnalazione dei lock mutex. I semafori si creano tramite il codice:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

In analogia a quanto descritto per i lock mutex, il primo e l'ultimo parametro designano un attributo di sicurezza e un nome per il semaforo. Il secondo e il terzo parametro indicano il valore iniziale e il valore massimo del semaforo. In questo caso, il valore iniziale del semafo-

ro è 1, mentre il suo valore massimo è 5. Se termina senza errori, `CreateSemaphore()` restituisce un riferimento HANDLE al lock mutex; in caso contrario, restituisce NULL.

I semafori si acquisiscono con la stessa funzione trattata per i lock mutex, cioè `WaitForSingleObject()`. Per acquisire il semaforo Sem di questo esempio, si invoca:

```
WaitForSingleObject(Semaphore, INFINITE);
```

Se il valore del semaforo è  $> 0$ , esso si trova nello stato *signaled*, e viene quindi acquisito dal thread chiamante; altrimenti, il thread chiamante è sospeso (per un tempo imprecisato, a causa del parametro `INFINITE`) in attesa che il semaforo diventi *signaled*.

Per i semafori di Win32, l'equivalente dell'operazione `signal()` è la funzione `ReleaseSemaphore()`. Essa accetta tre parametri: (1) il riferimento HANDLE al semaforo, (2) il valore di cui incrementare il semaforo e (3) un puntatore al valore precedente del semaforo. Si può aumentare `Sem` di 1 nel modo seguente:

```
ReleaseSemaphore(Sem, 1, NULL);
```

`ReleaseSemaphore()` e `ReleaseMutex()` restituiscono 0 se eseguite senza errori e un valore non nullo in caso contrario.

## 6.11 Note bibliografiche

Il problema della mutua esclusione fu discusso per la prima volta in un classico lavoro di [Dijkstra 1965a]. L'algoritmo di Dekker (Esercizio 6.9), la prima soluzione software corretta al problema della mutua esclusione per due processi, fu sviluppato dal matematico olandese T. Dekker. L'algoritmo è anche esaminato in [Dijkstra 1965a]; una soluzione più semplice si trova in [Peterson 1981] (Figura 6.2).

[Dijkstra 1965b] contiene la prima soluzione al problema della mutua esclusione per  $n$  processi. Si tratta però di una soluzione che non garantisce un limite superiore al tempo d'attesa dei processi che richiedano l'ingresso nelle proprie sezioni critiche. [Knuth 1966] presenta il primo algoritmo con un limite pari a  $2^n$  turni, e [deBruijn 1967] ne propone un raffinamento che riduce l'attesa a  $n^2$  turni. [Eisenberg e McGuire 1972] riescono a portare l'attesa al limite inferiore di  $n-1$  turni. Un altro algoritmo con questa proprietà, ma più facile da comprendere e programmare, è l'algoritmo del fornaio, proposto in [Lamport 1974]. [Burns 1978] sviluppa l'algoritmo basato sull'hardware che soddisfa il requisito dell'attesa limitata.

Trattazioni generali del problema della mutua esclusione si trovano in [Lamport 1986] e [Lamport 1991]. Una raccolta di algoritmi per la mutua esclusione è in [Raynal 1986].

Il concetto di semaforo è suggerito da [Dijkstra 1965a]. [Patil 1971] studia la misura in cui i semafori sono in grado di risolvere i problemi di sincronizzazione. [Parnas 1975] illustra alcuni errori nelle argomentazioni di Patil. [Kosoraju 1973] prosegue il lavoro di Patil esibendo un problema irrisolvibile dalle sole operazioni `wait()` e `signal()`. [Lipton 1974] analizza le varie limitazioni delle diverse primitive per la sincronizzazione.

I classici problemi del coordinamento dei processi presentati in questo capitolo fungono da paradigma per un'ampia classe di problemi di controllo della concorrenza. Il problema del buffer limitato, dei cinque filosofi e del barbiere che dorme (Esercizio 6.38) sono proposti in [Dijkstra 1965a] e [Dijkstra 1971]. Il problema dei fumatori di sigarette (Esercizio 6.2) è stato sviluppato da Patil [1971]; quello dei lettori-scrittori è dovuto a [Courtois et al. 1971]. Le questioni legate alla lettura e alla scrittura concorrenti sono discusse in [Lamport 1977]. Il problema della sincronizzazione di processi indipendenti è trattato da [Lamport 1976].

Il concetto di sezione critica è dovuto a [Hoare 1972] e [Brinch-Hansen 1972]. Il concetto di monitor è sviluppato da [Brinch-Hansen 1973], e una sua completa descrizione si trova in [Hoare 1974]. [Kessels 1977] propone un'estensione del concetto di monitor che incorpora un meccanismo automatico per le segnalazioni. I frutti dell'esperienza fatta nell'uso dei monitor in programmi concorrenti sono presentati in [Lampson e Redell 1979]. Discussioni generali sulla programmazione concorrente si trovano in [Ben-Ari 1990] e [Birrel 1989].

L'ottimizzazione delle prestazioni delle primitive per la gestione dei lock è discussa in molti lavori, fra cui [Lamport 1987], [Mellor-Crummey e Scott 1991] e [Anderson 1990]. L'impiego di oggetti condivisi che non richiede l'uso di sezioni critiche è illustrato da [Herlihy 1993], [Bershad 1993] e [Kopetz e Reisinger 1993]. Nuove istruzioni hardware utili per l'implementazione di primitive per la sincronizzazione sono discusse in lavori come [Culler et al. 1998], [Goodman et al. 1989], [Barnes 1993] e [Herlihy e Moss 1993].

Alcuni dettagli sui lock di Solaris sono presentati da [Mauro e McDougall 2007]. Si noti che i lock usati dal kernel sono implementati anche a beneficio dei thread utenti: lo stesso tipo di lock è disponibile all'interno e all'esterno del kernel. I dettagli sulla sincronizzazione in Windows 2000 si trovano in [Solomon e Russinovich 2000]. Goetz et al. [2006] presenta una trattazione dettagliata della programmazione concorrente in Java e del pacchetto `java.util.concurrent`.

Lo schema di registrazione con scrittura anticipata è introdotto per la prima volta nel [System R da Gray et al. 1981]. Il concetto di serializzabilità è formulato da [Eswaran et al. 1976], in relazione al loro lavoro sul controllo della concorrenza nel System R. Il protocollo per la gestione dei lock a due fasi è dovuto a [Eswaran et al. 1976]. Lo schema di controllo della concorrenza basato sulla marcatura temporale è in [Reed 1983]. Diversi algoritmi per il controllo della concorrenza basati sulla marcatura temporale sono spiegati in Bernstein e Goodman [1980]. Adl-Tabatabai et al. [2007] tratta il tema della memoria transazionale.

## Capitolo 7

# Stallo dei processi



### OBIETTIVI

- Descrizione approfondita delle situazioni di stallo (*deadlock*) che impediscono il completamento del lavoro a gruppi di processi concorrenti.
- Presentazione dei metodi differenti per prevenire o impedire le situazioni di stallo.

In un ambiente con multiprogrammazione più processi possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il processo richiedente passa allo stato d'attesa. Se le risorse richieste sono trattenute da altri processi, a loro volta nello stato d'attesa, il processo potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di stallo (*deadlock*). Questo argomento è trattato brevemente anche nel Capitolo 6, nello studio dei semafori.

Un efficace esempio di situazione di stallo si può ricavare da una legge dello stato del Kansas approvata all'inizio del XX secolo, che in una sua parte recita: "Quando due treni convergono a un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato".

In questo capitolo si descrivono i metodi che un sistema operativo può usare per prevenire o affrontare le situazioni di stallo. Occorre in ogni caso notare che la maggior parte dei sistemi operativi attuali non offre strumenti di prevenzione di queste situazioni. Funzioni adatte a questo scopo saranno probabilmente aggiunte presto. Date le tendenze correnti, il problema dello stallo può diventare soltanto più importante: aumento del numero dei processi e delle risorse all'interno di un sistema, programmi multithread e preferenza attribuita ai sistemi con archivi di file e basi di dati anziché ai sistemi a lotti.

## 7.1 Modello del sistema

Un sistema è composto da un numero finito di risorse da distribuire tra più processi in competizione. Le risorse sono suddivise in tipi differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di CPU, spazio di memoria, file e dispositivi di I/O (come stampanti e lettori DVD), sono tutti esempi di tipi di risorsa. Se un sistema ha due unità d'elaborazione, tale tipo di risorsa ha due istanze. Analogamente, il tipo di risorsa *stampante* può avere cinque istanze.

Se un processo richiede un'istanza relativa a un tipo di risorsa, l'assegnazione di *qualsiasi* istanza di quel tipo può soddisfare la richiesta. Se ciò non si verifica significa che le istanze non sono identiche e le classi di risorse non sono state definite correttamente. Un sistema può, per esempio, avere due stampanti; se a nessuno interessa sapere quale sia la stampante in funzione, le due stampanti si possono definire come appartenenti alla stessa classe di risorse; se, però, una stampante si trova al nono piano e l'altra al piano terra, allora le due stampanti si possono considerare non equivalenti, e per definire ciascuna delle due può essere necessario ricorrere a classi di risorse distinte.

Prima di adoperare una risorsa, un sistema deve richiederla e, dopo averla usata, deve rilasciarla. Un processo può richiedere tutte le risorse necessarie per eseguire il compito assegnatogli, anche se ovviamente il numero delle risorse richieste non può superare quello totale delle risorse disponibili nel sistema: un processo non può richiedere tre stampanti se il sistema ne ha solo due.

Nelle ordinarie condizioni di funzionamento un processo può servirsi di una risorsa soltanto se rispetta la seguente sequenza.

1. **Richiesta.** Se la richiesta non si può soddisfare immediatamente – per esempio, perché la risorsa è attualmente in possesso di un altro processo – il processo richiedente deve attendere finché non possa acquisire tale risorsa.
2. **Uso.** Il processo può operare sulla risorsa (se, per esempio, la risorsa è una stampante, il processo può effettuare una stampa).
3. **Rilascio.** Il processo rilascia la risorsa.

La richiesta e il rilascio di risorse avvengono tramite chiamate di sistema, come illustrato nel Capitolo 2. Ne sono esempi le chiamate di sistema `request()` e `release()`, `open()` e `close()`, `allocate()` e `free()`. La richiesta e il rilascio di altre risorse si possono eseguire per mezzo delle operazioni `wait()` e `signal()` su semafori. Quindi, ogni volta che si usa una risorsa, il sistema operativo controlla che il processo utente ne abbia fatto richiesta e che questa gli sia stata assegnata. Una tabella di sistema registra lo stato di ogni risorsa e, se questa è assegnata, indica il processo relativo. Se un processo richiede una risorsa già assegnata a un altro processo, il processo richiedente può essere accodato agli altri processi che attendono tale risorsa.

Un gruppo di processi entra in stallo quando tutti i processi del gruppo attendono un evento che può essere causato solo da un altro processo che si trova nello stato di attesa. Gli eventi maggiormente discussi in questo capitolo sono l'acquisizione e il rilascio di risorse. Le risorse possono essere sia fisiche, per esempio, stampanti, unità a nastri, spazio di memoria e cicli di CPU, sia logiche, per esempio, file, semafori e monitor. Tuttavia anche altri eventi possono condurre a situazioni di stallo, come le funzioni di IPC trattate nel Capitolo 3.

Per esaminare una situazione di stallo si consideri un sistema con tre lettori CD. Si supponga che tre processi ne possiedano uno ciascuno. Se ogni processo richiedesse un altro lettore CD, i tre processi entrerebbero in stallo: ciascuno attenderebbe il rilascio del lettore, ma quest'evento potrebbe essere causato solo da uno degli altri processi che a loro volta attendono l'ulteriore lettore CD. Quest'esempio descrive uno stallo causato da processi che competono per acquisire lo stesso tipo di risorsa.

Le situazioni di stallo possono implicare anche diversi tipi di risorsa. Si consideri, per esempio, un sistema con una stampante e un lettore DVD, e si supponga che il processo  $P_i$  sia in possesso del lettore e il processo  $P_j$  della stampante. Se  $P_i$  richiedesse la stampante e  $P_j$  il lettore DVD, si avrebbe uno stallo.

Un programmatore che sviluppa applicazioni multithread deve prestare una particolare attenzione a questo problema: i programmi multithread sono ottimi candidati alle situazioni di stallo, poiché più thread possono competere per le risorse condivise.

## 7.2 Caratterizzazione delle situazioni di stallo

In una situazione di stallo, i processi non terminano mai l'esecuzione, e le risorse del sistema vengono bloccate impedendo l'esecuzione di altri processi. Prima di trattarne le soluzioni, descriviamo le caratteristiche del problema dello stallo.

### 7.2.1 Condizioni necessarie

Si può avere una situazione di stallo *solo se* si verificano contemporaneamente le seguenti quattro condizioni.

1. **Mutua esclusione.** Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare il processo richiedente fino al rilascio della risorsa.
2. **Possesso e attesa.** Un processo in possesso di almeno una risorsa attende di acquisire risorse già in possesso di altri processi.
3. **Impossibilità di prelazione.** Non esiste un diritto di prelazione sulle risorse, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Attesa circolare.** Deve esistere un insieme  $\{P_0, P_1, \dots, P_n\}$  di processi, tale che  $P_0$  attende una risorsa posseduta da  $P_1$ ,  $P_1$  attende una risorsa posseduta da  $P_2, \dots, P_{n-1}$  attende una risorsa posseduta da  $P_n$  e  $P_n$  attende una risorsa posseduta da  $P_0$ .

### STALLO CON LOCK MUTEX

Vediamo come si possa incorrere in situazioni di stallo in un programma multithread di Pthread che usa i lock mutex. La funzione `pthread_mutex_init()` inizializza un semaforo mutex su cui non è attivo un lock. I lock mutex sono acquisiti e restituiti per mezzo di `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente. Se un thread tenta di acquisire un mutex già impegnato, l'invocazione di `pthread_mutex_lock()` blocca il thread finché il possessore del mutex non invochi `pthread_mutex_unlock()`.

Il brano di codice seguente genera due lock mutex:

```
/* Crea e inizializza due lock mutex */
pthread_mutex_t primo_mutex;
pthread_mutex_t secondo_mutex;

pthread_mutex_init(&primo_mutex, NULL);
pthread_mutex_init(&secondo_mutex, NULL);
```

Si creano poi due thread, di nome `thread_uno` e `thread_due`, che possono accedere a entrambi i lock mutex. Sono eseguiti dalle funzioni `esegui_uno()` e `esegui_due()`, rispettivamente, come mostrato nella Figura 7.1.

In questo esempio, `thread_uno` tenta di acquisire i lock `mutex` nell'ordine (1) `primo_mutex`, (2) `secondo_mutex`, mentre `thread_due` tenta di acquisire i lock nell'ordine (1) `secondo_mutex`, (2) `primo_mutex`. Lo stallo è possibile nell'ipotesi che `thread_uno` acquisca `primo_mutex`, mentre `thread_due` acquisisca `secondo_mutex`.

Si noti che lo stallo, pur essendo possibile, non si verifica se `thread_uno` è in grado di acquisire e rilasciare entrambi i lock prima che `thread_due` tenti a sua volta di impossessarsene. L'esempio evidenzia un problema importante per la gestione dello stallo: è difficile identificare e sottoporre a test gli stalli che si verificano solo in determinate condizioni.

```
/* thread_uno esegue in questa funzione */
void *esegui_uno(void *param)
{
    pthread_mutex_lock(&primo_mutex);
    pthread_mutex_lock(&secondo_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&secondo_mutex);
    pthread_mutex_unlock(&primo_mutex);

    pthread_exit(0);
}

/* thread_due esegue in questa funzione */
void *esegui_due(void *param)
{
    pthread_mutex_lock(&secondo_mutex);
    pthread_mutex_lock(&primo_mutex);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&primo_mutex);
    pthread_mutex_unlock(&secondo_mutex);

    pthread_exit(0);
}
```

**Figura 7.1** Esempio di stallo.

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti; tuttavia è utile considerare separatamente ciascuna condizione (Paragrafo 7.4).

## 7.2.2 Grafo di assegnazione delle risorse

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta **grafo di assegnazione delle risorse**. Si tratta di un insieme di vertici  $V$  e un insieme di archi  $E$ , con l'insieme di vertici  $V$  composto da due sottoinsiemi:  $P = \{P_1, P_2, \dots, P_n\}$ , che rappresenta tutti i processi del sistema, e  $R = \{R_1, R_2, \dots, R_m\}$ , che rappresenta tutti i tipi di risorsa del sistema.

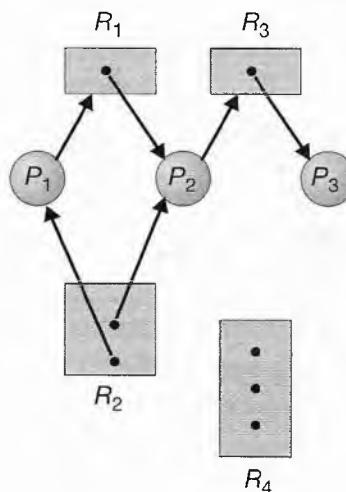
Un arco diretto dal processo  $P_i$  al tipo di risorsa  $R_j$  si indica  $P_i \rightarrow R_j$ , e significa che il processo  $P_i$  ha richiesto un'istanza del tipo di risorsa  $R_j$ , e attualmente attende tale risorsa. Un arco diretto dal tipo di risorsa  $R_j$  al processo  $P_i$  si indica  $R_j \rightarrow P_i$ , e significa che un'istanza del tipo di risorsa  $R_j$  è assegnata al processo  $P_i$ . Un arco orientato  $P_i \rightarrow R_j$  si chiama **arco di richiesta**, un arco orientato  $R_j \rightarrow P_i$  si chiama **arco di assegnazione**.

Graficamente ogni processo  $P_i$  si rappresenta con un cerchio e ogni tipo di risorsa  $R_j$  si rappresenta con un rettangolo. Giacché il tipo di risorsa  $R_j$  può avere più di un'istanza, ciascuna di loro si rappresenta con un puntino all'interno del rettangolo. Occorre notare che un arco di richiesta è diretto soltanto verso il rettangolo  $R_j$ , mentre un arco di assegnazione deve designare anche uno dei puntini del rettangolo.

Quando il processo  $P_i$  richiede un'istanza del tipo di risorsa  $R_j$ , si inserisce un arco di richiesta nel grafo di assegnazione delle risorse. Se questa richiesta può essere esaudita, si trasforma *immediatamente* l'arco di richiesta in un arco di assegnazione, che al rilascio della risorsa viene cancellato.

Nel grafo di assegnazione delle risorse della Figura 7.2 è illustrata la seguente situazione.

- ◆ Insiemi  $P$ ,  $R$  ed  $E$ :
  - ◊  $P = \{P_1, P_2, P_3\}$
  - ◊  $R = \{R_1, R_2, R_3, R_4\}$
  - ◊  $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- ◆ Istanze delle risorse:
  - ◊ un'istanza del tipo di risorsa  $R_1$
  - ◊ due istanze del tipo di risorsa  $R_2$



**Figura 7.2** Grafo di assegnazione delle risorse.

- ◊ un'istanza del tipo di risorsa  $R_3$
- ◊ tre istanze del tipo di risorsa  $R_4$
- ♦ Stati dei processi:
  - ◊ il processo  $P_1$  possiede un'istanza del tipo di risorsa  $R_2$  e attende un'istanza del tipo di risorsa  $R_1$
  - ◊ il processo  $P_2$  possiede un'istanza dei tipi di risorsa  $R_1$  ed  $R_2$  e attende un'istanza del tipo di risorsa  $R_3$
  - ◊ il processo  $P_3$  possiede un'istanza del tipo di risorsa  $R_3$

Data la definizione di grafo di assegnazione delle risorse, è facile mostrare che, se il grafo non contiene cicli, nessun processo del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo.

Se ciascun tipo di risorsa ha esattamente un'istanza, allora l'esistenza di un ciclo implica la presenza di uno stallo; se il ciclo riguarda solo un insieme di tipi di risorsa, ciascuno dei quali ha solo un'istanza, si è verificato uno stallo. Ogni processo che si trovi nel ciclo è in stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria e sufficiente per l'esistenza di uno stallo.

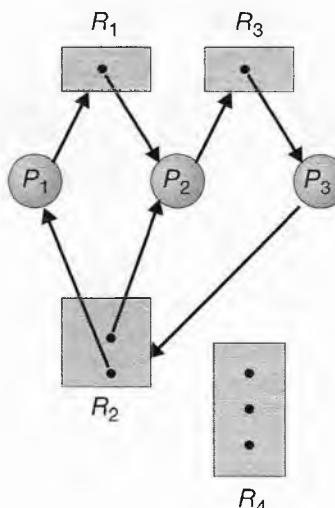
Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente uno stallo. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di uno stallo.

Per spiegare questo concetto conviene ritornare al grafo di assegnazione delle risorse della Figura 7.2. Si supponga che il processo  $P_3$  richieda un'istanza del tipo di risorsa  $R_2$ . Poiché attualmente non è disponibile alcuna istanza di risorsa, si aggiunge un arco di richiesta  $P_3 \rightarrow R_2$  al grafo, com'è illustrato nella Figura 7.3. A questo punto nel sistema ci sono due cicli minimi:

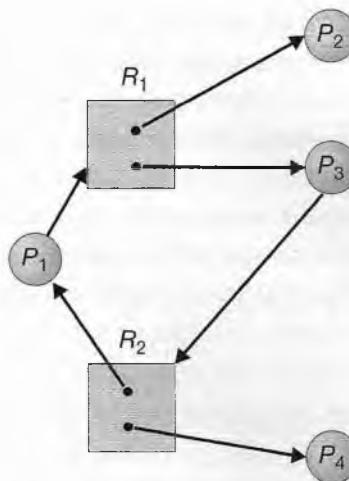
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

I processi  $P_1$ ,  $P_2$  e  $P_3$  sono in stallo: il processo  $P_2$  attende la risorsa  $R_3$ , posseduta dal processo  $P_3$ ; il processo  $P_3$ , invece, attende che il processo  $P_1$  o  $P_2$  rilasci la risorsa  $R_2$ ; inoltre il processo  $P_1$  attende che il processo  $P_2$  rilasci la risorsa  $R_1$ .



**Figura 7.3** Grafo di assegnazione delle risorse con uno stallo.



**Figura 7.4** Grafo di assegnazione delle risorse con un ciclo, ma senza stallo.

Si consideri ora il grafo di assegnazione delle risorse della Figura 7.4. Anche in questo esempio c'è un ciclo:

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

In questo caso, però, non si ha alcuno stallo: il processo  $P_4$  può rilasciare la propria istanza del tipo di risorsa  $R_2$ , che si può assegnare al processo  $P_3$ , rompendo il ciclo.

Per concludere, l'assenza di cicli nel grafo di assegnazione delle risorse implica l'*assenza* di situazioni di stallo nel sistema. Viceversa, la presenza di un ciclo non è sufficiente a implicare la presenza di uno stallo nel sistema. Questa osservazione è importante ai fini della gestione del problema delle situazioni di stallo.

## 7.3 Metodi per la gestione delle situazioni di stallo

Essenzialmente, il problema delle situazioni di stallo si può affrontare impiegando tre metodi:

- ◆ si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri *mai* in stallo;
- ◆ si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino;
- ◆ si può ignorare del tutto il problema, *fingendo* che le situazioni di stallo non possano mai verificarsi nel sistema.

Quest'ultima è la soluzione usata dalla maggior parte dei sistemi operativi, compresi UNIX e Windows. Nel seguito sono spiegati brevemente tutti questi metodi. Gli algoritmi relativi sono presentati in modo dettagliato nei Paragrafi dal 7.4 al 7.7.

Tuttavia, prima di procedere, vogliamo considerare anche l'opinione di quegli sviluppatori che hanno contestato il fatto che nessuno degli approcci di base si adatti da solo all'intero spettro di problemi di allocazione delle risorse nei sistemi operativi. Tali approcci possono essere combinati, in modo da permettere la selezione del migliore per ciascuna classe di risorse del sistema.

Per assicurare che non si verifichi mai uno stallo, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. Prevenire le situazioni di stallo significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie (Paragrafo 7.2.1). Questi metodi, discussi nel Paragrafo 7.4, prevengono le situazioni di stallo prescrivendo il modo in cui si devono fare le richieste delle risorse.

Per evitare le situazioni di stallo occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un processo richiederà e userà durante le sue attività. Con queste informazioni aggiuntive si può decidere se una richiesta di risorse da parte di un processo si può soddisfare o si debba invece sospendere. In tale processo di decisione il sistema tiene conto delle risorse correntemente disponibili, di quelle correntemente assegnate a ciascun processo, e delle future richieste e futuri rilasci di ciascun processo. Questi metodi sono discussi nel Paragrafo 7.5.

Se un sistema non impiega né un algoritmo per prevenire né un algoritmo per evitare gli stalli, tali situazioni possono verificarsi. In un ambiente di questo tipo il sistema può servirsi di un algoritmo che ne esamina lo stato, al fine di stabilire se si è verificato uno stallo e in tal caso ricorrere a un secondo algoritmo per il ripristino del sistema. Tali argomenti sono discussi nei Paragrafi 7.6 e 7.7.

Se un sistema non garantisce che le situazioni di stallo non possano mai verificarsi e non fornisce alcun meccanismo per la loro individuazione e per il ripristino del sistema, situazioni di stallo possono avvenire senza che ci sia la possibilità di capire cos'è successo. In questo caso la presenza di situazioni di stallo non rilevate può causare un degrado delle prestazioni del sistema; infatti la presenza di risorse assegnate a processi che non si possono eseguire determina lo stallo di un numero crescente di processi che richiedono tali risorse, fino al blocco totale del sistema che dovrà essere riavviato manualmente.

Sebbene questo modo di affrontare il problema delle situazioni di stallo non sembri fattibile, è in ogni caso usato in alcuni sistemi operativi perché è più economico dei metodi che altrimenti si dovrebbero adoperare per prevenire le situazioni di stallo, evitarle, o anche individuarle per il successivo ripristino. In molti sistemi, infatti, le situazioni di stallo accadono abbastanza raramente (magari una volta l'anno). In alcune circostanze il sistema è *congelato*, ma non è in stallo. Si consideri, per esempio, la situazione determinata da un processo d'elaborazione in tempo reale che si esegue con la priorità più elevata (o qualsiasi processo in esecuzione in un sistema con scheduler senza diritto di prelazione) e che non restituisce il controllo al sistema operativo. Il sistema deve così disporre di meccanismi non automatici di ripristino per le situazioni che non sono in modo specifico di stallo, che può impiegare anche per le situazioni di stallo.

## 7.4 Prevenire le situazioni di stallo

---

Com'è evidenziato nel Paragrafo 7.2.1, affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può *prevenire* il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare. Questo metodo è descritto nei particolari, con una trattazione di ciascuna delle quattro condizioni necessarie.

### 7.4.1 Mutua esclusione

Deve valere la condizione di mutua esclusione per le risorse non condivisibili: una stampante non può essere condivisa da più processi. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file

aperti per la sola lettura sono un buon esempio di risorsa condivisibile; è ovviamente consentito l'accesso contemporaneo da parte di più processi. Un processo non deve mai attendere una risorsa condivisibile. Ma poiché alcune risorse sono intrinsecamente non condivisibili, non si possono prevenire in generale le situazioni di stallo negando la condizione di mutua esclusione.

### 7.4.2 Possesso e attesa

Per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un processo che richiede una risorsa non ne possegga altre. Si può usare un protocollo che ponga la condizione che ogni processo, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano assegnate. Questa condizione si può realizzare imponendo che le chiamate di sistema che richiedono risorse per un processo precedano tutte le altre.

Un protocollo alternativo è quello che permette a un processo di richiedere risorse solo se non ne possiede: un processo può richiedere risorse e adoperarle, ma prima di richiederne altre deve rilasciare tutte quelle che possiede.

Per spiegare la differenza tra questi due protocolli si può considerare un processo che copi i dati da un DVD a un file in un disco, ordina il contenuto del file e quindi stampa i risultati. Nel caso del primo protocollo, il processo deve richiedere fin dall'inizio il lettore DVD, il file nel disco e una stampante. La stampante rimane in suo possesso per tutta la durata dell'esecuzione, anche se si usa solo alla fine.

Il secondo protocollo prevede che il processo richieda inizialmente solo il lettore DVD e il file nel disco. Il processo copia i dati dal DVD al disco, quindi rilascia le due risorse. A questo punto richiede il file nel disco e la stampante, e dopo aver copiato il file nella stampante rilascia le due risorse e termina.

Entrambi i protocolli presentano due svantaggi principali. Innanzitutto, l'utilizzo delle risorse può risultare poco efficiente, poiché molte risorse possono essere assegnate, ma non utilizzate, per un lungo periodo di tempo. Nel caso del secondo metodo dell'esempio precedente si possono rilasciare il lettore DVD e il file nel disco, per poi richiedere il file e la stampante, solo se si ha la certezza che i dati nel file restino immutati tra il rilascio e la seconda richiesta. Se non si può garantire quest'ultima condizione, è necessario richiedere tutte le risorse all'inizio per entrambi i protocolli.

Il secondo svantaggio è dovuto al fatto che si possono verificare situazioni di attesa indefinita. Un processo che richieda più risorse molto usate può trovarsi nella condizione di attenderne indefiniteamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro processo.

### 7.4.3 Impossibilità di prelazione

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non persista, si può impiegare il seguente protocollo. Se un processo che possiede una o più risorse ne richiede un'altra che non gli si può assegnare immediatamente (cioè il processo deve attendere), allora si esercita la prelazione su tutte le risorse attualmente in suo possesso. Si ha cioè il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il processo sta attendendo; il processo viene nuovamente avviato solo quando può ottenere sia le vecchie risorse sia quella che sta richiedendo.

In alternativa, quando un processo richiede alcune risorse, si verifica la disponibilità di queste ultime: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate a un processo che attende altre risorse. In tal caso si sottraggono le risorse desidera-

te a quest'ultimo processo e si assegnano al processo richiedente. Se le risorse non sono disponibili né sono possedute da un processo in attesa, il processo richiedente deve attendere. Durante l'attesa si può avere la prelazione su alcune sue risorse; ciò può accadere solo se un altro processo le richiede. Un processo si può avviare nuovamente solo quando riceve le risorse che sta richiedendo e recupera tutte quelle a esso sottratte durante l'attesa.

Questo protocollo è adatto a risorse il cui stato si può salvare e recuperare facilmente in un secondo tempo, come i registri della CPU e lo spazio di memoria, mentre non si può in generale applicare a risorse come le stampanti e le unità a nastri.

#### 7.4.4 Attesa circolare

La quarta e ultima condizione necessaria per una situazione di stallo è l'attesa circolare. Un modo per assicurare che tale condizione d'attesa non si verifichi consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e un ordine crescente di numerazione per le risorse richieste da ciascun processo.

Si supponga che  $R = \{R_1, R_2, \dots, R_m\}$  sia l'insieme dei tipi di risorse. A ogni tipo di risorsa si assegna un numero intero unico che permetta di confrontare due risorse e stabilirne la relazione di precedenza nell'ordinamento. Formalmente, si definisce una funzione iniettiva,  $f: R \rightarrow N$ , dove  $N$  è l'insieme dei numeri naturali. Se, ad esempio, l'insieme dei tipi di risorsa  $R$  contiene unità a nastri, unità a dischi e stampanti, la funzione  $f$  si può definire come segue:

$$f(\text{unità a nastri}) = 1$$

$$f(\text{unità a dischi}) = 5$$

$$f(\text{stampante}) = 12$$

Per prevenire il verificarsi di situazioni di stallo si può considerare il seguente protocollo: ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione. Ciò significa che un processo può richiedere inizialmente qualsiasi numero di istanze di un tipo di risorsa, ad esempio  $R_i$ , dopo di che il processo può richiedere istanze del tipo di risorsa  $R_j$  se e solo se  $f(R_j) > f(R_i)$ . Se sono necessarie più istanze dello stesso tipo di risorsa si deve presentare una singola richiesta per tutte le istanze. Ad esempio, con la funzione definita precedentemente, un processo che deve impiegare contemporaneamente l'unità a nastri e la stampante deve prima richiedere l'unità a nastri e poi la stampante. In alternativa, si può stabilire che un processo, prima di richiedere un'istanza del tipo di risorsa  $R_i$ , rilasci qualsiasi risorsa  $R_j$  tale che  $f(R_j) \geq f(R_i)$ .

Se si usa uno di questi due protocolli, la condizione di attesa circolare non può sussistere. Ciò si può dimostrare supponendo, per assurdo, che esista un'attesa circolare. Si supponga che l'insieme di processi coinvolti nell'attesa circolare sia  $\{P_0, P_1, \dots, P_n\}$ , dove  $P_i$  attende una risorsa  $R_i$  posseduta dal processo  $P_{i+1}$ . (Sugli indici si usa l'aritmetica modulare, quindi  $P_n$  attende una risorsa  $R_n$  posseduta da  $P_0$ .) Poiché il processo  $P_{i+1}$  possiede la risorsa  $R_i$  mentre richiede la risorsa  $R_{i+1}$ , è necessario che sia verificata la condizione  $f(R_i) < f(R_{i+1})$  per tutti gli  $i$ , ma ciò implica che  $f(R_0) < f(R_1) < \dots < f(R_n) < f(R_0)$ . Per la proprietà transitiva, risulta che  $f(R_0) < f(R_0)$ , il che è impossibile; quindi, non può esservi attesa circolare.

Questo schema si può implementare in un programma applicativo imponendo un ordine a tutti gli oggetti del sistema che richiedano sincronizzazione. Tutte le richieste inerenti a tali oggetti dovranno seguire l'ordine crescente. Per esempio, se l'ordinamento dei lock nel programma della Figura 7.1 fosse stato

$$f(\text{primo_mutex}) = 1$$

$$f(\text{secondo_mutex}) = 5$$

allora `thread_due` non avrebbe potuto richiedere i lock nell'ordine inverso. Si tenga presente che la semplice esistenza di un ordinamento delle risorse non protegge dallo stallo: è infatti responsabilità degli sviluppatori di applicazioni stendere programmi che rispettino tale ordinamento. Si noti anche che la funzione  $f$  andrebbe definita secondo l'ordine d'uso normale delle risorse del sistema. Per esempio, poiché generalmente l'unità a nastri si usa prima della stampante, è ragionevole definire  $f(\text{unità a nastri}) < f(\text{stampante})$ .

Come si è detto, è responsabilità dei programmatore rispettare l'ordinamento delle risorse; tuttavia, è possibile sviluppare del software in grado di verificare che l'acquisizione dei lock avvenga nell'ordine corretto, ed emetta avvisi appropriati in caso contrario. Uno di tali verificatori, disponibile per le versioni BSD di UNIX (per esempio, FreeBSD), è noto con il nome di `witness`. Esso usa lock mutex per proteggere le sezioni critiche, come descritto nel Capitolo 6, e tiene traccia dinamicamente delle relazioni d'ordine fra i lock del sistema. Si consideri nuovamente il programma della Figura 7.1. Se `thread_uno` è il primo thread ad acquisire i lock, e se ciò avviene nell'ordine (1) `primo_mutex`, (2) `secondo_mutex`, il programma `witness` registra il fatto che `primo_mutex` deve essere acquisito prima di `secondo_mutex`. Se, in seguito, `thread_due` acquisisce i lock violando quest'ordine, il programma `witness` produce sulla console del sistema un messaggio di notifica.

## 7.5 Evitare le situazioni di stallo

Gli algoritmi di prevenzione delle situazioni di stallo trattati nel Paragrafo 7.4 si basano sul controllo delle modalità di richiesta, così da assicurare che non si possa verificare almeno una delle condizioni necessarie perché si abbia uno stallo. Questo metodo può però causare effetti collaterali negativi, come uno scarso utilizzo dei dispositivi e una ridotta produttività del sistema.

Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. In un sistema con un'unità a nastri e una stampante, per esempio, il processo  $P$  può dichiarare che intende richiedere prima l'unità a nastri, poi la stampante, e che rilascerà entrambe solo in seguito. Il processo  $Q$ , invece, richiederà prima la stampante e poi l'unità a nastri. Una volta acquisita la sequenza completa delle richieste e dei rilasci di ogni processo, si può stabilire per ogni richiesta se il processo debba attendere o meno, per evitare una possibile situazione di stallo futura. In seguito a ogni richiesta, il sistema deve esaminare le risorse attualmente disponibili, le risorse attualmente assegnate a ogni processo e le richieste e i rilasci futuri per ciascun processo.

Gli algoritmi differiscono tra loro per la quantità e il tipo di informazioni richieste. Il modello più semplice e più utile richiede che ciascun processo dichiari il *numero massimo* delle risorse di ciascun tipo di cui necessita. Data un'informazione a priori per ogni processo sul massimo numero di risorse richiedibili per ciascun tipo, si può costruire un algoritmo capace di assicurare che il sistema non entri in stallo. Questo algoritmo definisce un metodo per **evitare lo stallo**, ed esamina dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare. Lo *stato* di assegnazione delle risorse è definito dal numero di risorse disponibili e assegnate e dalle richieste massime dei processi. Nei due paragrafi successivi esamineremo due algoritmi di impedimento alle situazioni di stallo.

### 7.5.1 Stato sicuro

Uno stato si dice *sicuro* se il sistema è in grado di assegnare risorse a ciascun processo (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo. Più formalmente, un sistema si trova in stato sicuro solo se esiste una **sequenza sicura**. Una sequenza di processi  $\langle P_1, P_2, \dots, P_n \rangle$  è una sequenza sicura per lo stato di assegnazione attuale se, per ogni  $P_i$ , le richieste che  $P_i$  può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i  $P_j$  con  $j < i$ . In questa situazione, se le risorse necessarie al processo  $P_i$  non sono disponibili immediatamente, allora  $P_i$  può attendere che tutti i  $P_j$  abbiano finito, e a quel punto  $P_i$  può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le risorse assegnate e terminare. Quando  $P_i$  termina,  $P_{i+1}$  può ottenere le risorse richieste, e così via. Se non esiste una sequenza di questo tipo, lo stato del sistema si dice *non sicuro*.

Uno stato sicuro non è di stallo. Viceversa, uno stato di stallo è uno stato non sicuro; comunque non tutti gli stati non sicuri sono stati di stallo (Figura 7.5). Uno stato non sicuro *potrebbe* condurre a uno stallo. Finché lo stato rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e di stallo. In uno stato non sicuro il sistema operativo non può impedire ai processi di richiedere risorse in modo da causare uno stallo: ciò che accade negli stati non sicuri dipende dal comportamento dei processi.

Per illustrare meglio quel che si è detto sopra, si consideri un sistema con 12 unità a nastri magnetici e 3 processi:  $P_0$ ,  $P_1$  e  $P_2$ . Il processo  $P_0$  può richiedere 10 unità a nastri, il processo  $P_1$  può richiederne 4 e il processo  $P_2$  può richiedere fino a 9. Supponendo che all'istante  $t_0$  il processo  $P_0$  possieda 5 unità a nastri, e che i processi  $P_1$  e  $P_2$  ne possiedano 2 ciascuno, restano libere 3 unità a nastri.

	<i>Richieste massime</i>	<i>Unità possedute</i>
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

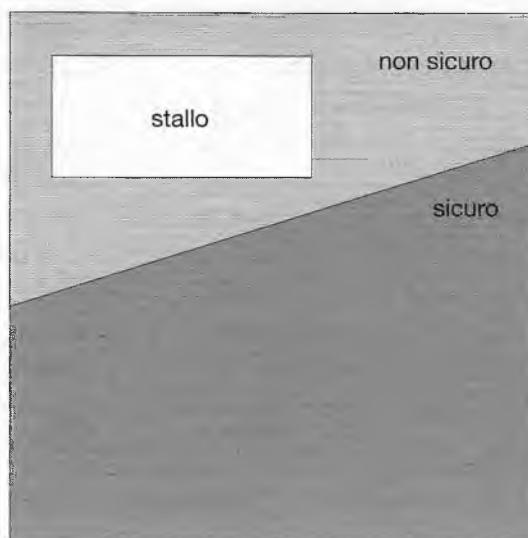


Figura 7.5 Spazi degli stati sicuri, non sicuri e di stallo.

All'istante  $t_0$ , il sistema si trova in uno stato sicuro. La sequenza  $\langle P_1, P_0, P_2 \rangle$  soddisfa la condizione di sicurezza, poiché al processo  $P_1$  si possono assegnare immediatamente tutte le unità a nastri richieste, che saranno poi restituite (a questo punto sono disponibili 5 unità a nastri), quindi il processo  $P_0$  può avere tutte le unità a nastri richieste e restituirle (il sistema ha 10 unità a nastri disponibili) e infine il processo  $P_2$  potrebbe avere tutte le sue unità a nastri e restituirle (sono disponibili tutte e 12 le unità a nastri).

Un sistema può passare da uno stato sicuro a uno stato non sicuro. Si supponga che all'istante  $t_1$  il processo  $P_2$  richieda un'ulteriore unità a nastri e che questa gli sia assegnata: il sistema non si trova più nello stato sicuro. A questo punto, si possono assegnare tutte le unità a nastri richieste soltanto al processo  $P_1$ . Al momento della restituzione, il sistema avrà solo 4 unità a nastri disponibili. Poiché al processo  $P_0$  sono assegnate 5 unità a nastri, ma il numero massimo è 10, il processo può richiederne altre 5, ma poiché queste non sono disponibili il processo  $P_0$  deve attendere. Analogamente, il processo  $P_2$  può richiedere altre 6 unità a nastri ed è costretto ad attendere; il risultato è una situazione di stallo. L'errore è stato commesso nel soddisfare la richiesta di un'ulteriore unità a nastri fatta dal processo  $P_2$ . Se  $P_2$  avesse atteso il termine di uno degli altri processi e il conseguente rilascio delle sue risorse, la situazione di stallo si sarebbe potuta evitare.

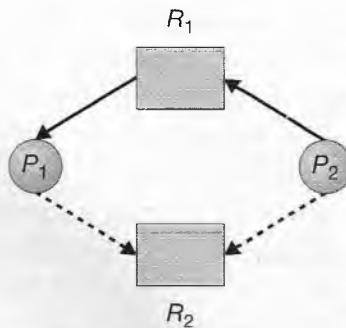
Dato il concetto di stato sicuro, si possono definire algoritmi che permettano di evitare le situazioni di stallo. L'idea è semplice: è sufficiente assicurare che il sistema rimanga sempre in uno stato sicuro. Il sistema si trova inizialmente in uno stato sicuro; ogni volta che un processo richiede una risorsa, il sistema deve stabilire se la risorsa è attualmente disponibile oppure se il processo debba attendere. Si soddisfa la richiesta solo se l'assegnazione lascia il sistema in uno stato sicuro.

In questo modo, se un processo richiede una risorsa attualmente disponibile può essere comunque costretto ad attendere. Quindi, l'utilizzo delle risorse può essere inferiore rispetto a quello che si avrebbe in assenza di un algoritmo per evitare le situazioni di stallo.

### 7.5.2 Algoritmo con grafo di assegnazione delle risorse

Quando il sistema per l'assegnazione delle risorse è tale che ogni tipo di risorsa ha una sola istanza, per evitare le situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse definito nel Paragrafo 7.2.2. Oltre agli archi di richiesta e di assegnazione, si introduce un nuovo tipo di arco, l'arco di reclamo (*claim edge*). Un **arco di reclamo**  $P_i \rightarrow R_j$  indica che il processo  $P_i$  può richiedere la risorsa  $R_j$  in un qualsiasi momento futuro. Quest'arco ha la stessa direzione dell'arco di richiesta, ma si rappresenta con una linea tratteggiata. Quando il processo  $P_i$  richiede la risorsa  $R_j$ , l'arco di reclamo  $P_i \rightarrow R_j$  diventa un arco di richiesta. Analogamente, quando  $P_i$  rilascia la risorsa  $R_j$ , l'arco di assegnazione  $R_j \rightarrow P_i$  diventa un arco di reclamo  $P_i \rightarrow R_j$ . Occorre sottolineare che le risorse devono essere reclamate a priori nel sistema. Ciò significa che prima che il processo  $P_i$  inizi l'esecuzione, tutti i suoi archi di reclamo devono essere già inseriti nel grafo di assegnazione delle risorse. Questa condizione si può indebolire permettendo l'aggiunta di un arco di reclamo  $P_i \rightarrow R_j$  al grafo solo se tutti gli archi associati al processo  $P_i$  sono archi di reclamo.

Si supponga che il processo  $P_i$  richieda la risorsa  $R_j$ . La richiesta si può soddisfare solo se la conversione dell'arco di richiesta  $P_i \rightarrow R_j$  nell'arco di assegnazione  $R_j \rightarrow P_i$  non causa la formazione di un ciclo nel grafo di assegnazione delle risorse. Occorre ricordare che la sicurezza si controlla con un algoritmo di rilevamento dei cicli, e che un algoritmo per il rilevamento di un ciclo in questo grafo richiede un numero di operazioni dell'ordine di  $n^2$ , dove con  $n$  si indica il numero dei processi del sistema.



**Figura 7.6** Grafo di assegnazione delle risorse per evitare le situazioni di stallo.

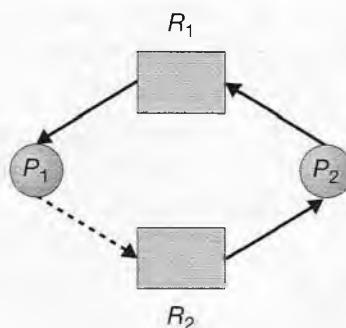
Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro, e il processo  $P_i$  deve attendere che si soddisfino le sue richieste.

Per illustrare questo algoritmo si consideri il grafo di assegnazione delle risorse della Figura 7.6. Si supponga che  $P_2$  richieda  $R_2$ . Sebbene sia attualmente libera,  $R_2$  non può essere assegnata a  $P_2$ , poiché, com'è evidenziato nella Figura 7.7, quest'operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro. Se, a questo punto,  $P_1$  richiedesse  $R_2$ , si avrebbe uno stallo.

### 7.5.3 Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. L'algoritmo per evitare le situazioni di stallo qui descritto, pur essendo meno efficiente dello schema con grafo di assegnazione delle risorse, si può applicare a tali sistemi, ed è noto col nome di **algoritmo del banchiere**. Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario per assicurare che la banca non assegna mai tutto il denaro disponibile, poiché, se ciò avvenisse, non potrebbe più soddisfare le richieste di tutti i suoi clienti.

Quando si presenta, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui necessita. Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato sicuro. Se si rispetta tale condizione, si assegnano le risorse, altrimenti il processo deve attendere che qualche altro processo ne rilasci un numero sufficiente.



**Figura 7.7** Stato non sicuro in un grafo di assegnazione delle risorse.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Sia  $n$  il numero di processi del sistema e  $m$  il numero dei tipi di risorsa. Sono necessarie le seguenti strutture dati.

- ◆ **Disponibili.** Un vettore di lunghezza  $m$  indica il numero delle istanze disponibili per ciascun tipo di risorsa;  $Disponibili[j] = k$ , significa che sono disponibili  $k$  istanze del tipo di risorsa  $R_j$ .
- ◆ **Massimo.** Una matrice  $n \times m$  definisce la richiesta massima di ciascun processo;  $Massimo[i, j] = k$  significa che il processo  $P_i$  può richiedere un massimo di  $k$  istanze del tipo di risorsa  $R_j$ .
- ◆ **Assegnate.** Una matrice  $n \times m$  definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni processo;  $Assegnate[i, j] = k$  significa che al processo  $P_i$  sono correntemente assegnate  $k$  istanze del tipo di risorsa  $R_j$ .
- ◆ **Necessità.** Una matrice  $n \times m$  indica la necessità residua di risorse relativa a ogni processo;  $Necessità[i, j] = k$  significa che il processo  $P_i$ , per completare il suo compito, può avere bisogno di altre  $k$  istanze del tipo di risorsa  $R_j$ . Si osservi che  $Necessità[i, j] = Massimo[i, j] - Assegnate[i, j]$ .

Col trascorrere del tempo, queste strutture dati variano sia nelle dimensioni sia nei valori.

Per semplificare la presentazione dell'algoritmo del banchiere, si usano le seguenti notazioni: supponendo che  $X$  e  $Y$  siano vettori di lunghezza  $n$ , si può affermare che  $X \leq Y$  se e solo se  $X[i] \leq Y[i]$  per ogni  $i = 1, 2, \dots, n$ . Ad esempio, se  $X = (1, 7, 3, 2)$  e  $Y = (0, 3, 2, 1)$ , allora  $Y \leq X$ ;  $Y < X$  se  $Y \leq X$  e  $Y \neq X$ .

Tutte le righe delle matrici *Assegnate* e *Necessità* sono considerabili vettori e si possono chiamare rispettivamente *Assegnate<sub>i</sub>* e *Necessità<sub>i</sub>*. Il vettore *Assegnate<sub>i</sub>* specifica le risorse correntemente assegnate al processo  $P_i$ , mentre il vettore *Necessità<sub>i</sub>* specifica le risorse che il processo  $P_i$  può ancora richiedere per completare il suo compito.

### 7.5.3.1 Algoritmo di verifica della sicurezza

L'algoritmo utilizzato per scoprire se il sistema è o non è in uno stato sicuro si può descrivere come segue.

1. Siano *Lavoro* e *Fine* vettori di lunghezza rispettivamente  $m$  e  $n$ , e inizializza *Lavoro* = *Disponibili* e *Fine*[ $i$ ] = falso, per  $i = 1, 2, \dots, n - 1$ ;
2. cerca un indice  $i$  tale che valgano contemporaneamente le seguenti relazioni:
  - a)  $Fine[i] == \text{falso}$
  - b)  $Necessità_i \leq Lavoro$
 se tale  $i$  non esiste, esegue il passo 4;
3.  $Lavoro = Lavoro + Assegnate_i$   
 $Fine[i] = \text{vero}$   
 torna al passo 2
4. se  $Fine[i] == \text{vero}$  per ogni  $i$ , allora il sistema è in uno stato sicuro.

Per determinare se uno stato è sicuro tale algoritmo può richiedere un numero di operazioni dell'ordine di  $m \times n^2$ .

### 7.5.3.2 Algoritmo di richiesta delle risorse

Si descrive ora l'algoritmo che determina se le richieste possano essere garantite in uno stato sicuro.

Sia  $Richieste_i$  il vettore delle richieste per il processo  $P_i$ . Se  $Richieste_i[j] == k$ , allora il processo  $P_i$  richiede  $k$  istanze del tipo di risorsa  $R_j$ . Se il processo  $P_i$  fa una richiesta di risorse, si svolgono le seguenti azioni:

1. se  $Richieste_i \leq Necessità_i$ , esegue il passo 2, altrimenti riporta una condizione d'errore, poiché il processo ha superato il numero massimo di richieste;
2. se  $Richieste_i \leq Disponibili$ , esegue il passo 3, altrimenti  $P_i$  deve attendere poiché le risorse non sono disponibili;
3. simula l'assegnazione al processo  $P_i$  delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:

$$Disponibili = Disponibili - Richieste_i$$

$$Assegnate_i = Assegnate_i + Richieste_i$$

$$Necessità_i = Necessità_i - Richieste_i$$

Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al processo  $P_i$  si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro,  $P_i$  deve attendere  $Richieste_i$  e si ripristina il vecchio stato di assegnazione delle risorse.

### 7.5.3.3 Un esempio

Illustriamo infine l'uso dell'algoritmo del banchiere, considerando un sistema con cinque processi, da  $P_0$  a  $P_4$ , e tre tipi di risorse:  $A$ ,  $B$ , e  $C$ . Il tipo di risorse  $A$  ha 10 istanze, il tipo  $B$  ha 5 istanze e il tipo  $C$  ha 7 istanze. Si supponga che all'istante  $T_0$  si sia verificata la seguente "istantanea":

	<i>Assegnate</i>	<i>Massimo</i>	<i>Disponibili</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	7 5 3	3 3 2
$P_1$	2 0 0	3 2 2	
$P_2$	3 0 2	9 0 2	
$P_3$	2 1 1	2 2 2	
$P_4$	0 0 2	4 3 3	

Il contenuto della matrice  $Necessità$  è definito come  $Massimo - Assegnate$ :

	<i>Necessità</i>
	<i>A B C</i>
$P_0$	7 4 3
$P_1$	1 2 2
$P_2$	6 0 0
$P_3$	0 1 1
$P_4$	4 3 1

Il sistema si trova attualmente in uno stato sicuro; infatti, la sequenza  $\langle P_1, P_3, P_4, P_2, P_0 \rangle$  soddisfa i criteri di sicurezza. Si supponga ora che il processo  $P_1$  richieda un'altra istanza del tipo di risorsa  $A$  e due istanze del tipo  $C$ , quindi  $Richieste_1 = (1, 0, 2)$ . Per stabilire se questa richiesta si possa esaudire immediatamente si verifica la condizione  $Richieste_1 \leq Disponibili$  (vale a dire  $(1, 0, 2) \leq (3, 3, 2)$ ), che risulta vera, quindi, supponendo che questa richiesta sia stata soddisfatta, si ottiene il seguente nuovo stato:

	<i>Assegnate</i>	<i>Necessità</i>	<i>Disponibili</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	7 4 3	2 3 0
$P_1$	3 0 2	0 2 0	
$P_2$	3 0 2	6 0 0	
$P_3$	2 1 1	0 1 1	
$P_4$	0 0 2	4 3 1	

Occorre stabilire se questo nuovo stato del sistema sia sicuro; a tale scopo si esegue l'algoritmo di verifica della sicurezza da cui risulta che la sequenza  $\langle P_1, P_3, P_4, P_0, P_2 \rangle$  rispetta il requisito di sicurezza. Quindi si può soddisfare immediatamente la richiesta del processo  $P_1$ .

Tuttavia, dovrebbe essere chiaro che, quando il sistema si trova in questo stato, una richiesta di  $(3, 3, 0)$  da parte di  $P_4$  non si può soddisfare finché non siano disponibili le risorse. Inoltre, una richiesta di  $(0, 2, 0)$  da parte di  $P_0$  non si potrebbe soddisfare neanche se le risorse fossero disponibili, poiché lo stato risultante sarebbe non sicuro. L'implementazione dell'algoritmo del banchiere è lasciata come esercizio di programmazione.

## 7.6 Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo per prevenire o evitare situazioni di stallo, è possibile che si verifichi effettivamente. In tal caso il sistema deve fornire i seguenti algoritmi:

- un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo.

Nell'analisi seguente sono trattati i suddetti argomenti che riguardano sia sistemi con una sola istanza di ciascun tipo di risorsa, sia sistemi con più istanze. Tuttavia, a questo punto, occorre notare che uno schema di rilevamento e ripristino richiede un carico che include sia i costi operativi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, sia i potenziali costi connessi al ripristino da una situazione di stallo.

### 7.6.1 Istanza singola di ciascun tipo di risorsa

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento di situazioni di stallo che fa uso di una variante del grafo di assegnazione delle risorse, detta **grafo d'attesa**, ottenuta dal grafo di assegnazione delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra i processi.

Più precisamente, un arco da  $P_i$  a  $P_j$  del grafo d'attesa implica che il processo  $P_i$  attende che il processo  $P_j$  rilasci una risorsa di cui  $P_i$  ha bisogno. Un arco  $P_i \rightarrow P_j$  esiste nel grafo d'attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi  $P_i \rightarrow R_q$  e  $R_q \rightarrow P_j$  per qualche risorsa  $R_q$ . Nella Figura 7.8 sono illustrati un grafo di assegnazione delle risorse e il corrispondente grafo d'attesa.

Come in precedenza, nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve *conservare* il grafo d'attesa e *invocare periodicamente un algoritmo* che cerchi un ciclo all'interno del grafo. L'algoritmo per il rilevamento di un ciclo all'interno di un grafo richiede un numero di operazioni dell'ordine di  $n^2$ , dove con  $n$  si indica il numero dei vertici del grafo.

## 7.6.2 Più istanze di ciascun tipo di risorsa

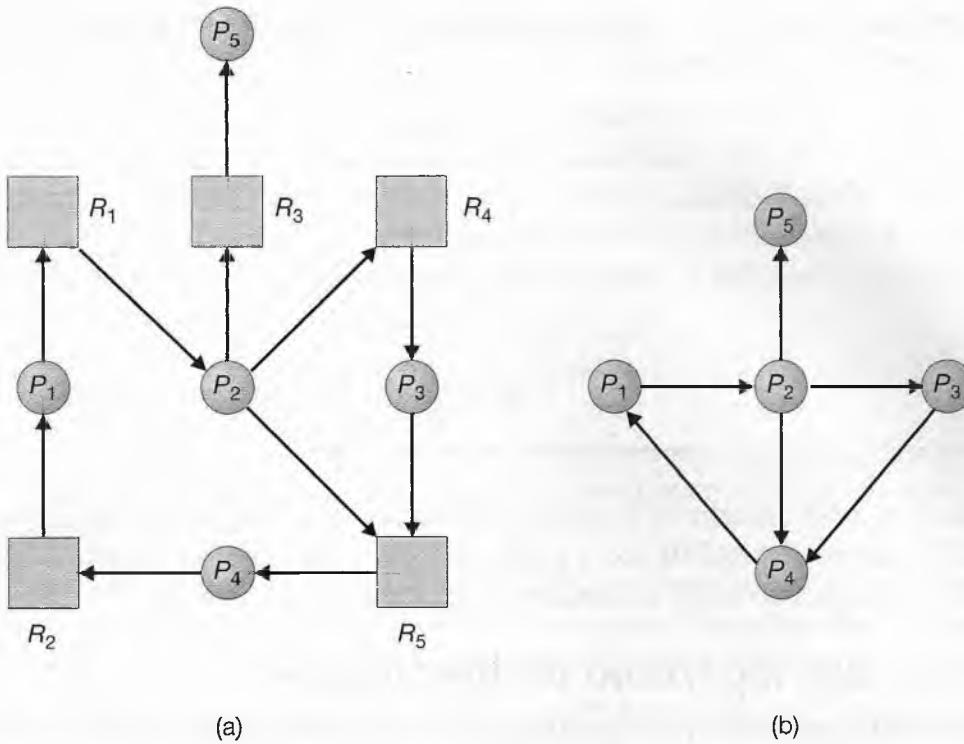
Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. Il seguente algoritmo di rilevamento di situazioni di stallo è, invece, applicabile a tali sistemi. Esso si serve di strutture dati variabili nel tempo, simili a quelle adoperate nell'algoritmo del banchiere (Paragrafo 7.5.3).

- ◆ *Disponibili*. Vettore di lunghezza  $m$  che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- ◆ *Assegnate*. Matrice  $n \times m$  che definisce il numero delle istanze di ciascun tipo di risorse correntemente assegnate a ciascun processo.
- ◆ *Richieste*. Matrice  $n \times m$  che indica la richiesta attuale di ciascun processo. Se  $\text{Richieste}[i, j] = k$ , significa che il processo  $P_i$  sta richiedendo altre  $k$  istanze del tipo di risorsa  $R_j$ .

La relazione  $\leq$  tra due vettori si definisce come nel Paragrafo 7.5.3. Per semplificare la notazione, le righe delle matrici *Assegnate* e *Richieste* si trattano come vettori e, nel seguito, sono indicate rispettivamente come *Assegnate<sub>i</sub>* e *Richieste<sub>i</sub>*. L'algoritmo di rilevamento descritto indaga su ogni possibile sequenza di assegnazione per i processi che devono ancora essere completati. Questo algoritmo si può confrontare con quello del banchiere del Paragrafo 7.5.3.

1. Siano *Lavoro* e *Fine* vettori di lunghezza rispettivamente  $m$  e  $n$ , inizializza *Lavoro* = *Disponibili*, per  $i = 1, 2, \dots, n$ , se *Assegnate<sub>i</sub>* ≠ 0, allora *Fine*[ $i$ ] = falso, altrimenti *Fine*[ $i$ ] = vero;
2. cerca un indice  $i$  tale che valgano contemporaneamente le seguenti relazioni:
  - a) *Fine*[ $i$ ] == falso
  - b) *Richieste<sub>i</sub>* ≤ *Lavoro*
 se tale  $i$  non esiste, esegue il passo 4;
3. *Lavoro* = *Lavoro* + *Assegnate<sub>i</sub>*;  
*Fine*[ $i$ ] = vero  
 torna al passo 2
4. se *Fine*[ $i$ ] == falso per qualche  $i$ ,  $0 \leq i < n$ , allora il sistema è in stallo, inoltre, se *Fine*[ $i$ ] == falso, il processo  $P_i$  è in stallo.

Tale algoritmo richiede un numero di operazioni dell'ordine di  $m \times n^2$  per controllare se il sistema è in stallo.



**Figura 7.8** (a) Grafo di assegnazione delle risorse; (b) grafo d'attesa corrispondente.

Può meravigliare che le risorse del processo  $P_i$  siano richieste (passo 3) non appena risulta valida la condizione  $Richieste_i \leq Lavoro$  (passo 2.b). Tale condizione garantisce che  $P_i$  non è correntemente coinvolto in uno stallo, quindi, assumendo un atteggiamento ottimistico, si suppone che  $P_i$  non intenda richiedere altre risorse per completare il proprio compito, e che restituisca presto tutte le risorse. Se non si rispetta l'ipotesi fatta, si può verificare uno stallo, che sarà rilevato quando si richiamerà nuovamente l'algoritmo di rilevamento.

Per illustrare questo algoritmo, si consideri un sistema con cinque processi, da  $P_0$  a  $P_4$ , e tre tipi di risorse:  $A$ ,  $B$ , e  $C$ . Il tipo di risorsa  $A$  ha 7 istanze, il tipo  $B$  ha 2 istanze e il tipo  $C$  ne ha 6. Si supponga di avere, all'istante  $T_0$ , il seguente stato di assegnazione delle risorse:

	<i>Assegnate</i>	<i>Richieste</i>	<i>Disponibili</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
$P_0$	0 1 0	0 0 0	0 0 0
$P_1$	2 0 0	2 0 2	
$P_2$	3 0 3	0 0 0	
$P_3$	2 1 1	1 0 0	
$P_4$	0 0 2	0 0 2	

Il sistema non è in stallo. Infatti eseguendo l'algoritmo per la sequenza  $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ , risulta  $Fine[i] == \text{vero}$  per ogni  $i$ .

Si supponga ora che il processo  $P_2$  richieda un'altra istanza di tipo  $C$ . La matrice *Richieste* viene modificata come segue:

	<i>Richieste</i>		
	<i>A</i>	<i>B</i>	<i>C</i>
$P_0$	0	0	0
$P_1$	2	0	2
$P_2$	0	0	1
$P_3$	1	0	0
$P_4$	0	0	2

Ora il sistema è in stallo. Anche se si possono reclamare le risorse possedute dal processo  $P_0$ , il numero delle risorse disponibili non è sufficiente per soddisfare le richieste degli altri processi, quindi si verifica uno stallo composto dai processi  $P_1$ ,  $P_2$ ,  $P_3$  e  $P_4$ .

### 7.6.3 Uso dell'algoritmo di rilevamento

Per sapere quando è necessario ricorrere all'algoritmo di rilevamento si devono considerare i seguenti fattori:

1. *frequenza* (presunta) con la quale si verifica uno stallo;
2. *numero* dei processi che sarebbero influenzati da tale stallo.

Se le situazioni di stallo sono frequenti, è necessario ricorrere spesso all'algoritmo per il loro rilevamento. Le risorse assegnate a processi in stallo rimangono inattive fino all'eliminazione dello stallo. Inoltre, il numero dei processi coinvolti nel ciclo di stallo può aumentare.

Le situazioni di stallo si verificano solo quando qualche processo fa una richiesta che non si può soddisfare immediatamente; può essere una richiesta che chiude una catena di processi in attesa. Il caso estremo è quello nel quale l'algoritmo di rilevamento si usa ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. In questo caso non si identifica soltanto il gruppo di processi in stallo, ma anche il processo che ha "causato" lo stallo, anche se, in verità, ciascuno dei processi in stallo è un elemento del ciclo all'interno del grafo di assegnazione delle risorse, quindi tutti i processi sono, congiuntamente, responsabili dello stallo. Se esistono tipi di risorsa diversi, una singola richiesta può causare più cicli nel grafo delle risorse, ciascuno dei quali è completato dalla richiesta più recente ed è causato da un processo identificabile.

Naturalmente, l'uso dell'algoritmo di rilevamento per ogni richiesta aumenta notevolmente il carico nei termini di tempo di calcolo. Un'alternativa meno dispendiosa è quella in cui l'algoritmo di rilevamento s'invoca a intervalli meno frequenti, per esempio una volta ogni ora, oppure ogni volta che l'utilizzo della CPU scende sotto il 40 per cento, poiché uno stallo può rendere inefficienti le prestazioni del sistema e quindi causare una drastica riduzione dell'utilizzo della CPU. Non è conveniente richiedere l'algoritmo di rilevamento in momenti arbitrari, poiché nel grafo delle risorse possono coesistere molti cicli e, normalmente, non si può dire quale fra i tanti processi in stallo abbia "causato" lo stallo.

## 7.7 Ripristino da situazioni di stallo

Una situazione di stallo si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza dello stallo, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico dalla situazione di stallo. Uno stallo si può eliminare in due modi: il primo prevede semplicemente la terminazione di uno o più processi per interrompere l'attesa circolare; il secondo esercita la prelazione su alcune risorse in possesso di uno o più processi in stallo.

### 7.7.1 Terminazione di processi

Per eliminare le situazioni di stallo attraverso la terminazione di processi si possono adoperare due metodi; in entrambi il sistema recupera immediatamente tutte le risorse assegnate ai processi terminati.

- ◆ **Terminazione di tutti i processi in stallo.** Chiaramente questo metodo interrompe il ciclo di stallo, ma l'operazione è molto onerosa; questi processi possono aver già fatto molti calcoli i cui risultati si annullano e probabilmente dovranno essere ricalcolati.
- ◆ **Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo.** Questo metodo causa un notevole carico, poiché, dopo aver terminato ogni processo, si deve impiegare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo.

Procurare la terminazione di un processo può essere un'operazione tutt'altro che semplice: se il processo si trova nel mezzo dell'aggiornamento di un file, la terminazione lascia il file in uno stato scorretto; analogamente, se il processo si trova nel mezzo di una stampa di dati, prima di stampare un lavoro successivo, il sistema deve reimpostare la stampante riportandola a uno stato corretto.

Se si adopera il metodo di terminazione parziale, dato un insieme di processi in stallo occorre determinare quale processo, o quali processi, costringere alla terminazione nel tentativo di sciogliere la situazione di stallo. Analogamente ai problemi di scheduling della CPU, si tratta di scegliere un criterio. Le considerazioni sono essenzialmente economiche: si dovrebbero arrestare i processi la cui terminazione causa il minimo costo. Sfortunatamente, il termine *minimo costo* non è preciso. La scelta dei processi è influenzata da diversi fattori, tra cui i seguenti:

1. la priorità dei processi;
2. il tempo trascorso dalla computazione e il tempo ancora necessario per completare i compiti assegnati ai processi;
3. la quantità e il tipo di risorse impiegate dai processi (ad esempio, se si può avere facilmente la prelazione sulle risorse);
4. la quantità di ulteriori risorse di cui i processi hanno ancora bisogno per completare i propri compiti;
5. il numero di processi che si devono terminare;
6. il tipo di processi: interattivi o a lotti.

### 7.7.2 Prelazione su risorse

Per eliminare uno stallo si può esercitare la prelazione sulle risorse: le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l'interruzione del ciclo di stallo.

Se per gestire le situazioni di stallo s'impiega la prelazione, si devono considerare i seguenti problemi.

1. **Selezione di una vittima.** Occorre stabilire quali risorse e quali processi si devono sotoporre a prelazione. Come per la terminazione dei processi, è necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero delle risorse possedute da un processo in stallo, e la quantità di tempo già spesa durante l'esecuzione da un processo in stallo.
2. **Ristabilimento di un precedente stato sicuro.** Occorre stabilire che cosa fare con un processo cui è stata sottratta una risorsa. Poiché è stato privato di una risorsa necessaria, la sua esecuzione non può continuare normalmente, quindi deve essere ricondotto a un precedente stato sicuro dal quale essere riavviato. Poiché, in generale, è difficile stabilire quale stato sia sicuro, la soluzione più semplice consiste nel terminare il processo e quindi riavviarlo. Certamente, è più efficace ristabilire un precedente stato sicuro del processo che sia sufficiente allo scioglimento della situazione di stallo, ma questo metodo richiede che il sistema mantenga più informazioni sullo stato di tutti i processi in esecuzione.
3. **Attesa indefinita.** È necessario assicurare che non si verifichino situazioni d'attesa indefinita, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo.

In un sistema in cui la scelta della vittima avviene soprattutto secondo fattori di costo, può accadere che si scelga sempre lo stesso processo; in questo caso il processo non riesce mai a completare il suo compito; si tratta di una situazione d'attesa indefinita che si deve affrontare in qualsiasi sistema concreto. Chiaramente è necessario assicurare che un processo possa essere prescelto come vittima solo un numero finito (e ridotto) di volte; la soluzione più diffusa prevede l'inclusione del numero di terminazioni, per successivi riavvii, tra i fattori di costo.

## 7.8 Sommario

---

Uno stallo si verifica quando in un insieme di processi ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In linea di principio, i metodi di gestione delle situazioni di stallo sono tre:

- ◆ impiegare un protocollo che prevenga o eviti le situazioni di stallo, assicurando che il sistema non vi entri mai;
- ◆ permettere al sistema di entrare in stallo e quindi effettuarne il ripristino;
- ◆ ignorare del tutto il problema fingendo che nel sistema non si verifichino mai situazioni di stallo. Tale "soluzione" è adottata dalla maggior parte dei sistemi operativi, compresi UNIX e Windows.

Una situazione di stallo può presentarsi solo se all'interno del sistema si verificano contemporaneamente quattro condizioni necessarie: *mutua esclusione, possesso e attesa, impossibilità di prelazione e attesa circolare*. Per prevenire il verificarsi di situazioni di stallo è necessario assicurare che almeno una delle suddette condizioni necessarie non sia soddisfatta.

Un altro metodo per evitare le situazioni di stallo, meno rigido dell'algoritmo di prevenzione, consiste nel disporre di informazioni a priori su come ciascun processo intende impiegare le risorse. L'algoritmo del banchiere, per esempio, richiede la conoscenza del numero massimo di ogni classe di risorse che può essere richiesto da ciascun processo. Servendosi di queste informazioni si può definire un algoritmo per evitare le situazioni di stallo.

Se un sistema non usa alcun protocollo per assicurare che non avvengano situazioni di stallo, è necessario un metodo di rilevamento e ripristino. Per stabilire se si sia verificato uno stallo, è necessario ricorrere a un algoritmo di rilevamento; nel caso in cui si rilevi uno stallo, il sistema deve attuare il ripristino terminando alcuni processi coinvolti, oppure sottraendo risorse a qualcuno di essi.

Nel caso in cui la prelazione sia applicata alla prevenzione dello stallo, occorre tenere conto di tre questioni: la selezione di una vittima, il ristabilimento di un precedente stato sicuro e l'attesa indefinita. Nei sistemi che selezionano le vittime principalmente sulla base di fattori di costo, possono presentarsi situazioni di attesa indefinita, causando l'impossibilità del processo scelto di portare a termine il suo compito.

Infine, è stato sostenuto da parte di alcuni ricercatori che nessuno di questi approcci sia in grado, da solo, di risolvere appropriatamente l'intera gamma di problemi legati all'allocazione delle risorse nei sistemi operativi. Ciononostante, questi metodi di base possono essere combinati in modo da fornire strategie ottimali per ogni classe di risorse dei sistemi.

## Esercizi pratici

- 7.1 Elencate tre esempi di stallo di processi che non siano correlati all'informatica.
- 7.2 Supponete che un sistema sia in uno stato non sicuro. Mostrate che è possibile che i processi completino l'esecuzione senza entrare in una situazione di stallo.
- 7.3 Un metodo possibile per prevenire gli stalli consiste nel disporre di una singola risorsa di ordine più elevato che deve essere richiesta prima di ogni altra risorsa. Lo stallo è possibile, ad esempio, se i thread multipli tentano di accedere agli oggetti di sincronizzazione *A…E*. (Tali oggetti di sincronizzazione possono includere mutex, semafori, variabili condition, e simili.) Possiamo prevenire gli stalli aggiungendo un sesto oggetto *F*. Ogni volta che un thread vuole acquisire un lock di sincronizzazione per ciascun oggetto *A…E*, deve prima acquisire il lock per l'oggetto *F*. Questa soluzione è conosciuta come **inclusione (containment)**: i lock per gli oggetti *A…E* sono inclusi all'interno del lock per l'oggetto *F*. Paragonate questo schema con quello ad attesa circolare del Paragrafo 7.4.4.
- 7.4 Dimostrate che l'algoritmo di sicurezza presentato nel Paragrafo 7.5.3 richieda un numero di operazioni dell'ordine di  $m \times n^2$ .
- 7.5 Considerate un sistema informatico che esegua 5000 task al mese e non disponga né di uno schema per prevenire gli stalli né di uno schema per evitarli. Gli stalli si verificano circa due volte al mese e l'operatore deve terminare e rieseguire circa 10 task per ogni stallo. Ogni task costa circa 2 dollari (in tempo del processore), e i task terminati tendono a essere circa a metà del loro lavoro quando vengono interrotti.

Un programmatore di sistema ha stimato che un algoritmo per evitare gli stalli (come l'algoritmo del banchiere) potrebbe essere installato nel sistema con un incremento del 10% circa del tempo medio di esecuzione per task. Siccome la macchina ha attualmente il 30% di periodo d'inattività, tutti i 5000 task al mese potrebbero ancora essere eseguiti, nonostante il tempo di completamento aumenterebbe in media del 20 per cento circa.

- a. Quali sono i vantaggi dell'installazione dell'algoritmo per evitare gli stalli?
- b. Quali sono gli svantaggi dell'installazione dell'algoritmo per evitare gli stalli?

7.6 Un sistema può individuare quali dei suoi processi stanno morendo? Se la vostra risposta è affermativa, spiegate com'è possibile. Se è negativa, spiegate come il sistema può trattare il problema dell'attesa indefinita.

7.7 Considerate la seguente politica di allocazione delle risorse. Le richieste e i rilasci di risorse sono sempre possibili. Se una richiesta di risorse non può essere soddisfatta perché queste non sono disponibili, si controllano tutti i processi bloccati in attesa di risorse. Se un processo bloccato possiede le risorse desiderate, queste possono essere prelevate perché vengano date al processo che le richiede. Il vettore delle risorse attese dal processo bloccato è aggiornato in modo da includere le risorse sottratte al processo.

Si consideri ad esempio un sistema con tre tipi di risorse e il vettore *Disponibili* inizializzato a (4,2,2). Se il processo  $P_0$  richiede (2,2,1), le ottiene. Se  $P_1$  richiede (1,0,1), le ottiene. Poi, se  $P_0$  chiede (0,0,1) viene bloccato (risorsa non disponibile). Se  $P_2$  ora chiede (2,0,0) ottiene la risorsa disponibile (1,0,0) e quella che è stata assegnata a  $P_0$  (poiché  $P_0$  è bloccato). Il vettore *Assegnate* di  $P_0$  scende a (1,2,1) e il suo vettore *Necessità* passa a (1,0,1).

- a. Possono verificarsi stalli? Se la risposta è affermativa, fornite un esempio. In caso di risposta negativa, specificate quale condizione necessaria non si può verificare.
- b. Può verificarsi un blocco indefinito? Spiegate la risposta.

7.8 Supponete di aver codificato l'algoritmo di sicurezza per evitare gli stalli e che ora vi sia richiesto di implementare l'algoritmo di rilevamento delle situazioni di stallo. È possibile farlo utilizzando semplicemente il codice dell'algoritmo di sicurezza e ridefinendo  $Massimo_i = Attesa_i + Assegnate_i$ , dove  $Attesa_i$  è un vettore che specifica le risorse attese dal processo  $i$  e  $Assegnate_i$  è definito come nel Paragrafo 7.5? Motivate la risposta.

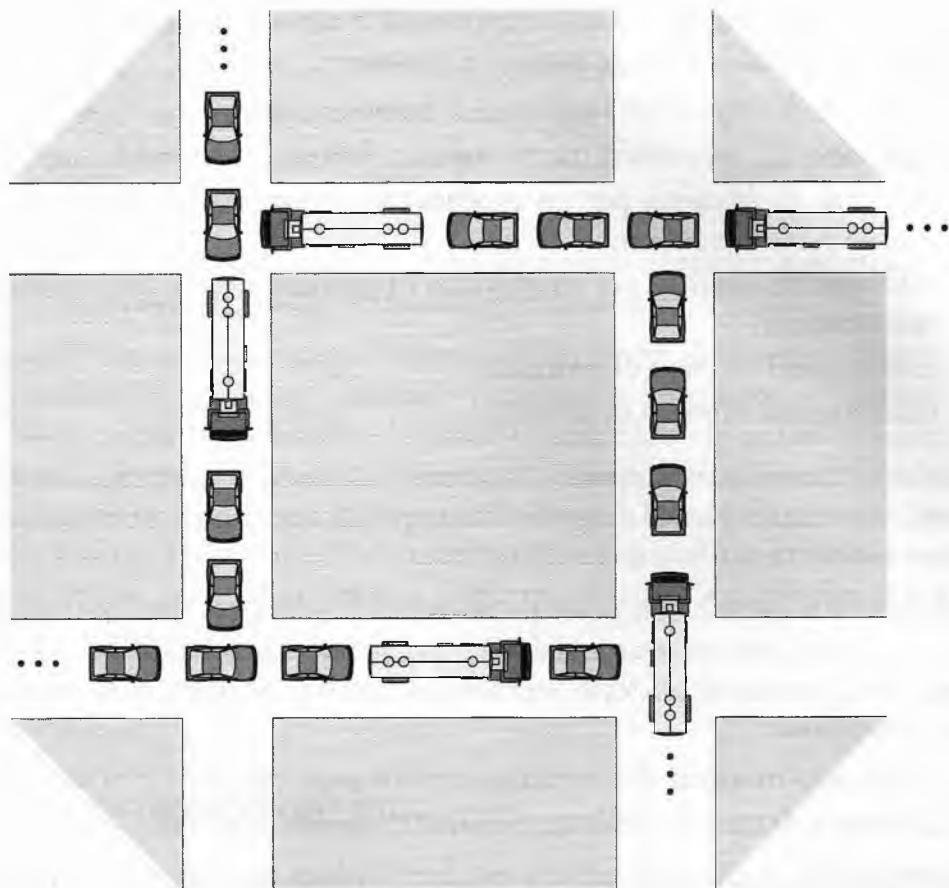
7.9 È possibile che uno stallo coinvolga un solo processo? Argomentate la risposta.

## Esercizi

7.10 Considerate lo stallo di traffico automobilistico illustrato nella Figura 7.9.

- a. dimostrate che le quattro condizioni necessarie per lo stallo valgono anche in questo esempio;
- b. fissate una semplice regola che eviti le situazioni di stallo in questo sistema.

7.11 Considerate la situazione di stallo che potrebbe verificarsi con il problema dei cinque filosofi, allorquando essi ricevano le bacchette, una per volta. Chiarite come le quat-



**Figura 7.9** Stallo di traffico automobilistico per l’Esercizio 7.10.

tre condizioni necessarie per lo stallo abbiano validità in questo contesto. Discutete come, eliminando una delle quattro condizioni, lo stallo potrebbe essere evitato.

- 7.12 Un rimedio al pericolo dello stallo è rappresentato dalla disponibilità di una sola risorsa, di grado superiore, che deve essere richiesta prima di qualunque altra. Se, a esempio, svariati thread tentano di accedere agli oggetti  $A, \dots, E$ , lo stallo è possibile. (Tali oggetti possono includere mutex, semafori, variabili condizionali, e così via.) Ma con l’aggiunta di un sesto oggetto  $F$ , possiamo neutralizzare il pericolo: ogni volta un thread intenda acquisire il lock che regola la sincronizzazione di un oggetto  $A, \dots, E$ , esso dovrà prima impossessarsi del lock relativo a  $F$ . Questa soluzione è nota come **contenimento**. I lock relativi agli oggetti  $A, \dots, E$  sono “contenuti” nel lock dell’oggetto  $F$ . Confrontate questo modello con quello dell’attesa circolare, descritto nel Paragrafo 7.4.4.
- 7.13 Confrontate il modello dell’attesa circolare con le varie strategie di neutralizzazione dello stallo (quali l’algoritmo del banchiere) rispetto alle seguenti problematiche:
- carico di esecuzione;
  - produttività del sistema.
- 7.14 In un sistema reale, né le risorse disponibili, né le richieste di risorse da parte dei processi sono coerenti in lunghi periodi (mesi). Le risorse si guastano o sono sostituite, nuovi processi vanno e vengono, si acquistano e si aggiungono nuove risorse al sistema. Se le situazioni di stallo si controllano con l’algoritmo del banchiere, dite quali

tra le seguenti modifiche si possono apportate con sicurezza, senza introdurre la possibilità di situazioni di stallo, e in quali circostanze:

- aumento di *Disponibili* (aggiunta di nuove risorse);
- riduzione di *Disponibili* (risorsa rimossa definitivamente dal sistema);
- aumento di *Massimo* per un processo (il processo necessita di più risorse di quante siano permesse);
- riduzione di *Massimo* per un processo (il processo decide che non ha bisogno di tante risorse);
- aumento del numero di processi;
- riduzione del numero di processi.

- 7.15 Considerate un sistema composto da quattro risorse dello stesso tipo condivise da tre processi, ciascuno dei quali necessita di non più di due risorse. Dimostrate che non si possono verificare situazioni di stallo.
- 7.16 Considerate un sistema composto da  $m$  risorse dello stesso tipo, condivise da  $n$  processi. Le risorse possono essere richieste e rilasciate dai processi solo una alla volta. Dimostrate che non si possono verificare situazioni di stallo se si rispettano le seguenti condizioni:
- la richiesta massima di ciascun processo è compresa tra 1 e  $m$  risorse;
  - la somma di tutte le richieste massime è minore di  $m + n$ .
- 7.17 Considerate il problema dei cinque filosofi, supponendo che le bacchette siano disposte al centro del tavolo e che ciascuna coppia di bacchette possa essere usata da un filosofo. Supponete che le richieste di bacchette avvengano una per volta. Descrivete una semplice regola che determini se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.
- 7.18 Considerate lo stesso contesto del problema precedente. Ipotizzate, ora, che ogni filosofo richieda tre bacchette per mangiare e che, anche qui, le richieste siano avanzate separatamente. Descrivete delle semplici regole che determinino se una certa richiesta possa essere soddisfatta senza causare stallo, data la distribuzione corrente delle bacchette tra i filosofi.
- 7.19 Potete ricavare un algoritmo del banchiere che sia adatto a un solo tipo di risorsa dall'algoritmo generale del banchiere, semplicemente riducendo la dimensione dei vari array di 1. Dimostrate con un esempio che l'algoritmo del banchiere generale non può essere implementato applicando individualmente a ciascun tipo di risorsa l'algoritmo per un solo tipo di risorsa.
- 7.20 Considerate la seguente “istantanea” di un sistema:

	<i>Assegnate</i>	<i>Massimo</i>	<i>Disponibili</i>
	<i>A B C D</i>	<i>A B C D</i>	<i>A B C D</i>
$P_0$	0 0 1 2	0 0 1 2	1 5 2 0
$P_1$	1 0 0 0	1 7 5 0	
$P_2$	1 3 5 4	2 3 5 6	
$P_3$	0 6 3 2	0 6 5 2	
$P_4$	0 0 1 4	0 6 5 6	

Impiegando l'algoritmo del banchiere rispondete alle seguenti domande:

- dite qual è il contenuto della matrice *Necessità*;
- dite se il sistema è in uno stato sicuro;
- dite se a fronte di una richiesta per  $(0, 4, 2, 0)$ , fatta dal processo  $P_1$ , essa può essere soddisfatta immediatamente.

- 7.21 Quale ottimistica ipotesi è alla base dell'algoritmo di rilevamento delle situazioni di stallo? Come potrebbe essere invalidata questa ipotesi?
- 7.22 Un ponte a corsia unica collega i due villaggi di North Tunbridge e South Tunbridge, nel Vermont. Gli agricoltori dei due villaggi usano tale ponte per portare i loro prodotti nella cittadina confinante. Il ponte rimane bloccato se un contadino diretto a nord e uno diretto a sud vi salgono contemporaneamente. (I contadini del Vermont sono ostinati: non farebbero mai marcia indietro.) Progettate un algoritmo che eviti lo stallo tramite i semafori. Non curatevi, all'inizio, dell'attesa indefinita, ossia la situazione in cui i contadini diretti a nord impediscono a quelli diretti a sud di salire sul ponte, o viceversa.
- 7.23 Modificate la soluzione al problema dell'Esercizio 7.22 in modo da eliminare l'attesa indefinita.

## Problemi di programmazione

- 7.24 Elaborate un programma multithread che implementi l'algoritmo del banchiere illustrato nel Paragrafo 7.5.3. Create  $n$  thread che richiedano alla banca le risorse, e le disimpegnino. Il banchiere soddisferà la richiesta solo se ciò può avvenire lasciando il sistema in uno stato sicuro. Potete scegliere, per il programma, i thread di Pthreads o di Win32. È importante che l'accesso ai dati condivisi sia protetto dall'accesso corrente; a tale scopo, si possono impiegare i lock mutex, disponibili sia nella API di Pthreads sia in quella di Win32. L'argomento, per entrambe queste librerie, è trattato nel progetto alla fine del Capitolo 6.

## 7.9 Note bibliografiche

[Dijkstra 1965a] è stato uno dei primi e più influenti studiosi del fenomeno dello stallo. [Holt 1972] è stato il primo a formalizzare la nozione di stallo tramite un modello basato sulla teoria dei grafi simile a quello presentato in questo capitolo; nello stesso lavoro, si è inoltre occupato dell'attesa indefinita. [Hyman 1985] presenta l'esempio di situazione di stallo tratto dalla legislazione del Kansas. Uno studio recente sulla gestione dello stallo è fornito da [Levine 2003].

I diversi algoritmi di prevenzione sono stati suggeriti da [Havender 1968], che ha progettato lo schema di ordinamento delle risorse per il sistema IBM OS/360.

L'algoritmo del banchiere è stato sviluppato per un singolo tipo di risorsa da [Dijkstra 1965a], ed è stato esteso a più tipi di risorse da [Habermann 1969]. Gli Esercizi 7.15 e 7.16 sono tratti da [Holt 1971].

L'algoritmo di rilevamento delle situazioni di stallo per istanze multiple di un tipo di risorsa, descritto nel Paragrafo 7.6.2, è presentato in [Coffman et al. 1971].

# Gestione della memoria

Scopo principale di un sistema di calcolo è eseguire programmi; durante l'esecuzione, i programmi e i dati cui essi accedono devono trovarsi almeno parzialmente in memoria centrale.

Per migliorare l'utilizzo della CPU e la velocità con cui essa risponde ai suoi utenti, il calcolatore deve tenere in memoria parecchi processi. Esistono molti metodi di gestione della memoria e l'efficacia di ciascun algoritmo dipende dalla situazione. La scelta di un metodo di gestione della memoria, per un sistema specifico, dipende da molti fattori, in particolar modo dall'*architettura* del sistema; ogni algoritmo richiede infatti uno specifico supporto hardware.

## Capitolo 8

# Memoria centrale



### OBIETTIVI

- Descrizione dettagliata delle diverse organizzazioni dei dispositivi per la gestione della memoria.
- Analisi delle diverse tecniche di gestione della memoria, comprese paginazione e segmentazione.
- Descrizione accurata di Intel Pentium, che supporta sia la segmentazione pura sia la segmentazione con paginazione.

Nel Capitolo 5 si spiega come sia possibile condividere la CPU tra un insieme di processi. Uno dei risultati dello scheduling della CPU consiste nella possibilità di migliorare sia l'utilizzo della CPU sia la rapidità con cui il calcolatore risponde ai propri utenti; per ottenere questo aumento delle prestazioni occorre tenere in memoria parecchi processi: la memoria deve, cioè, essere *condivisa*.

In questo capitolo si presentano i diversi metodi di gestione della memoria. Gli algoritmi di gestione della memoria variano dal metodo iniziale sulla nuda macchina ai metodi di paginazione e segmentazione. Ogni metodo presenta vantaggi e svantaggi. La scelta di un metodo specifico di gestione della memoria dipende da molti fattori, in particolar modo dall'architettura del sistema, infatti molti algoritmi richiedono specifiche caratteristiche dell'architettura; progetti recenti integrano in modo molto stretto hardware e il sistema operativo.

## 8.1 Introduzione

Come abbiamo visto nel Capitolo 1, la memoria è fondamentale nelle operazioni di un moderno sistema di calcolo; consiste in un ampio vettore di parole o byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; decodificata (e ciò può comportare il prelievo di operandi dalla memoria) ed eseguita sugli eventuali operandi; i risultati si possono salvare in memoria. La memoria *vede* soltanto un flusso d'indirizzi di memoria, e non *sà* come sono generati (contatore di programma, indicizzazione, riferimenti indiretti, indirizzamenti immediati e così

via), oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare *come* un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

L'analisi che sviluppiamo nel seguito comprende una panoramica degli aspetti basilari dell'architettura, dagli spazi di indirizzi logici e fisici agli indirizzi fisici reali, offrendo una loro analisi comparativa. Seguono temi quali caricamento dinamico, collegamento e condivisione di librerie.

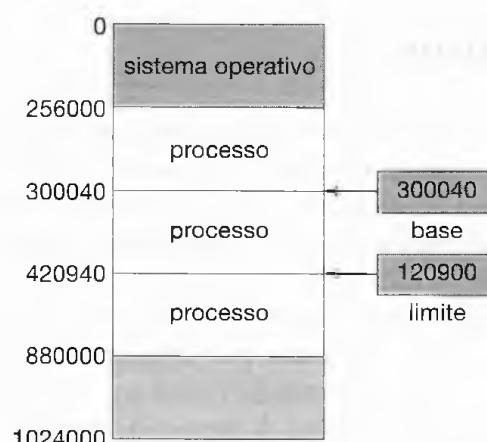
### 8.1.1 Dispositivi essenziali

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. Vi sono istruzioni macchina che accettano gli indirizzi di memoria come argomenti, ma nessuna accetta gli indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi per la memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro.

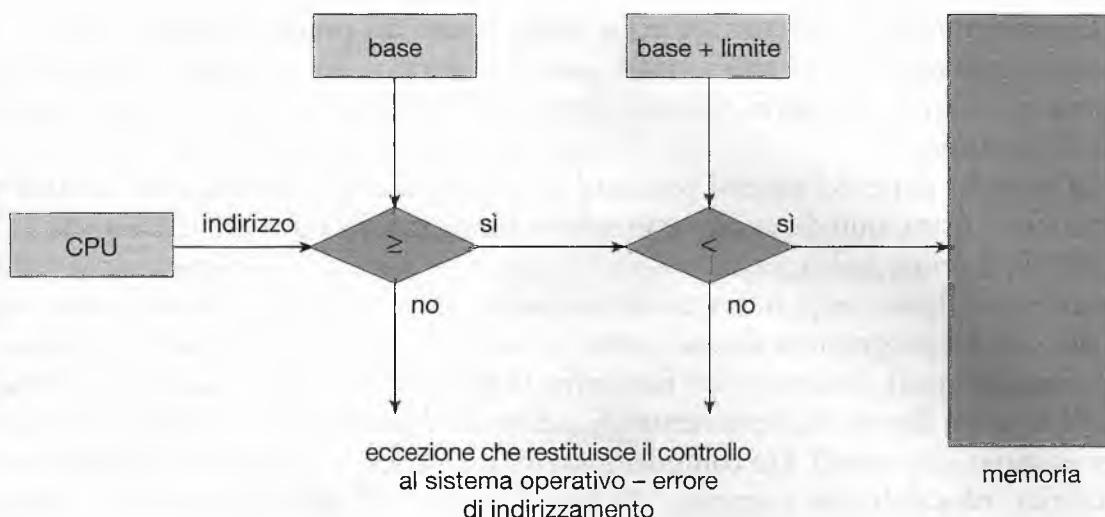
I registri incorporati nella CPU sono accessibili, in genere, nell'arco di un ciclo dell'orologio di sistema. Molte CPU sono capaci di decodificare istruzioni ed effettuare semplici operazioni sui contenuti dei registri alla velocità di una o più operazioni per ciclo. Ciò non vale per la memoria centrale, cui si accede attraverso una transazione sul bus della memoria. Nei casi in cui l'accesso alla memoria richieda molti cicli d'orologio, il processore entra necessariamente in **stallo (stall)**, poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra CPU e memoria centrale. Un buffer di memoria, detto **cache**, è in grado di conciliare le differenti velocità (Paragrafo 1.8.3).

Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica: è anche necessario proteggere il sistema operativo dall'accesso dei processi utenti, e salvaguardare i processi utenti l'uno dall'altro. Tale protezione deve essere messa in atto a livello dei dispositivi; come si vedrà lungo tutto il capitolo, essa può essere realizzata con meccanismi diversi. In questo paragrafo evidenziamo una delle possibili implementazioni.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato. A tal fine, occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti **registri base** e **registri limite**.



**Figura 8.1** I registri base e limite definiscono lo spazio degli indirizzi logici.



**Figura 8.2** Protezione hardware degli indirizzi tramite registri base e limite.

te, come illustrato nella Figura 8.1. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Ad esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso alle locazioni di memoria di indirizzi compresi tra 300040 e 420939, estremi inclusi.

Per mettere in atto il meccanismo di protezione, la CPU confronta *ciascun* indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di memoria riservata ad altri utenti comporta l'invio di un segnale di eccezione che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (Figura 8.2). Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati, sia del sistema operativo sia degli altri utenti.

Solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente nella modalità di sistema, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce la medesima operazione ai programmi utenti.

Grazie all'esecuzione nella modalità di sistema, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utenti nelle aree di memoria a loro riservate; di generare copie del contenuto di queste regioni di memoria (*dump*) a scopi diagnostici, qualora si verifichino errori; di modificare i parametri delle chiamate di sistema, e così via.

### 8.1.2 Associazione degli indirizzi

In genere un programma risiede in un disco in forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito all'interno di un processo. Secondo il tipo di gestione della memoria adoperato, durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti forma la coda d'ingresso (*input queue*).

La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e nel caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria.

La maggior parte dei sistemi consente ai processi utenti di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio d'indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere 00000. Quest'assetto influisce sugli indirizzi che un programma utente può usare. Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari stadi, alcuni dei quali possono essere facoltativi (Figura 8.3), in cui gli indirizzi sono rappresentabili in modi diversi. Generalmente gli indirizzi del programma sorgente sono simbolici (per esempio, contatore). Un compilatore di solito associa (*bind*) questi indirizzi simbolici a indirizzi rilocabili (per esempio, "14 byte dall'inizio di questo modulo"). L'editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (per esempio, 74014). Ogni associazione rappresenta una corrispondenza da uno spazio d'indirizzi a un altro.

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi fase del seguente percorso.

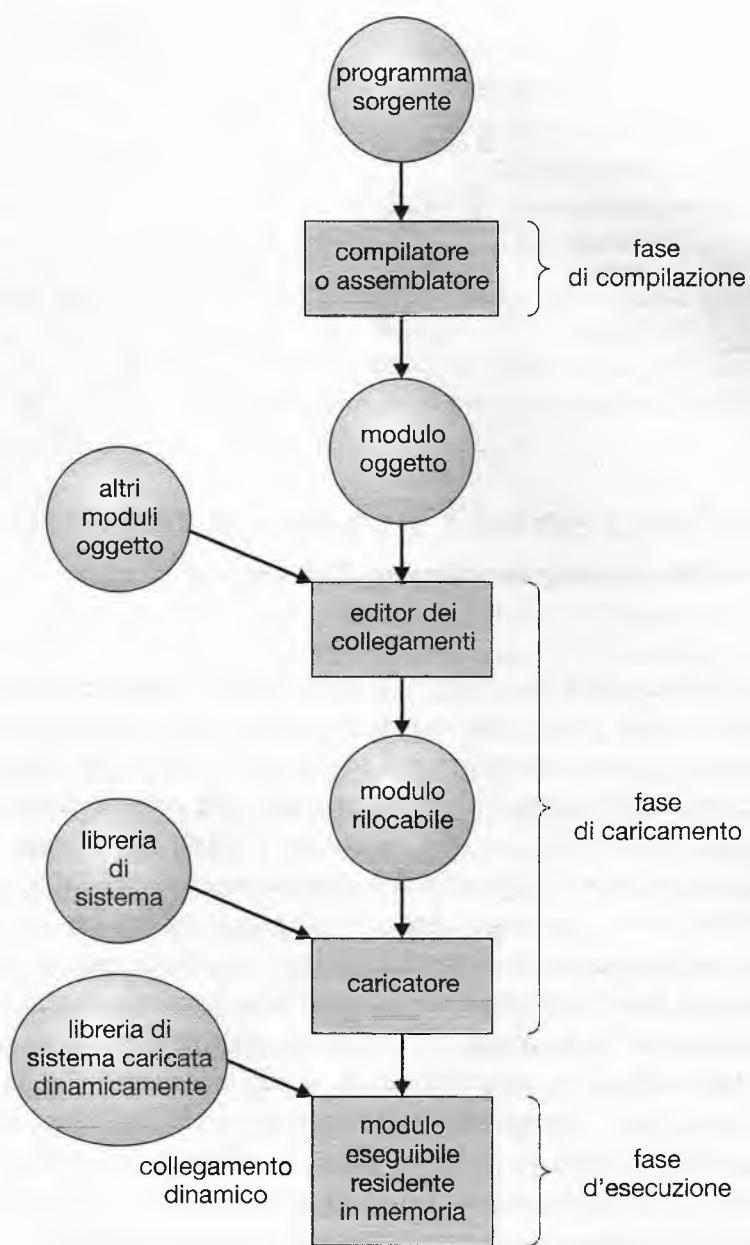
- ◆ **Compilazione.** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare codice assoluto. Se, per esempio, è noto a priori che un processo utente inizia alla locazione  $r$ , anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice. I programmi per MS-DOS nel formato identificato dall'estensione .COM sono collegati al tempo di compilazione.
- ◆ **Caricamento.** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare codice rilocabile. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- ◆ **Esecuzione.** Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. Per realizzare questo schema sono necessarie specifiche caratteristiche dell'architettura; questo argomento è trattato nel Paragrafo 8.1.3. La maggior parte dei sistemi operativi d'uso generale impiega questo metodo.

Una gran parte di questo capitolo è dedicata alla spiegazione di come i vari tipi di associazione degli indirizzi si possano realizzare efficacemente in un calcolatore e, inoltre, alla discussione delle caratteristiche dell'architettura appropriate alla realizzazione di queste funzioni.

### 8.1.3 Spazi di indirizzi logici e fisici a confronto

Un indirizzo generato dalla CPU di solito si indica come indirizzo logico, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel registro dell'indirizzo di memoria (*memory address register*, MAR) di solito si indica come indirizzo fisico.

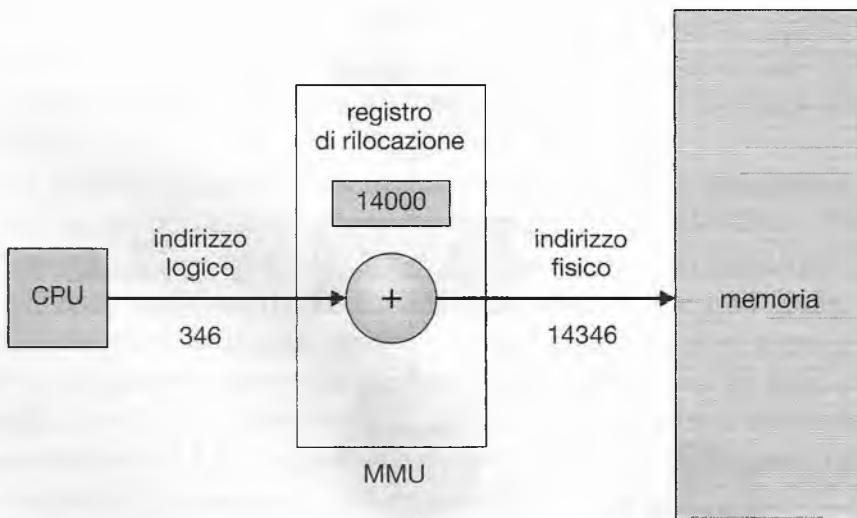
I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con i metodi di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce,



**Figura 8.3** Fasi di elaborazione per un programma utente.

di solito, agli indirizzi logici col termine **indirizzi virtuali**; in questo testo si usano tali termini in modo intercambiabile. L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**. Quindi, con lo schema di associazione degli indirizzi nella fase d'esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*memory-management unit*, MMU). Come si illustra nei Paragrafi dal 8.3 al 8.7, si può scegliere tra diversi metodi di realizzazione di tale associazione; di seguito s'illustra un semplice schema di associazione degli indirizzi che impiega una MMU; si tratta di una generalizzazione dello schema con registro di base descritto nel Paragrafo 8.1.1.



**Figura 8.4** Rilocazione dinamica tramite un registro di rilocazione.

Com'è illustrato nella Figura 8.4, il registro di base è ora denominato **registro di rilocazione**: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si *somma* a tale indirizzo il valore contenuto nel registro di rilocazione. Ad esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocalizzato alla locazione 14000; un accesso alla locazione 346 corrisponde alla locazione 14346. Quando il sistema operativo MS-DOS, eseguito sulla famiglia di CPU Intel 80x86, carica ed esegue processi impiega quattro registri di rilocazione.

Il programma utente non considera mai gli indirizzi fisici *reali*. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò semplicemente come un numero. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi *logici*, l'architettura del sistema converte gli indirizzi logici in indirizzi *fisici*. Il collegamento nella fase d'esecuzione è trattato nel Paragrafo 8.1.2. La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (nell'intervallo da 0 a *max*) e gli indirizzi fisici (nell'intervallo da  $r + 0$  a  $r + max$  per un valore di base  $r$ ). L'utente genera solo indirizzi logici e *pensa* che il processo sia eseguito nelle posizioni da 0 a *max*. Il programma utente fornisce indirizzi logici che, prima d'essere usati, si devono far corrispondere a indirizzi fisici.

Il concetto di *spazio d'indirizzi logici* associato a uno *spazio d'indirizzi fisici* separato è fondamentale per una corretta gestione della memoria.

#### 8.1.4 Caricamento dinamico

Per migliorare l'utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono in memoria secondaria in un formato di caricamento rilocalizzabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, si controlla innanzitutto che sia stata caricata, altrimenti si richiama il caricatore di collegamento rilocalizzabile per caricare in memoria la procedura richiesta e ag-

giornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata.

Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura che non si adopera non viene caricata. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola.

Il caricamento dinamico non richiede un intervento particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

### 8.1.5 Collegamento dinamico e librerie condivise

La Figura 8.3 mostra anche le librerie **collegate dinamicamente**. Alcuni sistemi operativi consentono solo il **collegamento statico**, in cui le librerie di sistema del linguaggio sono trattate come qualsiasi altro modulo oggetto e combinate dal caricatore nell'immagine binaria del programma. Il concetto di collegamento dinamico è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell'esecuzione, si differisce il collegamento. Questa caratteristica si usa soprattutto con le librerie di sistema, per esempio le librerie di procedure del linguaggio. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre all'interno dell'immagine eseguibile di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Tutto ciò richiede spazio nei dischi e in memoria centrale.

Con il collegamento dinamico, invece, per ogni riferimento a una procedura di libreria s'inserisce all'interno dell'immagine eseguibile una piccola porzione di codice di riferimento (*stub*), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l'esecuzione, il codice di riferimento controlla se la procedura richiesta è già in memoria, altrimenti provvede a caricarla; in ogni caso tale codice sostituisce se stesso con l'indirizzo della procedura, che viene poi eseguita. In questo modo, quando si raggiunge nuovamente quel segmento del codice, si esegue direttamente la procedura di libreria, senza costi aggiuntivi per il collegamento dinamico. Con questo metodo tutti i processi che usano una libreria del linguaggio si limitano a eseguire la stessa copia del codice della libreria.

Questa caratteristica si può estendere anche agli aggiornamenti delle librerie, per esempio, la correzione di errori. Una libreria si può sostituire con una nuova versione, e tutti i programmi che fanno riferimento a quella libreria usano automaticamente quella. Senza il collegamento dinamico tutti i programmi di questo tipo devono subire una nuova fase di collegamento per accedere alla nuova libreria. Affinché i programmi non eseguano accidentalmente nuove versioni di librerie incompatibili, sia nel programma sia nella libreria si inserisce un'informazione relativa alla versione. È possibile caricare in memoria più di una versione della stessa libreria, ciascun programma si serve dell'informazione sulla versione per decidere quale copia debba usare. Se le modifiche sono di piccola entità, il numero di versione resta invariato; se l'entità delle modifiche diviene rilevante, si aumenta anche il numero di versione. Perciò, solo i programmi compilati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporate nella libreria stessa. I programmi collegati prima dell'installazione della nuova libreria continuano ad avvalersi della vecchia libreria. Questo sistema è noto anche con il nome di **librerie condivise**.

A differenza del caricamento dinamico, il collegamento dinamico richiede generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria. Questo concetto è sviluppato nel contesto della paginazione, nel Paragrafo 8.4.4.

## 8.2 Avvicendamento dei processi (swapping)

Per essere eseguito, un processo deve trovarsi in memoria centrale, ma si può trasferire temporaneamente in **memoria ausiliaria** (*backing store*) da cui si riporta in memoria centrale al momento di riprenderne l'esecuzione. Si consideri, per esempio, un ambiente di multiprogrammazione con un algoritmo circolare (*round-robin*) per lo scheduling della CPU. Trascorso un quanto di tempo, il gestore di memoria scarica dalla memoria il processo appena terminato e carica un altro processo nello spazio di memoria appena liberato; questo procedimento è illustrato nella Figura 8.5 e si chiama **avvicendamento dei processi in memoria** – o, più brevemente, **avvicendamento** o **scambio** (*swapping*). Nel frattempo lo scheduler della CPU assegna un quanto di tempo a un altro processo presente in memoria. Quando esaurisce il suo quanto di tempo, ciascun processo viene scambiato con un altro processo. In teoria il gestore della memoria può avvicendare i processi in modo sufficientemente rapido da far sì che alcuni siano presenti in memoria, pronti per essere eseguiti, quando lo scheduler della CPU voglia riassegnare la CPU stessa. Anche il quanto di tempo deve essere sufficientemente lungo da permettere che un processo, prima d'essere sostituito, esegua quantità ragionevole di calcolo.

Una variante di questo criterio d'avvicendamento dei processi s'impiega per gli algoritmi di scheduling basati sulle priorità. Se si presenta un processo con priorità maggiore, il gestore della memoria può scaricare dalla memoria centrale il processo con priorità inferiore.

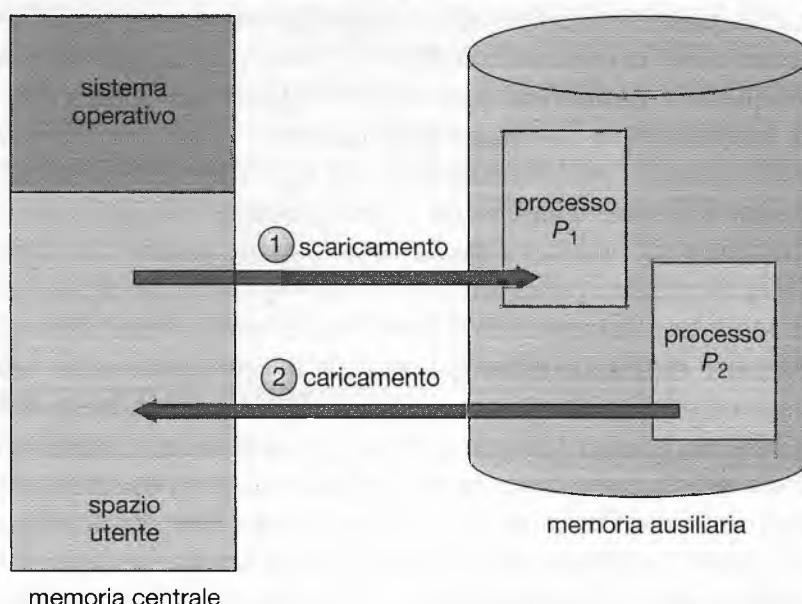


Figura 8.5 Avvicendamento di due processi con un disco come memoria ausiliaria.

re per fare spazio all'esecuzione del processo con priorità maggiore. Quando il processo con priorità maggiore termina, si può ricaricare in memoria quello con priorità minore e continuare la sua esecuzione (questo tipo d'avvicendamento è talvolta chiamato **roll out, roll in**).

Normalmente, un processo che è stato scaricato dalla memoria si deve ricaricare nello spazio di memoria occupato in precedenza. Questa limitazione è dovuta al metodo di associazione degli indirizzi. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si effettua nella fase di assemblaggio o di caricamento, il processo non può essere ricaricato in posizioni diverse. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si compie nella fase d'esecuzione, un processo può essere riversato in uno spazio di memoria diverso, poiché gli indirizzi fisici si calcolano nella fase d'esecuzione.

L'avvicendamento dei processi richiede una memoria ausiliaria. Tale memoria ausiliaria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema mantiene una **coda dei processi pronti** (*ready queue*) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria. Quando lo scheduler della CPU decide di eseguire un processo, richiama il dispatcher, che controlla se il primo processo della coda si trova in memoria. Se non si trova in memoria, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria e vi carica il processo richiesto dallo scheduler della CPU, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.

In un siffatto sistema d'avvicendamento, il tempo di cambio di contesto (*context-switch time*) è abbastanza elevato. Per avere un'idea delle sue dimensioni si pensi a un processo utente di 100 MB e a una memoria ausiliaria costituita da un disco ordinario con velocità di trasferimento di 50 MB al secondo. Il trasferimento effettivo del processo di 100 MB da e in memoria richiede:

$$100 \text{ MB} / 50 \text{ MB al secondo} = 2 \text{ secondi}$$

Ipotizzando che la latenza media sia di 8 millisecondi, l'operazione richiede 2008 millisecondi. Poiché l'avvicendamento richiede lo scaricamento e il caricamento di un processo, il tempo totale è di circa 4016 millisecondi.

Occorre notare che la maggior parte del tempo d'avvicendamento è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla *quantità* di memoria interessata. In un calcolatore con 4 GB di memoria centrale e un sistema operativo residente di 1 GB, la massima dimensione possibile per un processo utente è di 3 GB. Tuttavia possono essere presenti molti processi utenti con dimensione minore, per esempio 100 MB. Lo scaricamento di un processo di 100 MB può concludersi in 2 secondi, mentre per lo scaricamento di 3 GB sono necessari 60 secondi. Perciò sarebbe utile sapere esattamente quanta memoria *sia* effettivamente usata da un processo utente e non solo quanta questo *potrebbe* teoricamente usarne, poiché in questo caso è necessario scaricare solo quanto è effettivamente utilizzato, riducendo il tempo d'avvicendamento. Affinché questo metodo risulti efficace, l'utente deve tenere informato il sistema su tutte le modifiche apportate ai requisiti di memoria; un processo con requisiti di memoria dinamici deve impiegare chiamate di sistema (`request memory` e `release memory`) per informare il sistema operativo delle modifiche da apportare alla memoria.

L'avvicendamento dei processi è soggetto ad altri vincoli. Per scaricare un processo dalla memoria è necessario essere certi che sia completamente inattivo. Particolare importanza

ha qualsiasi I/O pendente: mentre si vuole scaricare un processo per liberare la memoria, tale processo può essere nell'attesa del completamento di un'operazione di I/O. Tuttavia, se un dispositivo di I/O accede in modo asincrono alle aree di I/O della memoria (*buffer*) utente, il processo non è scaricabile. Si supponga che l'operazione di I/O sia stata accodata, perché il dispositivo era occupato. Se il processo  $P_2$  s'avvicendasse al processo  $P_1$ , l'operazione di I/O potrebbe tentare di usare la memoria che attualmente appartiene al processo  $P_2$ . Questo problema si può risolvere in due modi: non scaricando dalla memoria un processo con operazioni di I/O pendenti, oppure eseguendo operazioni di I/O solo in aree di memoria per l'I/O del sistema operativo. In questo modo i trasferimenti fra tali aree del sistema operativo e la memoria assegnata al processo possono avvenire solo quando il processo è presente in memoria centrale.

L'ipotesi che l'avvicendamento dei processi richieda pochi o nessun movimento delle testine dei dischi merita ulteriori spiegazioni (tale argomento è trattato nel Capitolo 12, che illustra la struttura della memoria secondaria). Affinché il suo uso sia quanto più veloce è possibile, generalmente si assegna l'area d'avvicendamento in una porzione di disco separata da quella riservata al file system.

Attualmente l'avvicendamento semplice si usa in pochi sistemi; richiede infatti un elevato tempo di gestione, e consente un tempo di esecuzione troppo breve per essere considerato una soluzione ragionevole al problema di gestione della memoria. Versioni modificate dell'avvicendamento dei processi si trovano comunque in molti sistemi.

Una sua forma modificata era, per esempio, usata in molte versioni di UNIX: normalmente era disabilitato e si avviava solo nel caso in cui molti processi si fossero trovati in esecuzione superando una quantità limite di memoria. L'avvicendamento dei processi sarebbe stato di nuovo sospeso qualora il carico del sistema fosse diminuito. La gestione della memoria in UNIX è descritta ampiamente nel Paragrafo 21.7.

I primi PC, avendo un'architettura troppo semplice per adottare metodi avanzati di gestione della memoria, eseguivano più processi di grandi dimensioni tramite una versione modificata dell'avvicendamento. Un primo esempio importante è dato dal sistema operativo Windows 3.1 di Microsoft che consente l'esecuzione concorrente di più processi presenti in memoria. Se si carica un nuovo processo, ma lo spazio disponibile in memoria centrale è insufficiente, si trasferisce nel disco un processo già presente in memoria centrale. Non si tratta di vero e proprio avvicendamento dei processi, poiché è l'utente, e non lo scheduler, che decide il momento in cui scaricare un processo in favore di un altro; inoltre, ciascun processo scaricato resta in tale stato fino a quando sarà selezionato, per l'esecuzione, dall'utente. Le versioni successive dei sistemi operativi Microsoft si avvantaggiano delle caratteristiche avanzate delle MMU, attualmente disponibili anche nei PC. Nel Paragrafo 8.4 e nel Capitolo 9 – dedicato alla memoria virtuale – vengono illustrate queste caratteristiche.

## 8.3 Allocazione contigua della memoria

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. In questo paragrafo si tratta l'allocazione contigua della memoria.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utenti. Il sistema operativo si può collocare sia in memoria bassa sia in memoria alta. Il fattore che incide in modo decisivo su tale scelta è generalmente la

posizione del vettore delle interruzioni. Poiché si trova spesso in memoria bassa, i programmati collocano di solito anche il sistema operativo in memoria bassa. Per questo motivo prendiamo in considerazione solo la situazione in cui il sistema operativo risiede in quest'area di memoria.

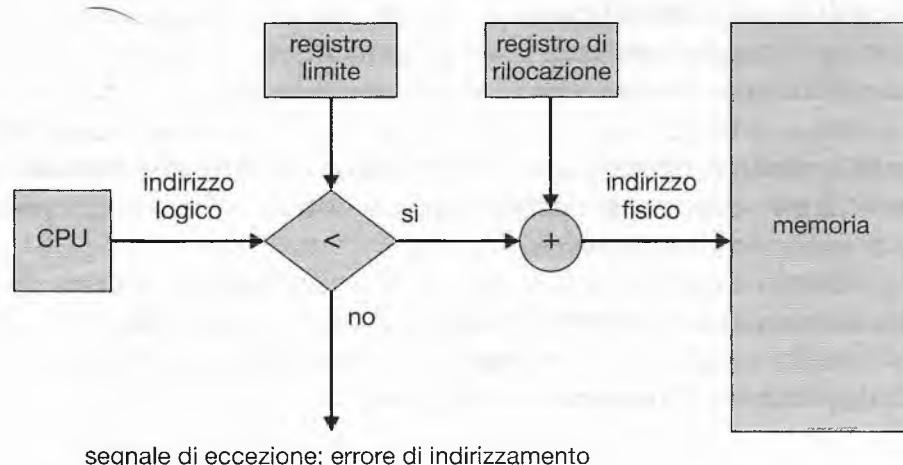
Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con l'**allocazione contigua della memoria**, ciascun processo è contenuto in una singola sezione contigua di memoria.

### 8.3.1 Rilocazione e protezione della memoria

Prima di trattare l'allocazione della memoria, dobbiamo soffermarci sulle questioni relative alla rilocazione e alla protezione della memoria. Tale protezione si può realizzare usando un registro di rilocazione, come si descrive nel Paragrafo 8.1.3, con un registro limite, come si descrive nel Paragrafo 8.1.1. Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600. Con i registri di rilocazione e limite, ogni indirizzo logico deve essere minore del contenuto del registro limite; la MMU fa corrispondere *dinamicamente* l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione (Figura 8.6).

Quando lo scheduler della CPU seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla CPU con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione.

Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi; se uno di questi, o un altro servizio del sistema operativo, non è comunemente usato, è inutile tenerne in memoria codice e dati, poiché lo spazio occupato si potrebbe usare per altri scopi. Talvolta questo codice si chiama codice **transiente** del sistema operativo, poiché s'inserisce secondo le necessità; l'uso di tale codice cambia le dimensioni del sistema operativo durante l'esecuzione del programma.



**Figura 8.6** Registri di rilocazione e limite.

### 8.3.2 Allocazione della memoria

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. Con il **metodo delle partizioni multiple** quando una partizione è libera può essere occupata da un processo presente nella coda d'ingresso; terminato il processo, la partizione diviene nuovamente disponibile per un altro processo. Originariamente questo metodo, detto mft, si usava nel sistema operativo IBM OS/360, ma attualmente non è più in uso. Il metodo descritto di seguito, detto mvt, è una generalizzazione del metodo con partizioni fisse e si usa soprattutto in ambienti d'elaborazione a lotti. Si noti, tuttavia, che molte idee che lo riguardano sono applicabili agli ambienti a partizione del tempo d'elaborazione nei quali si fa uso della segmentazione semplice per la gestione della memoria (Paragrafo 8.6).

Nello schema a partizione fissa il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un **buco** (*hole*). Infine, come avrete modo di vedere, la memoria contiene una serie di buchi di diverse dimensioni.

Quando entrano nel sistema, i processi vengono inseriti in una coda d'ingresso. Per determinare a quali processi si debba assegnare la memoria, il sistema operativo tiene conto dei requisiti di memoria di ciascun processo e della quantità di spazio di memoria disponibile. Quando a un processo si assegna dello spazio, il processo stesso viene caricato in memoria e può quindi competere per il controllo della CPU. Al termine, rilascia la memoria che gli era stata assegnata, e il sistema operativo può impiegarla per un altro processo presente nella coda d'ingresso.

In ogni dato istante è sempre disponibile una lista delle dimensioni dei blocchi liberi e della coda d'ingresso. Il sistema operativo può ordinare la coda d'ingresso secondo un algoritmo di scheduling. La memoria si assegna ai processi della coda finché si possono soddisfare i requisiti di memoria del processo successivo, cioè finché esiste un blocco di memoria (o buco) disponibile, sufficientemente grande da accogliere quel processo. Il sistema operativo può quindi attendere che si renda disponibile un blocco sufficientemente grande, oppure può scorrere la coda d'ingresso per verificare se sia possibile soddisfare le richieste di memoria più limitate di qualche altro processo.

In generale, è sempre presente un *insieme* di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema deve controllare se vi siano processi nell'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi.

Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimensione  $n$  data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti.

- ◆ **First-fit.** Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi.
- ◆ **Best-fit.** Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, sempre che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- ◆ **Worst-fit.** Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, sempre che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio precedente.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte nessuno dei due è chiaramente migliore dell'altro per quel che riguarda l'utilizzo della memoria ma, in genere, first-fit è più veloce.

### 8.3.3 Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**: quando si caricano e si rimuovono i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi. Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi.

Sia che si adotti l'uno o l'altro, l'impiego di un determinato criterio può influire sulla quantità di frammentazione: in alcuni sistemi dà migliori risultati la scelta del primo buco abbastanza grande, in altri dà migliori risultati la scelta del più piccolo tra i buchi abbastanza grandi; inoltre è necessario sapere qual è l'estremità assegnata di un blocco libero (se la parte inutilizzata è quella in alto o quella in basso). A prescindere dal tipo di algoritmo usato, la frammentazione esterna è un problema.

La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. L'analisi statistica dell'algoritmo che segue il criterio di scelta del primo buco abbastanza grande, per esempio, rivela che, pur con una certa ottimizzazione, per  $n$  blocchi assegnati, si perdono altri  $0,5 n$  blocchi a causa della frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria. Questa caratteristica è nota con il nome di **regola del 50 per cento**.

La frammentazione può essere sia interna sia esterna. Si consideri il metodo d'allocazione con più partizioni con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un buco di 2 byte. Il carico necessario per tener traccia di questo buco è sostanzialmente più grande del buco stesso. Il metodo generale prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta. La **frammentazione interna** consiste nella differenza tra questi due numeri; la memoria è interna a una partizione, ma non è in uso.

Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è fatta nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si compie nella fase d'esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti i buchi vengono spostati nell'altra direzione formando un grosso buco di memoria. Questo metodo può essere assai oneroso.

Un'altra possibile soluzione del problema della frammentazione esterna è data dal consentire la non contiguità dello spazio degli indirizzi logici di un processo, permettendo così di assegnare la memoria fisica ai processi dovunque essa sia disponibile. Due tecniche complementari conseguono questo risultato: la paginazione (Paragrafo 8.4) e la segmentazione (Paragrafo 8.6). Queste tecniche si possono anche combinare (Paragrafo 8.7).

## 8.4 Paginazione

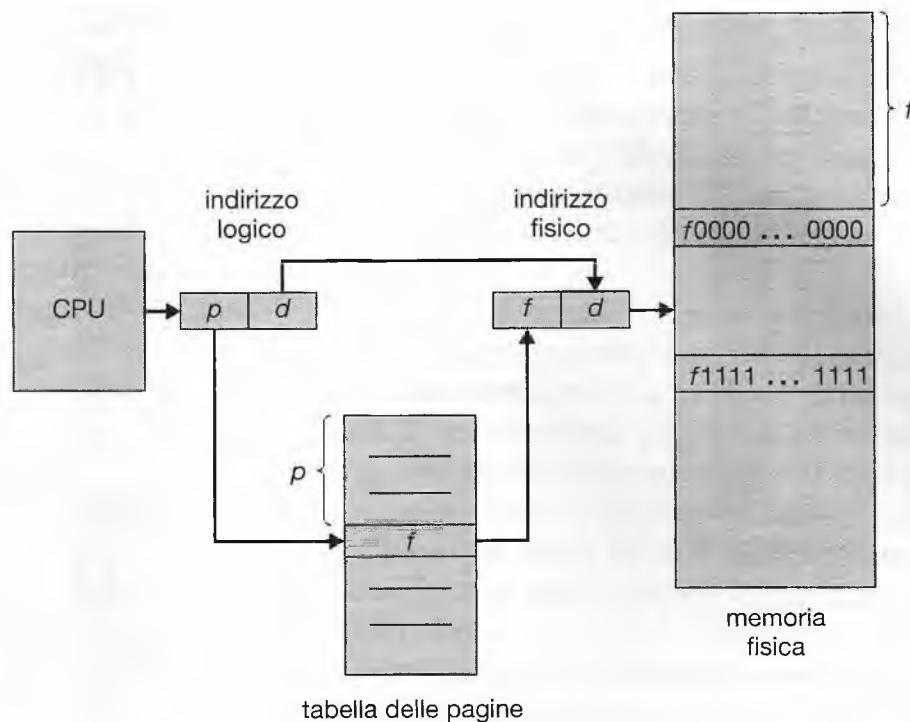
La paginazione è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. Elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria, una questione che riguarda la maggior parte dei metodi di gestione della memoria analizzati. Il problema insorge perché, quando alcuni frammenti di codice o dati residenti in memoria centrale devono essere scaricati, si deve trovare lo spazio necessario in memoria ausiliaria. I problemi di frammentazione relativi alla memoria centrale valgono anche per la memoria ausiliaria, con la differenza che in questo caso l'accesso è molto più lento, quindi è impossibile eseguire la compattazione. Grazie ai vantaggi offerti rispetto ai metodi precedenti, la paginazione nelle sue varie forme è comunemente usata in molti sistemi operativi.

Tradizionalmente, l'architettura del sistema offre specifiche caratteristiche per la gestione della paginazione. Recenti progetti (soprattutto le CPU a 64 bit) prevedono che il sistema di paginazione sia realizzato integrando strettamente l'architettura e il sistema operativo.

### 8.4.1 Metodo di base

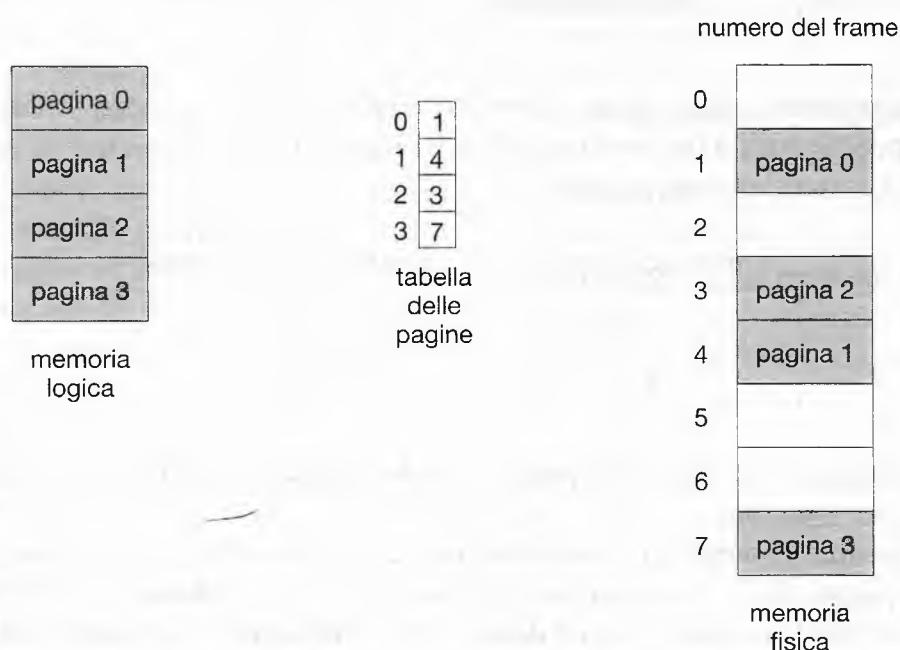
Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione costante, detti anche **frame** o **pagine fisiche**, e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagine**. Quando si deve eseguire un processo, si caricano le sue pagine nei frame disponibili, prendendole dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria.

L'architettura d'ausilio alla paginazione è illustrata nella Figura 8.7; ogni indirizzo generato dalla CPU è diviso in due parti: un **numero di pagina** ( $p$ ), e uno **scostamento** (*offset*) di pagina ( $d$ ). Il numero di pagina serve come indice per la **tabella delle pagine**, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con lo scostamento di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria. La Figura 8.8 illustra il modello di paginazione della memoria.

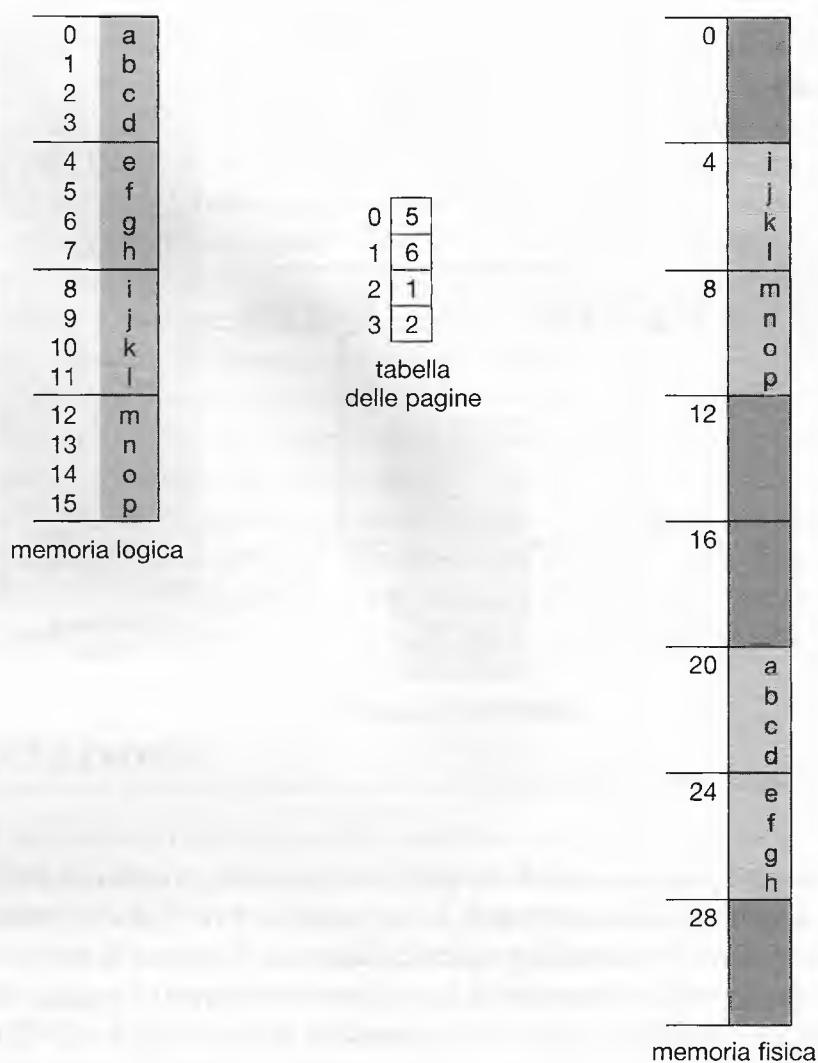


**Figura 8.7** Architettura di paginazione.

La dimensione di una pagina, così come quella di un frame, è definita dall'architettura del calcolatore ed è, in genere, una potenza di 2 compresa tra 512 byte e 16 MB. La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e scostamento di pagina. Se la dimensione dello spazio degli indirizzi logici è  $2^m$  e la dimensione di una pagina è di  $2^n$  unità di indiriz-



**Figura 8.8** Modello di paginazione di memoria logica e memoria fisica.



**Figura 8.9** Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

zamento (byte o parole), allora gli  $m - n$  bit più significativi di un indirizzo logico indicano il numero di pagina, e gli  $n$  bit meno significativi indicano lo scostamento di pagina. L'indirizzo logico ha quindi la forma seguente:

numero di pagina	scostamento di pagina
$p$	$d$
$m - n$	$n$

dove  $p$  è un indice della tabella delle pagine e  $d$  è lo scostamento all'interno della pagina indicata da  $p$ .

Come esempio concreto, anche se minimo, si consideri la memoria illustrata nella Figura 8.9; con pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), si mostra come si faccia corrispondere la memoria vista dall'utente alla memoria fisica. L'indirizzo logico 0 è la pagina 0 con scostamento 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 ( $= (5 \times 4) + 0$ ). All'indirizzo logico 3 (pagina 0, scostamento 3) corrisponde l'indirizzo fisico 23 ( $= (5 \times 4) + 3$ ). Per

quel che riguarda l'indirizzo logico 4 (pagina 1, scostamento 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico  $24 = (6 \times 4) + 0$ . All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

Il lettore può aver notato che la paginazione non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paginazione fa corrispondere un indirizzo fisico. L'uso della tabella delle pagine è simile all'uso di una tabella di registri base (o di rilocazione), uno per ciascun frame.

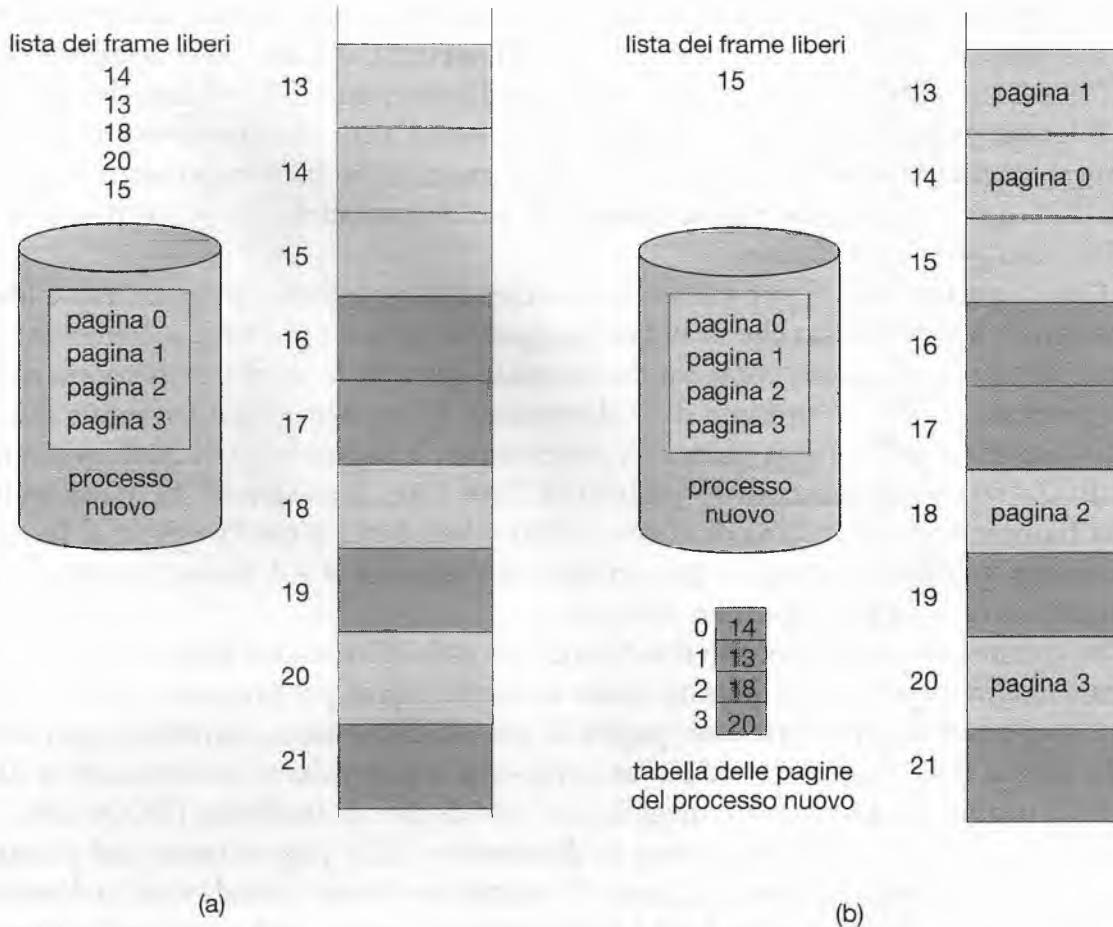
Con la paginazione si può evitare la frammentazione esterna: *qualsiasi* frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle pagine, l'*ultimo* frame assegnato può non essere completamente pieno. Se, per esempio, le pagine sono di 2048 byte, un processo di 72.766 byte necessita di 35 pagine più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di  $2048 - 1086 = 962$  byte. Il caso peggiore si ha con un processo che necessita di  $n$  pagine più un byte: si assegnano  $n + 1$  frame, quindi si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della pagina, ci si deve aspettare una frammentazione interna media di mezza pagina per processo. Questa considerazione suggerisce che conviene usare pagine di piccole dimensioni; tuttavia, a ogni elemento della tabella delle pagine è associato un carico che si può ridurre aumentando le dimensioni delle pagine. Inoltre, con un maggior numero di dati da trasferire, l'I/O su disco è più efficiente (Capitolo 12). Generalmente la dimensione delle pagine cresce col passare del tempo, come i processi, gli insiemi di dati e la memoria centrale; attualmente la dimensione tipica delle pagine è compresa tra 4 KB e 8 KB; in alcuni sistemi può essere anche maggiore. Alcune CPU e alcuni nuclei di sistemi operativi gestiscono anche pagine di diverse dimensioni; il sistema Solaris ad esempio usa pagine di 4 KB o 8 MB, secondo il tipo dei dati memorizzati nelle pagine. Sono in fase di studio e progettazione sistemi di paginazione che consentono la variazione dinamica della dimensione delle pagine.

Ciascun elemento della tabella delle pagine di solito è lungo 4 byte, ma anche questa dimensione può variare; un elemento di 32 bit può puntare a uno dei  $2^{32}$  frame; quindi se un frame è di 4 KB, un sistema con elementi di 4 byte può accedere a  $2^{44}$  byte (o 64 TB) di memoria fisica.

Quando si deve eseguire un processo, si esamina la sua dimensione espressa in pagine. Poiché ogni pagina del processo necessita di un frame, se il processo richiede  $n$  pagine, devono essere disponibili almeno  $n$  frame che, se ci sono, si assegnano al processo stesso. Si carica la prima pagina del processo in uno dei frame assegnati e s'inserisce il numero del frame nella tabella delle pagine relativa al processo in questione. La pagina successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella tabella delle pagine, e così via (Figura 8.10).

Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dall'utente e l'effettiva memoria fisica: il programma utente *vede* la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dall'utente e la memoria fisica è colmata dall'architettura di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti. Queste trasformazioni non sono visibili agli utenti e sono controllate dal sistema operativo. Si noti che



**Figura 8.10** Frame liberi; (a) prima e (b) dopo l'allocazione.

un processo utente, per definizione, non può accedere alle zone di memoria che non gli appartengono. Non ha modo di accedere alla memoria oltre quel che è previsto dalla sua tabella delle pagine; tale tabella riguarda soltanto le pagine che appartengono al processo.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei relativi particolari di allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata **tabella dei frame**, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

Inoltre, il sistema operativo deve sapere che i processi utenti operano nello spazio utente, e tutti gli indirizzi logici si devono far corrispondere a indirizzi fisici. Se un utente usa una chiamata di sistema con un indirizzo di memoria come parametro, per esempio per eseguire un'operazione di I/O all'indirizzo specificato, si deve tradurre tale indirizzo nell'indirizzo fisico corretto. Il sistema operativo conserva una copia della tabella delle pagine per ciascun processo, così come conserva una copia dei valori contenuti nel contatore di programma e nei registri. Questa copia si usa per tradurre gli indirizzi logici in indirizzi fisici ogni volta che il sistema operativo deve esplicitamente un indirizzo fisico a un indirizzo logico. La stessa copia è usata anche dal dispatcher della CPU per impostare l'architettura di paginazione quando a un processo sta per essere assegnata la CPU. La paginazione fa quindi aumentare la durata dei cambi di contesto.

## 8.4.2 Architettura di paginazione

Ogni sistema operativo segue metodi propri per memorizzare le tabelle delle pagine. La maggior parte dei sistemi impiega una tabella delle pagine per ciascun processo. Il PCB contiene, insieme col valore di altri registri, come il registro delle istruzioni, un puntatore alla tabella delle pagine. Per avviare un processo, il dispatcher ricarica i registri utente e imposta i corretti valori della tabella delle pagine fisiche, usando la tabella delle pagine presente in memoria e relativa al processo.

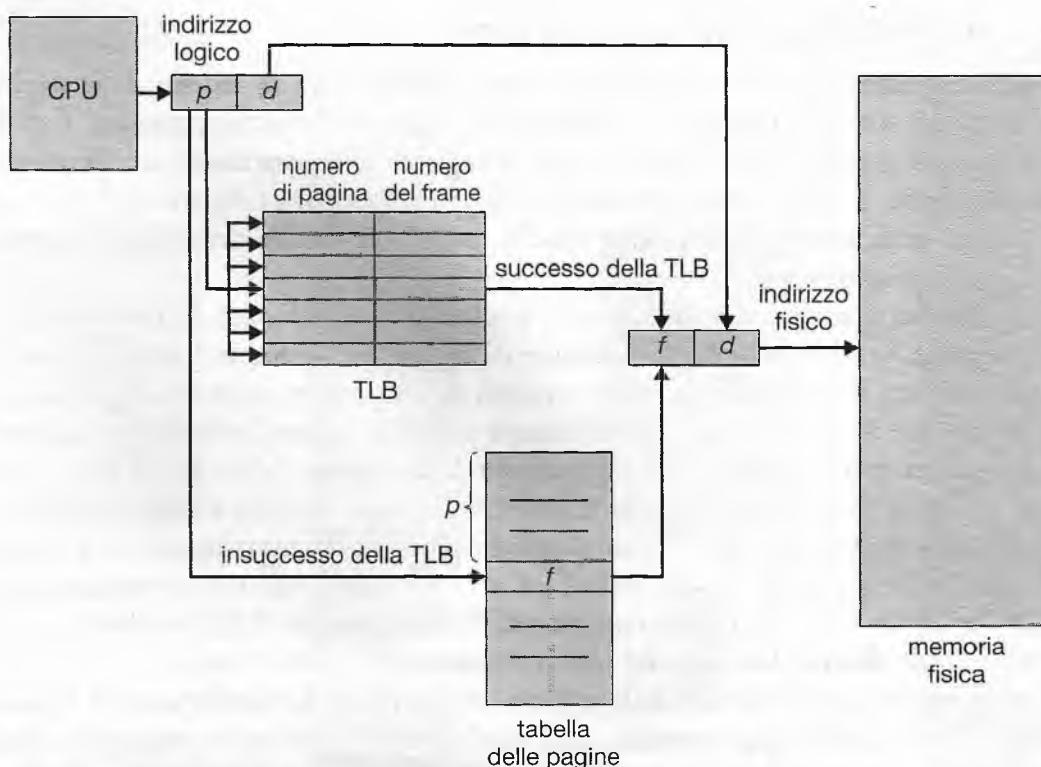
L'architettura d'ausilio alla tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice, si usa uno specifico insieme di registri. Per garantire un'efficiente traduzione degli indirizzi di paginazione, questi registri devono essere costruiti in modo da operare a una velocità molto elevata. Tale efficienza è determinante, poiché ogni accesso alla memoria passa attraverso il sistema di paginazione. Il dispatcher della CPU ricarica questi registri proprio come ricarica gli altri, ma le istruzioni di caricamento e modifica dei registri della tabella delle pagine sono privilegiate, quindi soltanto il sistema operativo può modificare la mappa della memoria. Il DEC PDP-11 è un esempio di tale tipo di architettura. Un indirizzo è costituito di 16 bit e la dimensione di una pagina è di 8 KB; la tabella delle pagine consiste quindi di otto elementi che sono mantenuti in registri veloci.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. La maggior parte dei calcolatori contemporanei usa comunque tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima si mantiene in memoria principale e un **registro di base della tabella delle pagine** (*page-table base register*, PTBR) punta alla tabella stessa. Il cambio delle tabelle delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto.

Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Per accedere alla locazione  $i$ , occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato del numero di pagina relativo a  $i$ , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato allo scostamento di pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo, per accedere a un byte occorrono *due* accessi alla memoria (uno per l'elemento della tabella delle pagine e uno per il byte stesso), quindi l'accesso alla memoria è rallentato di un fattore 2. Nella maggior parte dei casi un tale ritardo è intollerabile; sarebbe più conveniente ricorrere all'avvicendamento dei processi!

La soluzione tipica a questo problema consiste nell'impiego di una speciale, piccola cache di ricerca veloce, detta TLB (*translation look-aside buffer*). La TLB è una memoria associativa ad alta velocità in cui ogni suo elemento consiste di due parti: una chiave, o un indicatore (*tag*) e un valore. Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida, ma le memorie associative sono molto costose. Il numero degli elementi in una TLB è piccolo, spesso è compreso tra 64 e 1024.

La TLB si usa insieme con le tabelle delle pagine nel modo seguente: la TLB contiene una piccola parte degli elementi della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla TLB; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria. Tutta l'operazione può richiedere un tempo inferiore al 10 per cento in più di quanto sarebbe richiesto per un riferimento alla memoria senza paginazione.



**Figura 8.11** Architettura di paginazione con TLB.

Se nella TLB non è presente il numero di pagina, situazione nota come **insuccesso della TLB** (*TLB miss*), si deve consultare la tabella delle pagine in memoria. Il numero del frame così ottenuto si può eventualmente usare per accedere alla memoria (Figura 8.11). Inoltre, inserendo i numeri della pagina e del frame nella TLB, al riferimento successivo la ricerca sarà molto più rapida. Se la TLB è già piena d'elementi, il sistema operativo deve sceglierne uno per sostituirlo. I criteri di sostituzione variano dalla scelta dell'elemento usato meno recentemente (LRU) alla scelta casuale. Inoltre alcune TLB consentono che certi elementi siano **vincolati** (*wired down*), cioè non si possano rimuovere dalla TLB; in genere si vincolano gli elementi per il codice del kernel.

Alcune TLB memorizzano gli **identificatori dello spazio d'indirizzi** (*address-space identifier*, ASID) in ciascun elemento della TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, la TLB assicura che l'ASID per il processo attualmente in esecuzione corrisponda all'ASID associato alla pagina virtuale. La mancata corrispondenza dell'ASID viene trattata come un insuccesso della TLB. Inoltre, per fornire la protezione dello spazio d'indirizzi, l'ASID consente che la TLB contenga nello stesso istante elementi di diversi processi. Se la TLB non permette l'uso di ASID distinti, ogni volta che si seleziona una nuova tabella delle pagine, per esempio a ogni cambio di contesto, si deve **cancellare** (*flush*) la TLB, in modo da assicurare che il successivo processo in esecuzione non faccia uso di errate informazioni di traduzione. Potrebbero altrimenti esserci vecchi elementi della TLB contenenti indirizzi virtuali validi, ma con indirizzi fisici corrispondenti sbagliati o non validi, lasciati in sospeso dal precedente processo.

La percentuale di volte che un numero di pagina si trova nella TLB è detta **tasso di successi** (*hit ratio*). Un tasso di successi dell'80 per cento significa che il numero di pagina desiderato si trova nella TLB nell'80 per cento dei casi. Se la ricerca nella TLB richiede 20 nano-

secondi e sono necessari 100 nanosecondi per accedere alla memoria, allora, supponendo che il numero di pagina si trovi nella TLB, un accesso alla memoria richiede 120 nanosecondi. Se, invece, il numero non è contenuto nella TLB (20 nanosecondi), occorre accedere alla memoria per arrivare alla tabella delle pagine e al numero del frame (100 nanosecondi), quindi accedere al byte desiderato in memoria (100 nanosecondi); in totale sono necessari 220 nanosecondi. Per calcolare il tempo effettivo d'accesso alla memoria occorre tener conto della probabilità dei due casi:

$$\text{tempo effettivo d'accesso} = 0,80 \times 120 + 0,20 \times 220 = 140 \text{ nanosecondi}$$

In questo esempio si verifica un rallentamento del 40 per cento nel tempo d'accesso alla memoria (da 100 a 140 nanosecondi).

Per un tasso di successi del 98 per cento si ottiene il seguente risultato:

$$\text{tempo effettivo d'accesso} = 0,98 \times 120 + 0,02 \times 220 = 122 \text{ nanosecondi}$$

Aumentando il tasso di successi, il rallentamento del tempo d'accesso alla memoria scende al 22 per cento. L'impatto del tasso di successi nelle TLB è ulteriormente analizzato nel Capitolo 9.

### 8.4.3 Protezione

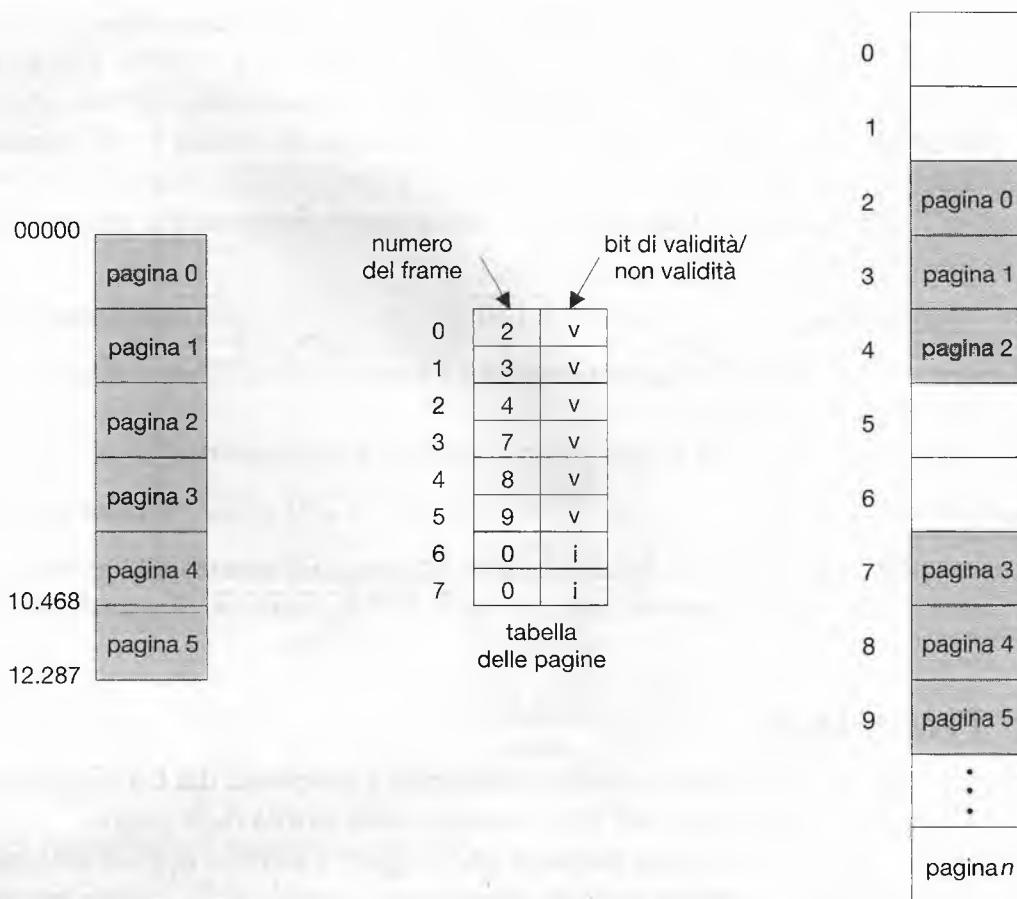
In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine.

Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero corretto del frame; quindi mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo causerebbe l'invio di un segnale di eccezione al sistema operativo, si avrebbe cioè una violazione della protezione della memoria.

Questo metodo si può facilmente estendere per fornire un livello di protezione più perfezionato. Si può progettare un'architettura che fornisca la protezione di sola lettura, di sola scrittura o di sola esecuzione. In alternativa, con bit di protezione distinti per ogni tipo d'accesso, si può ottenere una qualsiasi combinazione di tali tipi d'accesso; i tentativi illegali causano l'invio di un segnale di eccezione al sistema operativo.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un segnale di eccezione. Il sistema operativo concede o revoca la possibilità d'accesso a una pagina impostando in modo appropriato tale bit.

Per esempio, supponiamo che in un sistema con uno spazio di indirizzi di 14 bit (da 0 a 16.383) si possa avere un programma che deve usare soltanto gli indirizzi da 0 a 10.468. Con una dimensione delle pagine di 2 KB si ha la situazione mostrata nella Figura 8.12. Gli indirizzi nelle pagine 0, 1, 2, 3, 4 e 5 sono tradotti normalmente tramite la tabella delle pagine. D'altra parte, ogni tentativo di generare un indirizzo nelle pagine 6 o 7 trova non valido il bit di validità; in questo caso il calcolatore invia un segnale di eccezione al sistema operativo (riferimento di pagina non valido).



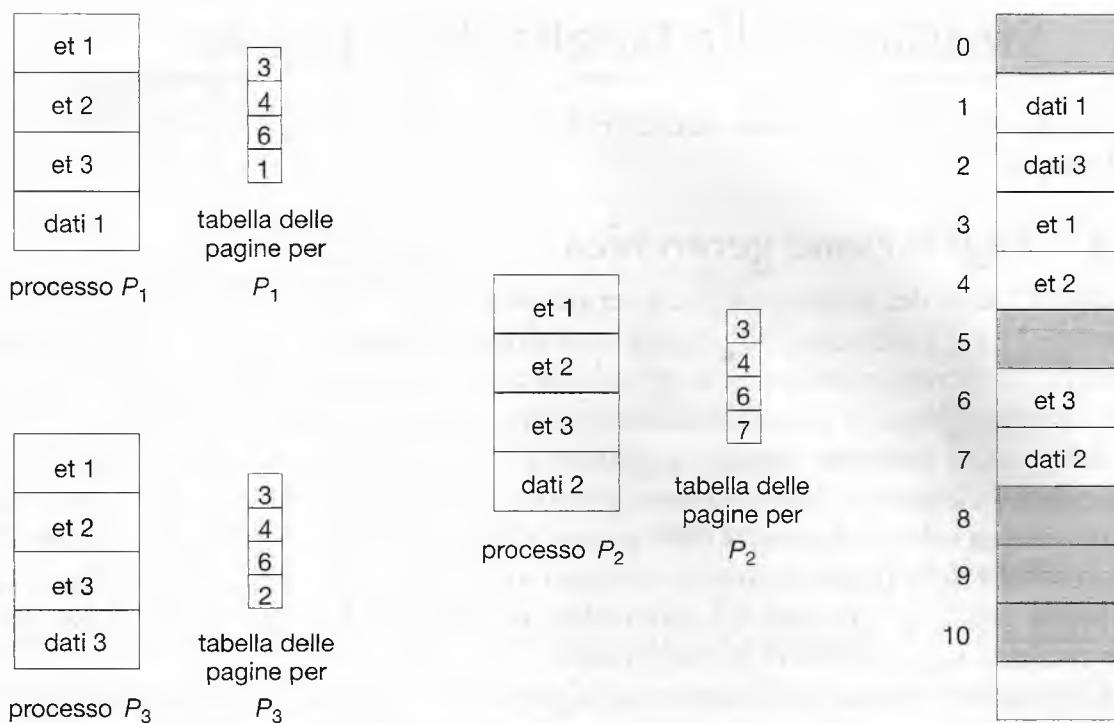
**Figura 8.12** Bit di validità (v) o non validità (i) in una tabella delle pagine.

Questo schema ha creato un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288 a 16.383 sono non validi. Tale problema è dovuto alla dimensione delle pagine di 2 KB e riflette la frammentazione interna della paginazione.

Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una gran parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria. Alcune architetture dispongono di registri, detti **registri di lunghezza della tabella delle pagine** (*page-table length register*, PTLR), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa l'invio di un segnale di eccezione al sistema operativo.

#### 8.4.4 Pagine condivise

Un altro vantaggio della paginazione consiste nella possibilità di *condividere* codice comune. Questa considerazione è importante soprattutto in un ambiente a partizione del tempo. Si consideri un sistema con 40 utenti, ciascuno dei quali usa un elaboratore di testi. Se tale programma è formato da 150 KB di codice e 50 KB di spazio di dati, per gestire i 40 utenti



**Figura 8.13** Condivisione di codice in un ambiente paginato.

sono necessari 8000 KB. Se però il codice è rientrante, può essere condiviso, come mostra la Figura 8.13: un elaboratore di testi, di tre pagine di 50 KB ciascuna (l'ampia dimensione delle pagine ha lo scopo di rendere più chiara la rappresentazione), condiviso da tre processi, ciascuno dei quali dispone della propria pagina di dati.

Il codice **rientrante**, detto anche **codice puro**, è un codice non automodificante: non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi variano, ovviamente, per ciascun processo.

In memoria fisica è presente una sola copia dell'elaboratore di testi: la tabella delle pagine di ogni utente fa corrispondere gli stessi frame contenenti l'elaboratore di testi, mentre le pagine dei dati si fanno corrispondere a frame diversi. Quindi per gestire 40 utenti sono sufficienti una copia dell'elaboratore di testi (150 KB) e 40 copie dei 50 KB di spazio di dati per ciascun utente; per un totale di 2150 KB, invece di 8000 KB; il risparmio è notevole.

Si possono condividere anche altri programmi d'uso frequente: compilatori, interfacce a finestre, librerie a collegamento dinamico, sistemi di basi di dati e così via. Per essere condivisibile, il codice deve essere rientrante. La natura di sola lettura del codice condiviso non si può affidare alla sola correttezza intrinseca del codice stesso, ma deve essere fatta rispettare dal sistema operativo. Tale condivisione della memoria tra processi di un sistema è simile al modo in cui i thread condividono lo spazio d'indirizzi di un task (Capitolo 4). Inoltre con riferimento al Capitolo 3, dove si descrive la memoria condivisa come un metodo di comunicazione tra processi, alcuni sistemi operativi realizzano la memoria condivisa impiegando le pagine condivise.

Oltre a permettere che più processi condividano le stesse pagine fisiche, l'organizzazione della memoria in pagine offre numerosi altri vantaggi; ne vedremo alcuni nel Capitolo 9.

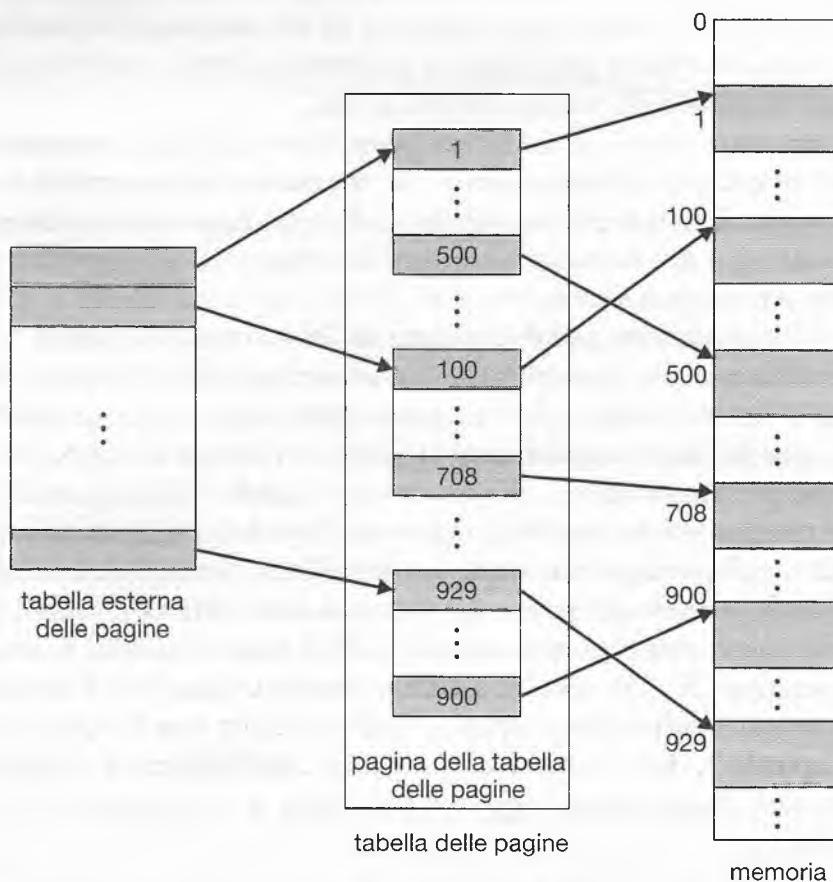
## 8.5 Struttura della tabella delle pagine

In questo paragrafo si descrivono alcune tra le tecniche più comuni per strutturare la tabella delle pagine.

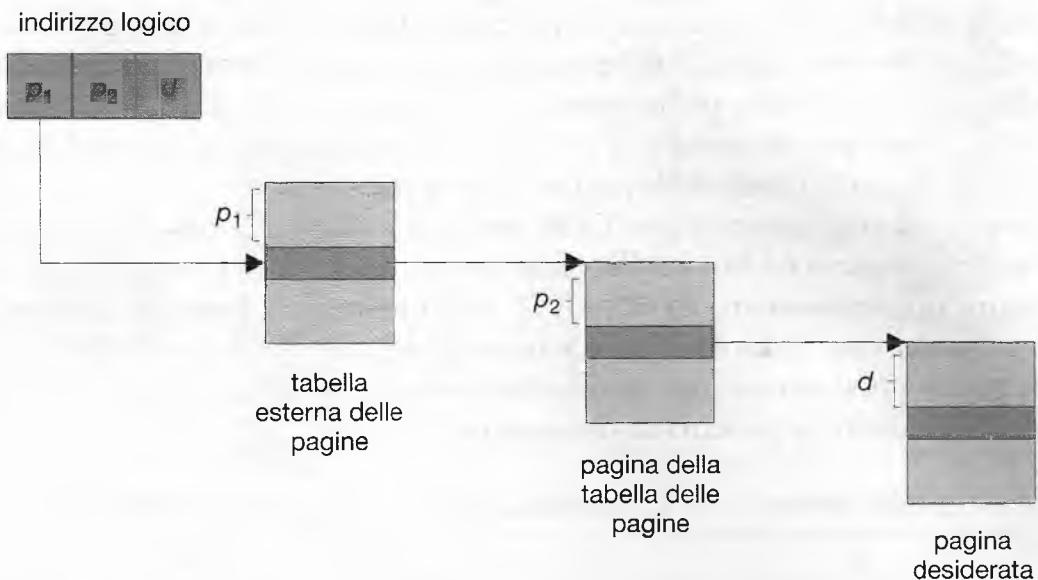
### 8.5.1 Paginazione gerarchica

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da  $2^{32}$  a  $2^{64}$  elementi). In un ambiente di questo tipo la stessa tabella delle pagine finirebbe per diventare eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d'indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 KB ( $2^{12}$ ), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi ( $2^{32}/2^{12}$ ). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 MB di spazio fisico d'indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (Figura 8.14). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Si suddivide ciascun indirizzo logico in un numero di pagina di 20 bit e in uno scostamento di pagina di 12 bit. Paginando la tabella delle



**Figura 8.14** Schema di una tabella delle pagine a due livelli.



**Figura 8.15** Traduzione degli indirizzi per un'architettura a 32 bit con paginazione a due livelli.

pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e uno scostamento di pagina di 10 bit. Quindi, l'indirizzo logico è:

numero di pagina	scostamento di pagina
$p_1$	$p_2$
10	10

12

dove  $p_1$  è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e  $p_2$  è lo scostamento all'interno della pagina indicata dalla tabella esterna delle pagine. Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella Figura 8.15. Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come **tabella delle pagine ad associazione diretta** (*forward-mapped page table*).

Anche l'architettura VAX ha una variante della paginazione a due livelli. Il VAX è una macchina a 32 bit con pagine di 512 byte. Lo spazio d'indirizzi logici di un processo è suddiviso in quattro sezioni uguali, ciascuna di  $2^{30}$  byte; ogni sezione rappresenta una parte differente dello spazio d'indirizzi logici di un processo. I 2 bit più significativi dell'indirizzo logico identificano la sezione appropriata, i successivi 21 bit rappresentano il numero logico di pagina all'interno di tale sezione e gli ultimi 9 bit lo scostamento nella pagina richiesta. Suddividendo in questo modo la tabella delle pagine, il sistema operativo può lasciare inutilizzate le diverse parti fino al momento in cui un processo ne fa richiesta. Nell'architettura VAX un indirizzo ha quindi la forma seguente:

sezione	pagina	scostamento
$s$	$p$	$d$

2                  21                  9

dove  $s$  denota il numero della sezione,  $p$  è un indice per la tabella delle pagine e  $d$  è lo scostamento all'interno della pagina. Anche quando si usa questo schema, la dimensione di una tabella delle pagine a un livello per un processo in un sistema VAX che usa una sezione è ancora  $2^{21}$  bit  $\times$  4 byte per elemento = 8 MB. Per ridurre ulteriormente l'uso della memoria centrale, il VAX pagina le tabelle delle pagine dei processi utenti.

Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per illustrare questo aspetto, si supponga che la dimensione delle pagine di questo sistema sia di 4 KB ( $2^{12}$ ). In questo caso, la tabella delle pagine conterrà fino a  $2^{52}$  elementi. Adottando uno schema di paginazione a due livelli, le tabelle interne delle pagine possono occupare convenientemente una pagina, o contenere  $2^{10}$  elementi di 4 byte. Gli indirizzi si presentano come segue:

pagina esterna	pagina interna	scostamento
$p_1$	$p_2$	$d$
42	10	12

La tabella esterna delle pagine consiste di  $2^{42}$  elementi, o  $2^{44}$  byte. La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcune CPU a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza.

La tabella esterna delle pagine si può suddividere in vari modi. Si può paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Si supponga che la tabella esterna delle pagine sia costituita di pagine di dimensione ordinaria ( $2^{10}$  elementi, o  $2^{12}$  byte); uno spazio d'indirizzi a 64 bit è ancora enorme:

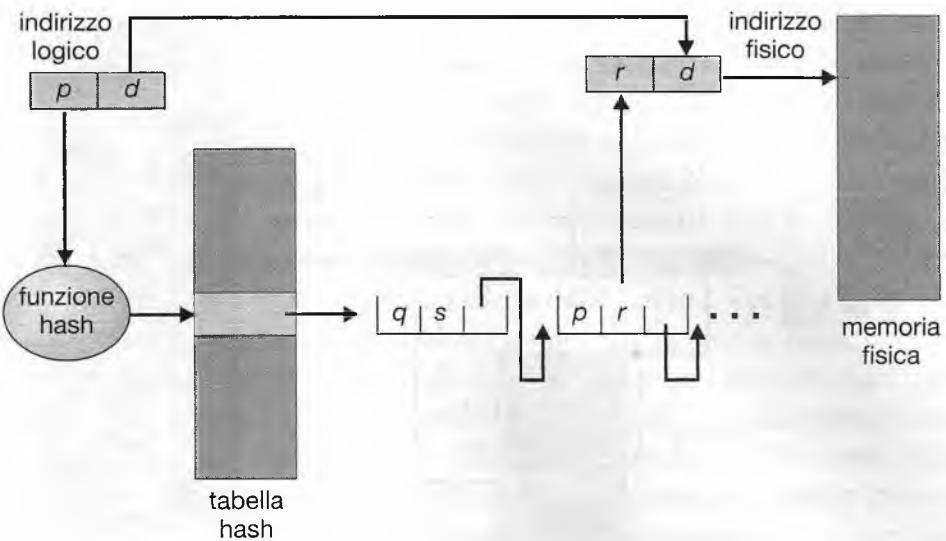
seconda pagina esterna	pagina esterna	pagina interna	scostamento
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

La tabella esterna delle pagine è ancora di  $2^{34}$  byte.

Il passo successivo sarebbe uno schema di paginazione a quattro livelli, in cui si pagina anche la tabella esterna di secondo livello delle pagine. L'UltraSPARC a 64 bit richiederebbe sette livelli di paginazione – con un numero proibitivo di accessi alla memoria – per tradurre ciascun indirizzo logico. Da questo esempio è possibile capire perché, per le architetture a 64 bit, le tabelle delle pagine gerarchiche sono in genere considerate inappropriate.

### 8.5.2 Tabella delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi relativi ad architetture oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**, in cui l'argomento della funzione hash è il numero della pagina virtuale. Per la gestione delle collisioni, ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi: (1) il nume-



**Figura 8.16** Tabella delle pagine di tipo hash.

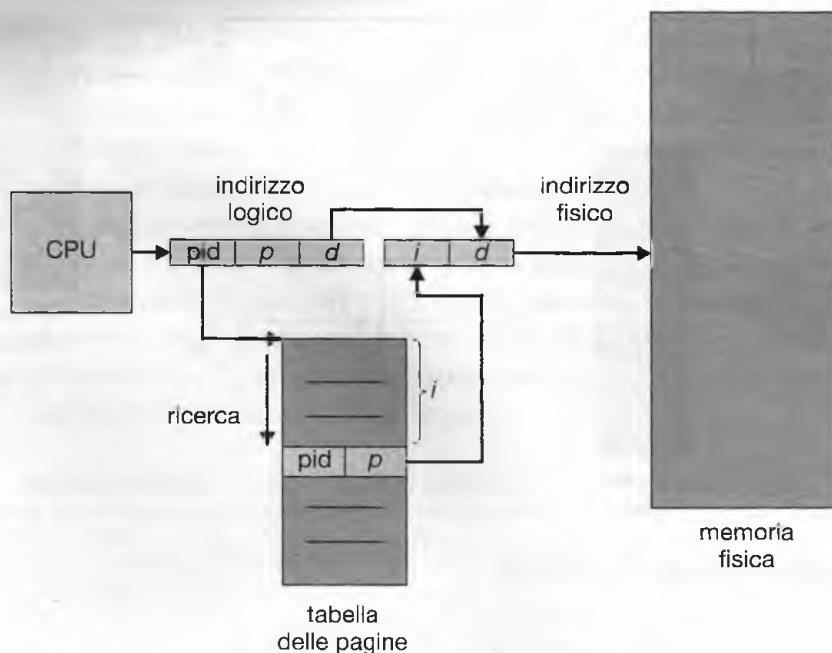
ro della pagina virtuale; (2) l'indirizzo del frame (pagina fisica) corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata (Figura 8.16). Le tabelle delle pagine di tipo hash sono particolarmente utili per gli spazi d'indirizzi sparsi, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi.

Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della **tabella delle pagine a gruppi** (*clustered page table*), simile alla tabella hash; ciascun elemento della tabella delle pagine contiene però i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue (per esempio 16). In questo modo si riduce lo spazio di memoria richiesto. Inoltre, poiché lo spazio degli indirizzi di molti programmi non è arbitrariamente sparso su pagine isolate ma distribuito per *raffiche* di riferimenti, le tabelle delle pagine a gruppi sfruttano bene questa caratteristica.

### 8.5.3 Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, oppure un elemento per ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa è una rappresentazione naturale della tabella, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'elemento dell'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è costituito dalla dimensione di ciascuna tabella delle pagine, che può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria proprio per sapere com'è impiegata la rimanente memoria fisica.



**Figura 8.17** Tabella delle pagine invertita.

Per risolvere questo problema si può fare uso della **tabella delle pagine invertita**. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell’indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica. Nella Figura 8.17 sono mostrate le operazioni di una tabella delle pagine invertita; si confronti questa figura con la Figura 8.7, che illustra il modo di operare per una tabella delle pagine ordinaria. Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio d’indirizzi (Paragrafo 8.4.2) in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi d’indirizzi diversi associati alla memoria fisica; l’identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente. Esempi di architetture che usano le tabelle delle pagine invertite sono l’UltraSPARC a 64 bit e la PowerPC.

Per illustrare questo metodo è possibile descrivere una versione semplificata della tabella delle pagine invertita dell’IBM RT. Ciascun indirizzo virtuale è una tripla del tipo seguente:

*<id-processo, numero di pagina, scostamento>*

Ogni elemento della tabella delle pagine invertita è una coppia *<id-processo, numero di pagina>* dove l’*id-processo* assume il ruolo di identificatore dello spazio d’indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell’indirizzo virtuale, formato da *<id-processo, numero di pagina>*, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull’elemento *i*, si genera l’indirizzo fisico *<i, scostamento>*. In caso contrario è stato tentato un accesso illegale a un indirizzo.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferi-

mento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza occorre esaminare tutta la tabella; questa ricerca richiede molto tempo. Per limitare l'entità del problema si può impiegare una tabella hash (come si descrive nel Paragrafo 8.5.2), che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Per migliorare le prestazioni, la ricerca si effettua prima nella TLB, quindi si consulta la tabella hash.

Nei sistemi che adottano le tabelle delle pagine invertite, l'implementazione della memoria condivisa è difficoltosa. Difatti, la condivisione si realizza solitamente tramite indirizzi virtuali multipli (uno per ogni processo che partecipa alla condivisione) associati a un unico indirizzo fisico. Il metodo è però inutilizzabile in presenza di tabelle invertite, perché, essendovi un solo elemento indicante la pagina virtuale corrispondente a ogni pagina fisica, questa non può avere più di un indirizzo virtuale associato. Una semplice tecnica per superare il problema consiste nel porre nella tabella delle pagine una sola associazione fra un indirizzo virtuale e l'indirizzo fisico condiviso; ciò comporta un errore dovuto all'assenza della pagina per ogni riferimento agli indirizzi virtuali non associati (*page fault*).

## 8.6 Segmentazione

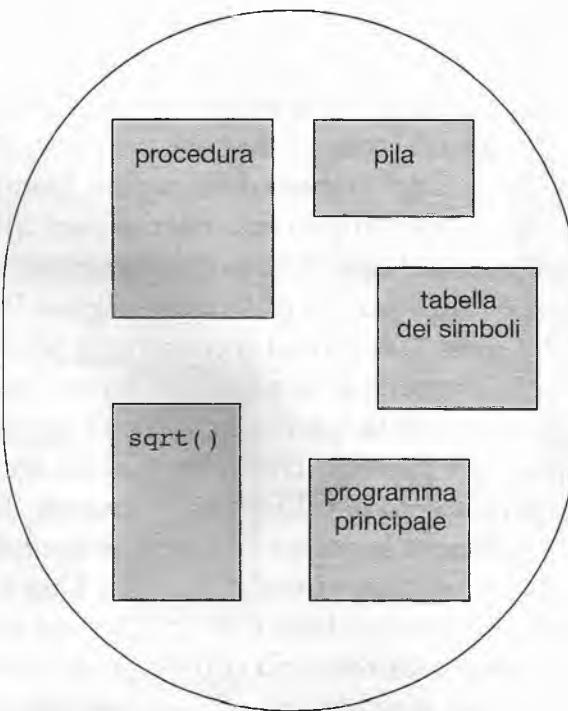
Un aspetto importante della gestione della memoria, inevitabile alla presenza della paginazione, è quello della separazione tra la visione della memoria dell'utente e l'effettiva memoria fisica. Lo spazio d'indirizzi *visto* dall'utente non coincide con l'effettiva memoria fisica, ma lo si fa corrispondere alla memoria fisica. I metodi che stabiliscono questa corrispondenza consentono di separare la memoria logica dalla memoria fisica.

### 8.6.1 Metodo di base

Ci si potrebbe chiedere se l'utente può considerare la memoria come un vettore lineare di byte, alcuni dei quali contengono istruzioni e altri dati. Molti risponderebbero di no. Gli utenti la vedono piuttosto come un insieme di segmenti di dimensione variabile non necessariamente ordinati (Figura 8.18).

La tipica struttura di un programma con cui i programmatori hanno familiarità è costituita di una parte principale e di un gruppo di procedure, funzioni o moduli, insieme con diverse strutture dati come tabelle, matrici, pile, variabili e così via. Ciascuno di questi moduli o elementi di dati si identifica con un nome: "tabella dei simboli", "funzione `sqrt()`", "programma principale", indipendentemente dagli indirizzi che questi elementi occupano in memoria. Non è necessario preoccuparsi del fatto che la tabella dei simboli sia memorizzata prima o dopo la funzione `sqrt()`. Ciascuno di questi segmenti ha una lunghezza variabile, definita intrinsecamente dallo scopo che il segmento stesso ha all'interno del programma. Gli elementi che si trovano all'interno di un segmento sono identificati dal loro scostamento, misurato dall'inizio del segmento: la prima istruzione del programma, il settimo elemento della tabella dei simboli, la quinta istruzione della funzione `sqrt()`, e così via.

La **segmentazione** è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente. Uno spazio d'indirizzi



**Figura 8.18** Un programma dal punto di vista dell'utente.

logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia lo scostamento all'interno del segmento, quindi l'utente fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e uno scostamento. Questo schema contrasta con la paginazione, in cui l'utente fornisce un indirizzo singolo, che l'architettura di paginazione suddivide in un numero di pagina e uno scostamento, non visibili dal programmatore.

Per semplicità i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una *coppia*

*<numero di segmento, scostamento>*

Normalmente il programma utente è stato compilato, e il compilatore struttura automaticamente i segmenti secondo il programma sorgente. Un compilatore per il linguaggio C può creare segmenti distinti per i seguenti elementi di un programma:

1. il codice;
2. le variabili globali;
3. lo heap, da cui si alloca la memoria;
4. le pile usate da ciascun thread;
5. la libreria standard del C.

Alle librerie collegate dal linker al momento della compilazione possono essere assegnati dei nuovi segmenti. Il caricatore preleva questi segmenti e assegna loro i numeri di segmento.

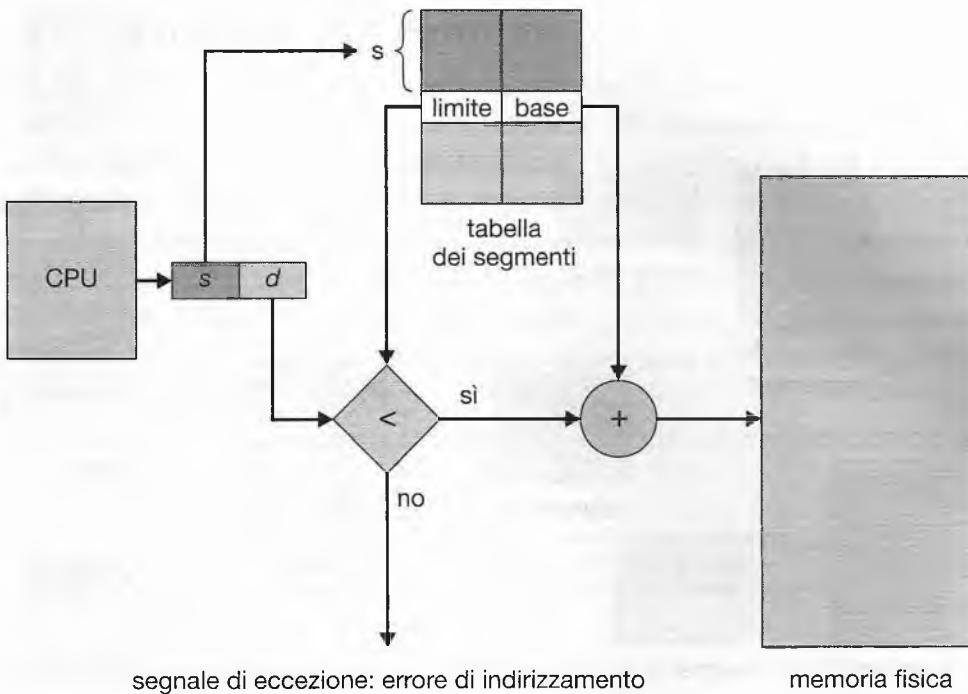


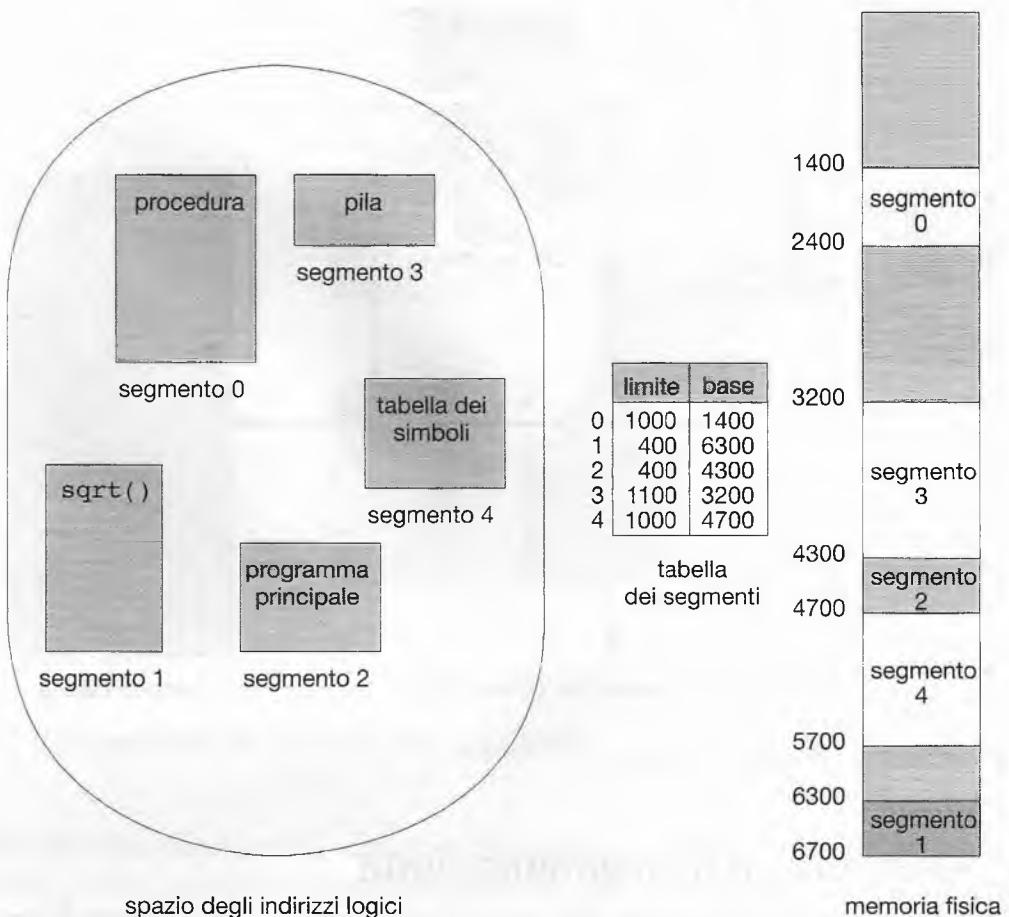
Figura 8.19 Architettura di segmentazione.

## 8.6.2 Architettura di segmentazione

Sebbene l'utente possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. Per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una **tabella dei segmenti**; ogni suo elemento è una coppia ordinata: la *base del segmento* e il *limite del segmento*. La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

L'uso della tabella dei segmenti è illustrato nella Figura 8.19. Un indirizzo logico è formato da due parti: un numero di segmento *s* e uno scostamento in tale segmento *d*. Il numero di segmento si usa come indice per la tabella dei segmenti; lo scostamento *d* dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti s'invia un segnale di eccezione al sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento). Se tale condizione è rispettata, si somma lo scostamento alla base del segmento per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è fondamentalmente un vettore di coppie di registri di base e limite.

Come esempio si può considerare la situazione illustrata nella Figura 8.20. Sono dati cinque segmenti numerati da 0 a 4, memorizzati in memoria fisica. La tabella dei segmenti ha un elemento distinto per ogni segmento, indicante l'indirizzo iniziale del segmento in memoria fisica (la base) e la lunghezza di quel segmento (il limite). Per esempio, il segmento 2 è lungo 400 byte e inizia alla locazione 4300, quindi un riferimento al byte 53 del segmento 2 si fa corrispondere alla locazione  $4300 + 53 = 4353$ . Un riferimento al segmento 3, byte 852, si fa corrispondere alla locazione  $3200$  (la base del segmento 3) +  $852 = 4052$ . Un riferimento al byte 1222 del segmento 0 causa l'invio di un segnale di eccezione al sistema operativo, poiché questo segmento è lungo 1000 byte.



**Figura 8.20** Esempio di segmentazione.

## 8.7 Un esempio: Pentium Intel

Paginazione e segmentazione presentano vantaggi e svantaggi, tanto che alcune architetture dispongono di entrambe le tecniche. In questo paragrafo prendiamo in esame l'architettura del Pentium Intel che utilizza, oltre alla segmentazione pura, la segmentazione mista a paginazione. Non ci inoltreremo in una trattazione completa della gestione della memoria del Pentium, ma ci limiteremo a illustrare i concetti fondamentali su cui è basata. La nostra analisi si concluderà con una panoramica della traduzione degli indirizzi di Linux nei sistemi Pentium.

In questi sistemi la CPU genera indirizzi logici, che confluiscono nell'unità di segmentazione. Questa produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare passa quindi all'unità di paginazione, la quale, a sua volta, genera l'indirizzo fisico all'interno della memoria centrale. Così, le unità di segmentazione e di paginazione formano un equivalente dell'unità di gestione della memoria (MMU). Il modello è rappresentato nella Figura 8.21.



**Figura 8.21** Traduzione degli indirizzi logici in indirizzi fisici in Pentium.

### 8.7.1 Segmentazione in Pentium

Nell'architettura Pentium un segmento può raggiungere la dimensione massima di 4 GB; il numero massimo di segmenti per processo è pari a 16 KB. Lo spazio degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8 KB segmenti riservati al processo; la seconda contiene fino a 8 KB segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella **tabella locale dei descrittori** (*local descriptor table*, LDT), quelle relative alla seconda partizione sono memorizzate nella **tabella globale dei descrittori** (*global descriptor table*, GDT). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite.

Un indirizzo logico è una coppia (*selettore, scostamento*), dove il selettore è un numero di 16 bit:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

in cui *s* indica il numero del segmento, *g* indica se il segmento si trova nella GDT o nella LDT e *p* contiene informazioni relative alla protezione. Lo scostamento è un numero di 32 bit che indica la posizione del byte (o della parola) all'interno del segmento in questione.

La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita a Pentium di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

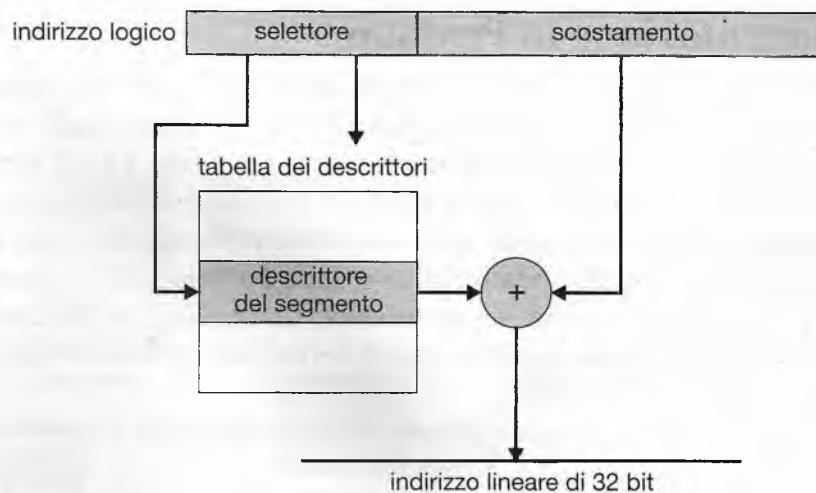
Un indirizzo lineare di Pentium è lungo 32 bit e si genera come segue. Il registro di segmento punta all'elemento appropriato all'interno della LDT o della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare un **indirizzo lineare**. Innanzitutto si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa l'invio di un segnale di eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dello scostamento al valore della base, ottenendo un indirizzo lineare di 32 bit. La Figura 8.22 illustra tale processo. Nel paragrafo successivo si considera come l'unità di paginazione trasforma questo indirizzo lineare in un indirizzo fisico.

### 8.7.2 Paginazione in Pentium

L'architettura Pentium prevede che le pagine abbiano una misura di 4 KB oppure di 4 MB. Per le prime, in Pentium vige uno schema di paginazione a due livelli che prevede la seguente scomposizione degli indirizzi lineari a 32 bit:

numero di pagina		scostamento di pagina
<i>p</i> <sub>1</sub>	<i>p</i> <sub>2</sub>	<i>d</i>
10	10	12

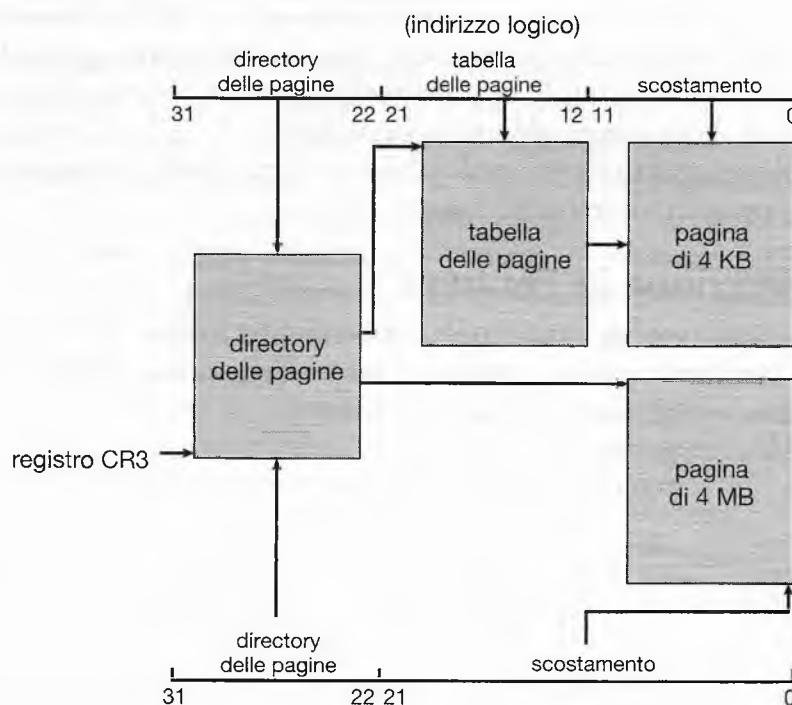
Lo schema di traduzione degli indirizzi per questa architettura, simile a quello rappresentato nella Figura 8.15, è mostrato in dettaglio nella Figura 8.23. I dieci bit più significativi puntano a un elemento nella tabella delle pagine più esterna, detta in Pentium **directory**



**Figura 8.22** Segmentazione in Pentium Intel.

delle pagine. (Il registro CR3 punta alla directory delle pagine del processo corrente.) Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono lo scostamento da applicare all'interno della pagina di 4 KB cui si fa riferimento nella tabella delle pagine.

Un elemento appartenente alla directory delle pagine è il flag Page Size; se impostato, indica che il frame non ha la dimensione standard di 4 MB, ma misura invece 4 KB. In questo caso, la directory di pagina punta direttamente al frame di 4 MB, scavalcando la tabella delle pagine interna; i 22 bit meno significativi nell'indirizzo lineare indicano lo scostamento nella pagina di 4 MB.



**Figura 8.23** Paginazione nell'architettura Pentium.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine del Pentium Intel possono essere trasferite sul disco. In questo caso, si ricorre a un bit in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Nel secondo caso, il sistema operativo può usare i 31 bit rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

### 8.7.3 Linux su sistemi Pentium

Si consideri, a scopo di esempio, il sistema operativo Linux montato sull'architettura Pentium Intel. Linux può essere eseguito dai processori più disparati, molti dei quali scarsamente predisposti alla segmentazione. Per questo motivo Linux non attribuisce un ruolo rilevante alla segmentazione, di cui fa un uso minimo. Su Pentium, Linux utilizza soltanto sei segmenti:

1. un segmento per il codice del kernel;
2. un segmento per i dati del kernel;
3. un segmento per il codice dell'utente;
4. un segmento per i dati dell'utente;
5. un segmento per lo stato del task (*task-state segment*, TSS);
6. un segmento di default per la tabella locale dei descrittori (LDT).

I segmenti per il codice e i dati dell'utente sono condivisi da tutti i processi eseguiti in modalità utente. Ciò è possibile perché tutti i processi usano il medesimo spazio degli indirizzi logici, e tutti i descrittori dei segmenti risiedono nella tabella globale dei descrittori (*global descriptor table*, GDT). Il TSS serve a memorizzare il contesto hardware dei processi durante il cambio di contesto. Il segmento LDT di default è solitamente condiviso da tutti i processi, ma inutilizzato; i processi che lo desiderino, tuttavia, possono creare e usare il proprio LDT privato.

Come s'è detto, i selettori dei segmenti comprendono un campo di due bit dedicato alla protezione. Pentium contempla pertanto 4 livelli di protezione, dei quali solo due sfruttati da Linux: modalità utente e kernel.

Sebbene Pentium sia dotato di uno schema di paginazione a due livelli, Linux è compatibile con un'intera gamma di piattaforme hardware, molte delle quali, essendo a 64 bit, rendono i due livelli poco consigliabili. Linux adotta pertanto una strategia di paginazione a tre livelli che funziona bene sia sulle architetture a 64 bit sia su quelle a 32 bit.

Un indirizzo lineare di Linux è composta da quattro parti:

directory globale	directory intermedia	tabella delle pagine	scostamento
-------------------	----------------------	----------------------	-------------

La Figura 8.24 illustra il modello di paginazione a tre livelli di Linux.

Il numero di bit delle quattro parti dell'indirizzo lineare dipende dall'architettura. Tuttavia, come s'è osservato poc'anzi, Pentium prevede solo due livelli di paginazione. È naturale dunque chiedersi come Linux applichi il proprio schema a tre livelli a Pentium. La risposta è che, in questo caso, la lunghezza intermedia del campo directory è di zero bit, ciò che porta a ignorare la directory intermedia.

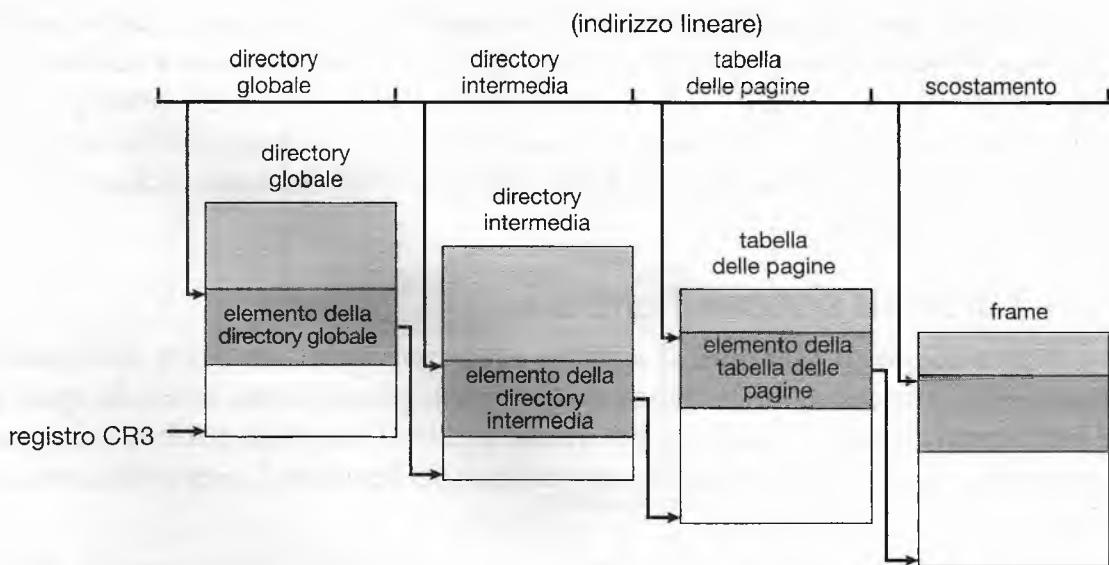


Figura 8.24 Paginazione a tre livelli di Linux.

Ogni task di Linux possiede il proprio insieme di tabelle delle pagine, come illustrato dalla Figura 8.23; il registro CR3 punta alla directory globale del task correntemente in esecuzione. Durante il cambio di contesto, il valore di CR3 è salvato nei segmenti TSS dei task coinvolti dal cambio.

## 8.8 Sommario

Gli algoritmi di gestione della memoria per sistemi operativi multiprogrammati variano da un metodo semplice, per sistema con singolo utente, fino alla segmentazione paginata. Il fattore più rilevante che incide maggiormente sulla scelta del metodo da seguire in un sistema particolare è costituito dall'architettura disponibile. È necessario controllare la validità di ogni indirizzo di memoria generato dalla CPU; inoltre tali indirizzi, se sono corretti, devono essere tradotti in un indirizzo fisico. Tali controlli per essere efficienti devono essere effettuati dall'architettura del sistema, che secondo le sue caratteristiche pone vincoli al sistema operativo.

Gli algoritmi di gestione della memoria analizzati (allocazione contigua, paginazione, segmentazione e combinazione di paginazione e segmentazione) differiscono per molti aspetti. Per confrontare i diversi metodi di gestione della memoria si possono considerare i seguenti elementi.

- ◆ **Architettura.** Un semplice registro di base oppure una coppia di registri di base e limite è sufficiente per i metodi con partizione singola e con più partizioni, mentre la paginazione e la segmentazione necessitano di tabelle di traduzione per definire la corrispondenza degli indirizzi.
- ◆ **Prestazioni.** Aumentando la complessità dell'algoritmo, aumenta anche il tempo necessario per tradurre un indirizzo logico in un indirizzo fisico. Nei sistemi più semplici è sufficiente fare un confronto o sommare un valore all'indirizzo logico; si tratta di operazioni rapide. Paginazione e segmentazione possono essere altrettanto rapide se

per realizzare la tabella s'impiegano registri veloci. Se però la tabella si trova in memoria, gli accessi alla memoria utente possono essere assai più lenti. Una TLB può limitare il calo delle prestazioni a un livello accettabile.

- ◆ **Frammentazione.** Un sistema multiprogrammato esegue le elaborazioni generalmente in modo più efficiente se ha un più elevato livello di multiprogrammazione. Per un dato gruppo di processi, il livello di multiprogrammazione si può aumentare solo compattando più processi in memoria. Per eseguire questo compito occorre ridurre lo spreco di memoria o la frammentazione. Sistemi con unità di allocazione di dimensione fissa, come lo schema con partizione singola e la paginazione, soffrono di frammentazione interna. Sistemi con unità di allocazione di dimensione variabile, come lo schema con più partizioni e la segmentazione, soffrono di frammentazione esterna.
- ◆ **Rilocazione.** Una soluzione al problema della frammentazione esterna è data dalla compattazione, che implica lo spostamento di un programma in memoria, senza che il programma stesso *si accorga* del cambiamento. Ciò richiede che gli indirizzi logici siano rilocati dinamicamente al momento dell'esecuzione. Se gli indirizzi si rilocano solo al momento del caricamento, non è possibile compattare la memoria.
- ◆ **Avvicendamento dei processi.** L'avvicendamento dei processi (*swapping*) si può incorporare in ogni algoritmo. I processi si copiano dalla memoria centrale alla memoria ausiliaria, e successivamente si ricopiano in memoria centrale a intervalli fissati dal sistema operativo, e generalmente stabiliti dai criteri di scheduling della CPU. Questo schema permette di inserire contemporaneamente in memoria più processi da eseguire.
- ◆ **Condivisione.** Un altro mezzo per aumentare il livello di multiprogrammazione è quello della condivisione del codice e dei dati tra diversi utenti. Poiché deve fornire piccoli pacchetti d'informazioni (pagine o segmenti) condivisibili, generalmente richiede l'uso della paginazione o della segmentazione. La condivisione permette di eseguire molti processi con una quantità di memoria limitata, ma i programmi e i dati condivisi si devono progettare con estrema cura.
- ◆ **Protezione.** Con la paginazione o la segmentazione, diverse sezioni di un programma utente si possono dichiarare di sola esecuzione, di sola lettura oppure di lettura e scrittura. Questa limitazione è necessaria per il codice e i dati condivisi, ed è utile, in genere, nei casi in cui sono richiesti semplici controlli nella fase d'esecuzione per l'individuazione degli errori di programmazione.

## Esercizi pratici

- 8.1 Citate due differenze tra indirizzi logici e fisici.
- 8.2 Considerate un sistema nel quale un programma possa essere separato in due parti: codice e dati. Il processore sa se necessita di un'istruzione (prelievo di istruzione) o dati (prelievo o memorizzazione di dati). Perciò, vengono fornite due coppie di registri base e limite: una per le istruzioni e una per i dati. La coppia di registri base e limite per le istruzioni è automaticamente a sola lettura, di modo che i programmi possano essere condivisi tra i diversi utenti. Discutete i vantaggi e gli svantaggi di questo schema.

- 8.3 Perché la dimensione delle pagine è sempre una potenza di due?
- 8.4 Considerate uno spazio degli indirizzi logici di 64 pagine, ciascuna delle quali di 1024 parole, rilocate in una memoria fisica di 32 frame.
- Quanti bit ci sono nell'indirizzo logico?
  - Quanti bit ci sono nell'indirizzo fisico?
- 8.5 Quale effetto si verifica se si permette a due voci di una tabella delle pagine di puntare allo stesso frame di pagina della memoria? Spiegate come questo effetto potrebbe essere utilizzato per diminuire il tempo necessario per copiare gran parte della memoria da uno spazio a un altro. Nel caso in cui vengano aggiornati alcuni byte della prima pagina, quale effetto si avrebbe sulla seconda pagina?
- 8.6 Descrivete un meccanismo per il quale un segmento potrebbe appartenere allo spazio degli indirizzi di due processi differenti.
- 8.7 In un sistema di segmentazione dinamicamente collegato è possibile condividere segmenti tra processi senza richiedere che abbiano lo stesso numero di segmento.
- Definite un sistema che permetta la connessione statica e la condivisione di segmenti senza richiedere che il numero del segmento sia lo stesso.
  - Descrivete uno schema di paginazione che permetta alle pagine di essere condivise senza richiedere ai numeri delle pagine di essere gli stessi.
- 8.8 Nell'IBM/370 la memoria viene protetta attraverso l'uso di *chiavi*. Una chiave è di 4 bit. Ogni blocco di memoria di 2 K è associato a una chiave (la chiave di memoria). Anche il processore è associato a una chiave (la chiave di protezione). Un'operazione di memorizzazione è permessa solo se entrambe le chiavi sono uguali oppure se una è uguale zero. Quale dei seguenti schemi di gestione della memoria potrebbe essere usato con successo con questo hardware?
- Macchina nuda.
  - Sistema a singolo utente.
  - Programmazione multipla con un numero fisso di processi.
  - Programmazione multipla con un numero variabile di processi.
  - Paginazione.
  - Segmentazione.

## Esercizi

- 8.9 Spiegate la differenza tra frammentazione interna e frammentazione esterna.
- 8.10 Considerate il seguente ciclo di produzione di codice binario eseguibile. Si usa un compilatore per generare il codice oggetto dei singoli moduli, e un editor per gestire la fase di link, ossia per combinare diversi moduli oggetto in un unico codice binario eseguibile. Come può l'editor modificare l'associazione tra istruzioni e dati, e gli indirizzi di memoria? Quali informazioni devono passare dal compilatore all'editor per facilitare l'editor nell'esecuzione di tale associazione?

- 8.11 Date cinque partizioni di memoria, pari a 100 KB, 500 KB, 200 KB, 300 KB e 600 KB, nell'ordine, e cinque processi di 212 KB, 417 KB, 112 KB e 426 KB, nell'ordine, come verrebbero allocati in memoria tali processi dagli algoritmi first-fit, best-fit e worst-fit? Quale algoritmo sfrutta più efficientemente la memoria?
- 8.12 La maggioranza dei sistemi consente ai programmi di aumentare durante l'esecuzione la memoria allocata al proprio spazio di indirizzi. (I dati posti nei segmenti heap dei programmi ne rappresentano un esempio.) Di che cosa c'è bisogno per agevolare l'allocazione dinamica della memoria, in ognuno dei seguenti casi?
- Allocazione contigua della memoria.
  - Segmentazione pura.
  - Paginazione pura.
- 8.13 Confrontate i modelli della segmentazione pura, paginazione pura e allocazione contigua in riferimento alle seguenti tematiche:
- frammentazione esterna;
  - frammentazione interna;
  - capacità di condividere codice tra i processi.
- 8.14 Perché i sistemi che si avvalgono della paginazione vietano ai processi di accedere alla memoria che non possiedono? In che modo potrebbe il sistema operativo concedere l'accesso a tale memoria estranea? Argomentate perché dovrebbe o non dovrebbe farlo.
- 8.15 Confrontate la paginazione con la segmentazione, in riferimento alla quantità di memoria richiesta dalle strutture di traduzione degli indirizzi al fine di convertire gli indirizzi virtuali in indirizzi fisici.
- 8.16 Il codice binario eseguibile di un programma, in molti sistemi, ha la seguente forma tipica. Il codice è memorizzato a partire da un piccolo indirizzo virtuale fisso, come, per esempio, 0. Al segmento del codice fa seguito il segmento dei dati, utilizzato per memorizzare le variabili del programma. Quando il programma dà avvio all'esecuzione, la pila è collocata all'altro estremo dello spazio degli indirizzi virtuali, e ha, dunque, la possibilità di espandersi verso gli indirizzi virtuali inferiori. Quale rilevanza assume la struttura ora descritta nelle seguenti circostanze?
- Allocazione contigua della memoria.
  - Segmentazione pura.
  - Paginazione pura.
- 8.17 Assumendo che la dimensione della pagina sia di 1 KB, quali sono i numeri di pagina e gli scostamenti per i seguenti indirizzi (indicati in numeri decimali):
- 2375
  - 19366
  - 30000
  - 256
  - 16385

- 8.18 Considerate uno spazio di indirizzo logico di 32 pagine con 1024 parole per pagina, rilocate in una memoria fisica di 16 frame.
- Quanti bit sono necessari all'indirizzo logico?
  - Quanti bit sono necessari all'indirizzo fisico?
- 8.19 Prendete in considerazione un sistema con un indirizzo logico di 32 bit e una dimensione di pagina di 4 KB. Il sistema supporta fino a 512 MB di memoria fisica. Quante voci ci sono in:
- una tabella delle pagine convenzionale a singolo livello;
  - una tabella delle pagine invertita.
- 8.20 Considerate un sistema di paginazione con la tabella delle pagine conservata in memoria.
- Se un riferimento alla memoria necessita di 200 nanosecondi per essere servito, di quanto necessiterà un riferimento alla memoria paginata?
  - Se si aggiungono TLB, e il 75 percento di tutti i riferimenti si trova in questi ultimi, quale sarà il tempo effettivo di riferimento alla memoria? (Ipotizzate che la ricerca di un elemento effettivamente presente nelle TLB richieda tempo zero.)
- 8.21 Spiegate perché segmentazione e paginazione si combinino talvolta in un unico schema.
- 8.22 Spiegate perché sia più facile condividere un modulo di codice rientrante usando la segmentazione anziché la paginazione pura.
- 8.23 Considerate la seguente tabella dei segmenti:

Segmento	Base	Lunghezza
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Calcolate gli indirizzi fisici corrispondenti ai seguenti indirizzi logici:

- $<0, 430>$
  - $<1, 10>$
  - $<2, 500>$
  - $<3, 400>$
  - $<4, 112>$
- 8.24 Qual è lo scopo di paginare le tabelle delle pagine?
- 8.25 Considerate il meccanismo gerarchico di paginazione impiegato dall'architettura VAX. Quante operazioni di memoria si effettuano quando un programma utente esegue un'operazione di caricamento in memoria?

- 8.26 Confrontate il meccanismo di paginazione segmentata con il modello delle tabelle hash delle pagine, adatto a gestire spazi degli indirizzi grandi. In quali circostanze è preferibile optare per l'uno o per l'altro?
- 8.27 Considerate lo schema di traduzione degli indirizzi di Intel Pentium mostrato nella Figura 8.22.
- Descrivete tutti i passi eseguiti da Intel Pentium nel tradurre un indirizzo logico in un indirizzo fisico.
  - Esponete i vantaggi offerti a un sistema operativo da un'architettura dotata di un così complesso sistema di traduzione degli indirizzi.
  - Dite se ci sono svantaggi in questo sistema di traduzione degli indirizzi; se sì, dite quali sono; altrimenti spiegate perché non è impiegato da ogni costruttore.

## Problemi di programmazione

- 8.28 Assumete che un sistema abbia un indirizzo virtuale di 32 bit con una dimensione della pagina di 4 KB. Scrivete un programma in C al quale viene passato un indirizzo virtuale (in decimale) dalla riga di comando e che fornisce in output il numero di pagina e gli scostamenti per l'indirizzo dato. Ad esempio, se il programma fosse invocato come segue:

```
./a.out 19986
```

il risultato sarebbe:

```
The address 19986 contains:  
page number = 4  
offset = 3602
```

Scrivere questo programma richiederà di utilizzare tipi di dati appropriati per memorizzare 32 bit. Suggeriamo inoltre di utilizzare tipi di dati `unsigned`.

## 8.9 Note bibliografiche

L'allocazione dinamica della memoria è analizzata nel Paragrafo 2.5 di [Knuth 1973], il quale, tramite i risultati di simulazioni, scopri che il criterio di scelta del primo buco abbastanza grande (*first-fit*) è in genere più vantaggioso del criterio di scelta del più piccolo tra i buchi abbastanza grandi (*best-fit*). Nella stessa opera, Knuth discute la regola del 50 per cento.

Il concetto di paginazione si può attribuire ai progettisti del sistema Atlas, descritto sia da [Kilburn et al. 1961] che da [Howarth et al. 1961]. Il concetto di segmentazione è stato discusso per la prima volta da [Dennis 1965]. Il primo sistema con segmentazione paginata è stato il ge 645, per il quale era stato originariamente progettato e realizzato il sistema operativo multics ([Organick 1972] e [Daley e Dennis 1967]).

Le tabelle delle pagine invertite sono trattate in un articolo sulla gestione della memoria dell'IBM RT, di [Chang e Mergen 1988].

[Jacob e Mudge 1997] prendono in esame la traduzione software degli indirizzi.

[Hennessy e Patterson 2002] si occupano degli aspetti architettonici di TLB, cache e MMU. Le tabelle delle pagine per spazi di indirizzi a 64 bit sono trattate in [Talluri et al. 1995]. Soluzioni alternative volte a rafforzare la protezione della memoria sono state elaborate e proposte da [Wahbe et al.

## Capitolo 9

# Memoria virtuale



### OBIETTIVI

- Descrizione dei vantaggi derivanti dalla memoria virtuale.
- Definizione dei concetti di paginazione su richiesta, algoritmi di sostituzione di pagina e allocazione dei frame.

Nel Capitolo 8 sono state esaminate le strategie di gestione della memoria impiegate nei calcolatori. Hanno tutte lo stesso scopo: tenere contemporaneamente più processi in memoria per permettere la multiprogrammazione; tuttavia esse tendono a richiedere che l'intero processo si trovi in memoria prima di essere eseguito.

La **memoria virtuale** è una tecnica che permette di eseguire processi che possono anche non essere completamente contenuti in memoria. Il vantaggio principale offerto da questa tecnica è quello di permettere che i programmi siano più grandi della memoria fisica; inoltre la memoria virtuale astrae la memoria centrale in un vettore di memorizzazione molto grande e uniforme, separando la memoria logica, com'è vista dall'utente, da quella fisica. Questa tecnica libera i programmatori da quel che riguarda i limiti della memoria. La memoria virtuale permette inoltre ai processi di condividere facilmente file e spazi d'indirizzi, e fornisce un meccanismo efficiente per la creazione dei processi. La memoria virtuale è però difficile da realizzare e, s'è usata scorrettamente, può ridurre di molto le prestazioni del sistema. In questo capitolo si esamina la memoria virtuale nella forma della paginazione su richiesta e se ne valutano complessità e costi.

## 9.1 Introduzione

Gli algoritmi di gestione della memoria delineati nel Capitolo 8 sono necessari perché, per l'attivazione di un processo, le istruzioni da eseguire si devono trovare all'interno della memoria fisica. Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo relativo in memoria fisica. Il caricamento dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatori.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica. In effetti, da un esame dei pro-

grammi reali risulta che in molti casi non è necessario avere in memoria l'intero programma; si considerino ad esempio le seguenti situazioni.

- ◆ Spesso i programmi dispongono di codice per la gestione di condizioni d'errore insolite. Poiché questi errori sono rari, se non inesistenti, anche i relativi segmenti di codice non si eseguono quasi mai.
- ◆ Spesso a array, liste e tabelle si assegna più memoria di quanta sia effettivamente necessaria. Un array si può dichiarare di 100 per 100 elementi, anche se raramente contiene più di 10 per 10 elementi. Una tabella dei simboli di un assemblatore può avere spazio per 3000 simboli, anche se un programma medio ne ha meno di 200.
- ◆ Alcune opzioni e caratteristiche di un programma sono utilizzabili solo di rado.

Anche nei casi in cui è necessario disporre di tutto il programma è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria può essere vantaggiosa per i seguenti motivi.

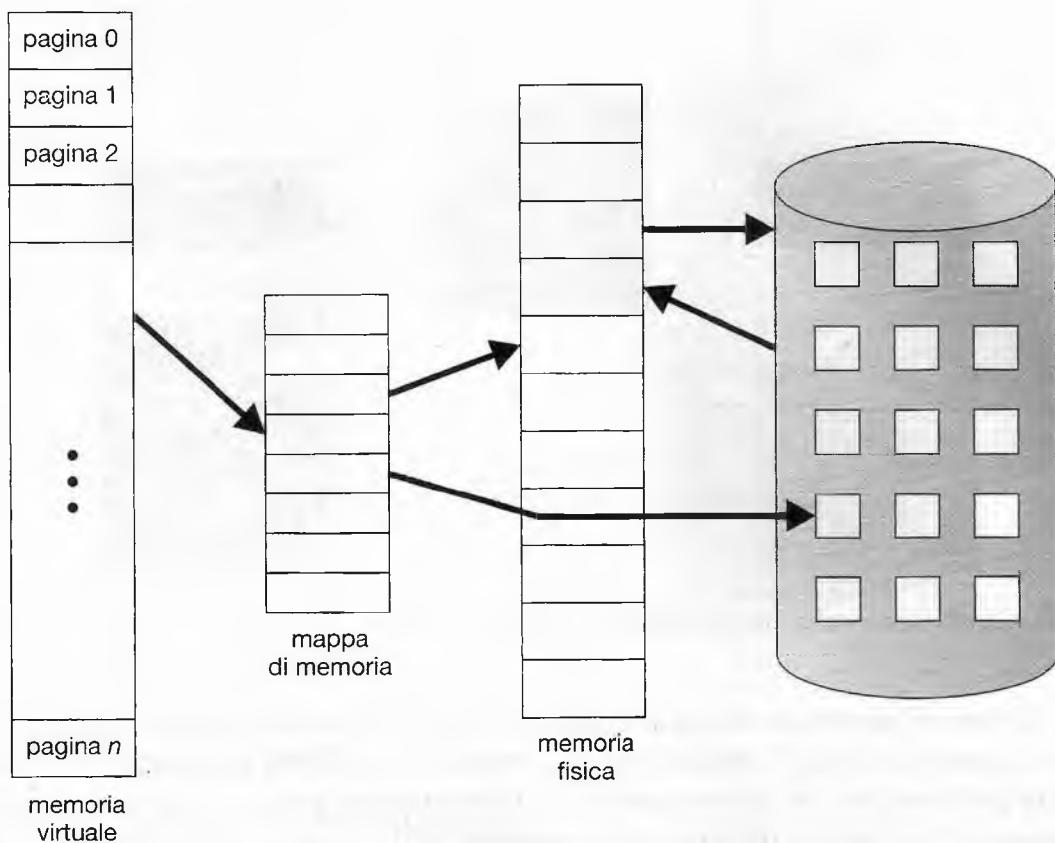
- ◆ Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno spazio degli indirizzi virtuali molto grande, semplificando così le operazioni di programmazione.
- ◆ Poiché ogni utente impiega meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della CPU senza aumentare il tempo di risposta o di completamento.
- ◆ Per caricare (o scaricare) ogni programma utente in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

La possibilità di eseguire un programma che non si trovi completamente in memoria apporterebbe quindi vantaggi sia al sistema sia all'utente.

La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmatore una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola, com'è illustrato nella Figura 9.1. La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile o di quale codice si debba inserire nelle sezioni sovrapponibili, ma può concentrarsi sul problema da risolvere.

L'espressione **spazio degli indirizzi virtuali** si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico – per esempio, l'indirizzo 0 – e si estende alla memoria contigua, come evidenziato dalla Figura 9.2. Come si ricorderà dal Capitolo 8, è tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (MMU) associare in memoria le pagine logiche alle pagine fisiche.

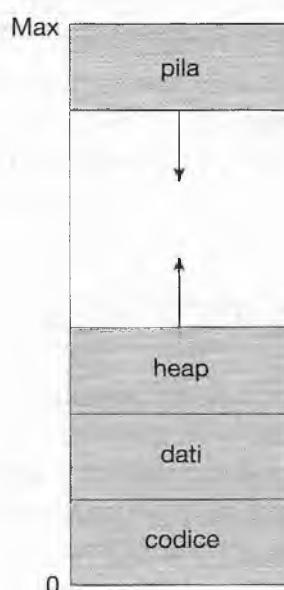
Si noti come, nella Figura 9.2, allo heap sia lasciato sufficiente spazio per crescere verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo alla pila di svilupparsi verso il basso nella memoria, a causa di ripetute chiamate di funzione. Lo spazio vuoto ben visibile (o buco) che separa lo heap dalla pila è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche realmente esistenti solo nel caso che lo heap o la pila crescano. Qualora contenga buchi, lo spazio degli indirizzi virtuali si definisce sparso. Un simile spazio degli indirizzi è doppiamente utile, poiché consente di riempire i buchi grazie all'espansione dei segmenti heap o pila, e di col-



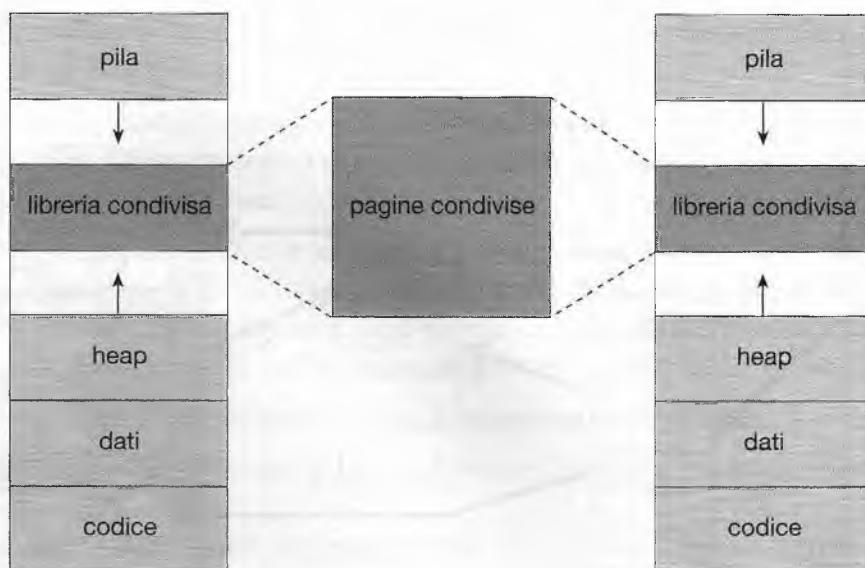
**Figura 9.1** Schema che mostra una memoria virtuale più grande di quella fisica.

legare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.

Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre, per due o più processi, il vantaggio di condividere i file e la memoria, mediante la condivisione delle pagine (Paragrafo 8.4.4). Ciò comporta i seguenti vantaggi.



**Figura 9.2** Spazio degli indirizzi virtuali.



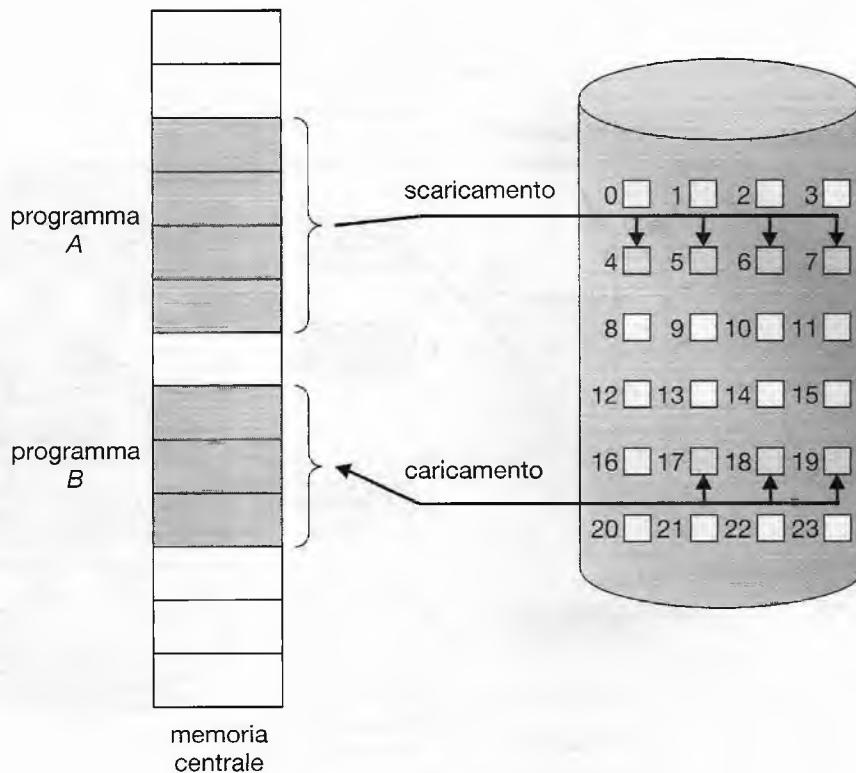
**Figura 9.3** Condivisione delle librerie tramite la memoria virtuale.

- Le librerie di sistema sono condivisibili da diversi processi associando l'oggetto condiviso a uno spazio degli indirizzi virtuali, procedimento detto **mappatura**. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi (Figura 9.3). In genere le librerie si associano allo spazio di ogni processo a loro collegato, in modalità di sola lettura.
- In maniera analoga, la memoria virtuale rende i processi in grado di condividere la memoria. Come si rammenterà dal Capitolo 3, due o più processi possono comunicare condividendo memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise, come illustrato dalla Figura 9.3.
- La memoria virtuale può consentire, per mezzo della chiamata di sistema `fork()`, che le pagine siano condivise durante la creazione di un processo, così da velocizzare la generazione dei processi.

Approfondiremo questi e altri vantaggi offerti dalla memoria virtuale nel corso di questo capitolo. In primo luogo, però, ci soffermeremo sulla memoria virtuale realizzata attraverso la paginazione su richiesta.

## 9.2 Paginazione su richiesta

Si consideri il caricamento in memoria di un eseguibile residente su disco. Una possibilità è quella di caricare l'intero programma nella memoria fisica al momento dell'esecuzione. Il problema, però, è che all'inizio non è detto che serva avere tutto il programma in memoria: se il programma, per esempio, fornisce all'avvio una lista di opzioni all'utente, è inutile caricare il codice per l'esecuzione di *tutte* le opzioni previste, senza tener conto di quella effettivamente scelta dall'utente. Una strategia alternativa consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica, detta **paginazione su richiesta**, comunemente adottata dai sistemi con memoria virtuale. Secondo questo schema, le pagine



**Figura 9.4** Trasferimento di una memoria paginata nello spazio contiguo di un disco.

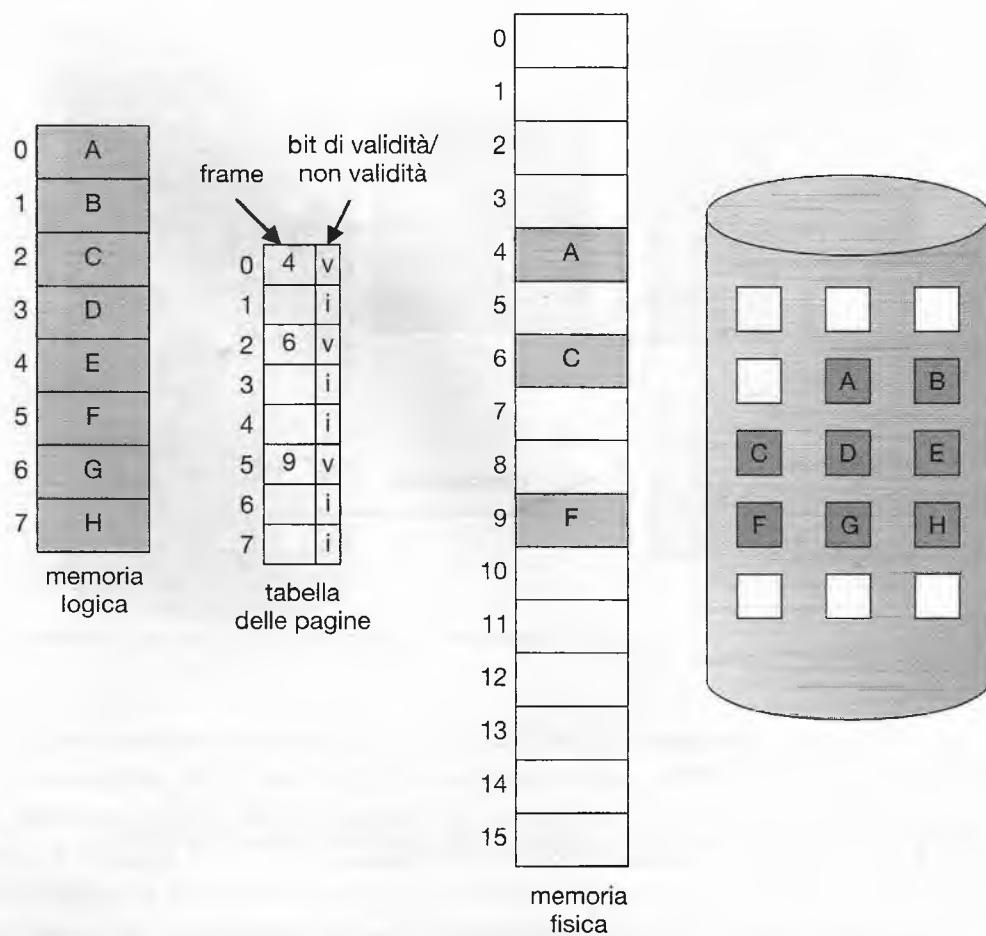
sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.

Un sistema di paginazione su richiesta è analogo a un sistema paginato con avvicendamento dei processi in memoria; si veda la Figura 9.4. I processi risiedono in memoria secondaria, generalmente costituita di uno o più dischi. Per eseguire un processo occorre caricarlo in memoria. Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un criterio d'avvicendamento "pigro" (*lazy swapping*): non si carica mai in memoria una pagina che non sia necessaria. Poiché stiamo considerando un processo come una sequenza di pagine, invece che come un unico ampio spazio d'indirizzi contiguo, l'uso del termine *avvicendamento dei processi* non è appropriato: non si manipolano interi processi ma singole pagine di processi. Nell'ambito della paginazione su richiesta, il modulo del sistema operativo che si occupa della sostituzione delle pagine si chiama **paginatore** (*pager*).

### 9.2.1 Concetti fondamentali

Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Anziché caricare in memoria tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare il trasferimento in memoria di pagine che non sono effettivamente usate, riducendo il tempo d'avvicendamento e la quantità di memoria fisica richiesta.

Con tale schema è necessario che l'architettura disponga di un qualche meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. A tal fine è utilizzabile lo schema basato sul bit di validità, descritto nel Paragrafo 8.5. In questo caso, però, il bit impostato come "valido" significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come "non valido" indica che la pagina non è valida (cioè



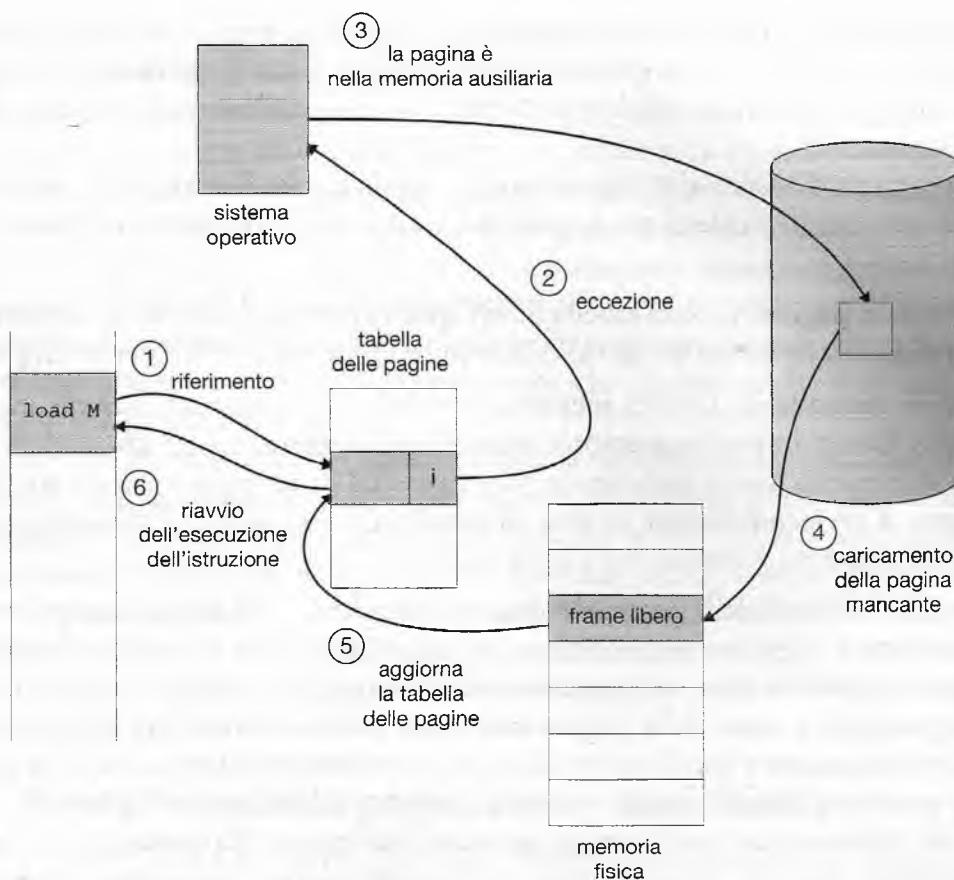
**Figura 9.5** Tabella delle pagine quando alcune pagine non si trovano nella memoria centrale.

non appartiene allo spazio d'indirizzi logici del processo) oppure è valida ma è attualmente nel disco. L'elemento della tabella delle pagine di una pagina caricata in memoria s'imposta come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido o contiene l'indirizzo che consente di trovare la pagina nei dischi. Tale situazione è illustrata nella Figura 9.5.

Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi. Quindi, se l'ipotesi del paginatore è esatta e si carichino tutte e solo le pagine che servono effettivamente, il processo è eseguito proprio come se fossero state caricate tutte le pagine. Durante l'esecuzione, il processo accede alle pagine **residenti in memoria**, e l'esecuzione procede come di consueto.

Se il processo tenta l'accesso a una pagina che non era stata caricata in memoria, l'accesso a una pagina contrassegnata come non valida causa un'**eccezione di pagina mancante** (*page fault trap*). L'architettura di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e invia un segnale di eccezione al sistema operativo; tale eccezione è dovuta a un "insuccesso" del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell'eccezione di pagina mancante (Figura 9.6) è chiara, e corrisponde ai passi seguenti.

1. Si controlla una tabella interna per questo processo; in genere tale tabella è conservata insieme al blocco di controllo di processo (PCB), allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.



**Figura 9.6** Fasi di gestione di un'eccezione di pagina mancante.

- Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua l'inserimento.
- Si individua un frame libero, ad esempio prelevandone uno dalla lista dei frame liberi.
- Si programma un'operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
- Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
- Si riavvia l'istruzione interrotta dal segnale di eccezione. A questo punto il processo può accedere alla pagina come se questa fosse già presente in memoria.

È addirittura possibile avviare l'esecuzione di un processo *senza* pagine in memoria. Quando il sistema operativo carica nel contatore di programma l'indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo accusa un'assenza di pagina. Una volta portata la pagina in memoria, il processo continua l'esecuzione, subendo assenze di pagine fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una **paginazione su richiesta pura**, vale a dire che una pagina non si trasferisce in memoria finché non sia richiesta.

In teoria alcuni programmi possono accedere a più pagine di memoria all'esecuzione di ogni istruzione (una pagina per l'istruzione e molte per i dati), causando più possibili eccezioni di pagine mancanti per ogni istruzione. In un caso simile le prestazioni del sistema

sarebbero inaccettabili. Un'analisi dei processi in esecuzione mostra che questo comportamento è molto improbabile. I programmi tendono ad avere una **località dei riferimenti**, descritta nel Paragrafo 9.6.1, quindi le prestazioni della paginazione su richiesta rientrano in un campo ragionevole.

I meccanismi d'ausilio alla paginazione su richiesta che l'architettura del calcolatore deve offrire sono quelli richiesti per la paginazione e l'avvicendamento dei processi in memoria:

- ◆ **tabella delle pagine.** Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- ◆ **memoria secondaria.** Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo d'avvicendamento; la sezione del disco usata a questo scopo si chiama **area d'avvicendamento**, o **area di scambio** (*swap space*). L'allocazione dell'area d'avvicendamento è trattata nel Capitolo 12.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione a seguito di un'eccezione di pagina mancante o assenza di pagina (*page fault*). Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) a causa della pagina mancante, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questa situazione è piuttosto comune: un'assenza di pagina si può verificare per qualsiasi riferimento alla memoria. Se l'assenza di pagina si presenta durante la fase di prelievo di un'istruzione, l'esecuzione si può riavviare prelevando nuovamente tale istruzione. Se si verifica durante il prelievo di un operando, l'istruzione deve essere di nuovo prelevata e decodificata, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione a tre indirizzi, come ad esempio la somma (**ADD**) del contenuto di **A** al contenuto di **B**, con risultato posto in **C**. I passi necessari per eseguire l'istruzione sono i seguenti:

1. prelievo e decodifica dell'istruzione (**ADD**);
2. prelievo del contenuto di **A**;
3. prelievo del contenuto di **B**;
4. addizione del contenuto di **A** al contenuto di **B**;
5. memorizzazione della somma in **C**.

Se l'assenza di pagina avviene al momento della memorizzazione in **C**, poiché **C** si trova in una pagina che non è in memoria, occorre prelevare la pagina desiderata, caricarla in memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'istruzione richiede una nuova operazione di prelievo della stessa, con nuova decodifica e nuovo prelievo dei due operandi; infine occorre ripetere l'addizione. In ogni modo il lavoro da ripetere non è molto, meno di un'istruzione completa, e la ripetizione è necessaria solo nel caso si verifichi un'assenza di pagina.

La difficoltà maggiore si presenta quando un'istruzione può modificare parecchie locazioni diverse. Si consideri, ad esempio, l'istruzione **MVC** (*move character*) del sistema IBM 360/370: quest'istruzione può spostare una sequenza di byte (fino a 256) da una locazione a un'altra con possibilità di sovrapposizione. Se una delle sequenze (quella d'origine o quella di destinazione) esce dal confine di una pagina, e se lo spostamento è stato effettuato solo in parte, si può verificare un'assenza di pagina. Inoltre, se le sequenze d'origine e di destina-

zione si sovrappongono, è probabile che la sequenza d'origine sia stata modificata, in tal caso non è possibile limitarsi a riavviare l'istruzione.

Il problema si può risolvere in due modi. In una delle due soluzioni il microcodice computa e tenta di accedere alle estremità delle due sequenze di byte. Un'eventuale assenza di pagina si può verificare solo in questa fase, prima che si apporti qualsiasi modifica. A questo punto si può compiere lo spostamento perché tutte le pagine interessate si trovano in memoria. L'altra soluzione si serve di registri temporanei per conservare i valori delle locazioni sovrascritte. Nel caso di un'assenza di pagina, si riscrivono tutti i vecchi valori in memoria prima che sia emesso il segnale di eccezione di pagina mancante. Questa operazione riporta la memoria allo stato in cui si trovava prima che l'istruzione fosse avviata, perciò si può ripetere la sua esecuzione.

Sebbene non si tratti certo dell'unico problema da affrontare per estendere un'architettura esistente con la funzionalità della paginazione su richiesta, illustra alcune delle difficoltà da superare. Il sistema di paginazione si colloca tra la CPU e la memoria di un calcolatore e deve essere completamente trasparente al processo utente. L'opinione comune che la paginazione si possa aggiungere a qualsiasi sistema è vera per gli ambienti senza paginazione su richiesta, nei quali un'eccezione di pagina mancante rappresenta un errore fatale, ma è falsa nei casi in cui un'eccezione di pagina mancante implica la necessità di caricare in memoria un'altra pagina e quindi riavviare il processo.

## 9.2.2 Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il **tempo d'accesso effettivo** per una memoria con paginazione su richiesta. Attualmente, nella maggior parte dei calcolatori il tempo d'accesso alla memoria, che si denota *ma*, varia da 10 a 200 nanosecondi. Finché non si verifichino assenze di pagine, il tempo d'accesso effettivo è uguale al tempo d'accesso alla memoria. Se però si verifica un'assenza di pagina, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola della memoria desiderata.

Supponendo che  $p$  sia la probabilità che si verifichi un'assenza di pagina ( $0 \leq p \leq 1$ ), è probabile che  $p$  sia molto vicina allo zero, cioè che ci siano solo poche assenze di pagine. Il **tempo d'accesso effettivo** è dato dalla seguente espressione:

$$\text{tempo d'accesso effettivo} = (1 - p) \times ma + p \times \text{tempo di gestione dell'assenza di pagina}$$

Per calcolare il tempo d'accesso effettivo occorre conoscere il tempo necessario alla gestione di un'assenza di pagina. Alla presenza di un'assenza di pagina si esegue la seguente sequenza:

1. segnale d'eccezione al sistema operativo;
2. salvataggio dei registri utente e dello stato del processo;
3. verifica che l'interruzione sia dovuta o meno a una pagina mancante;
4. controllo della correttezza del riferimento alla pagina e determinazione della locazione della pagina nel disco;
5. lettura dal disco e trasferimento in un frame libero:
  - a) attesa nella coda relativa a questo dispositivo finché la richiesta di lettura non sia servita;
  - b) attesa del tempo di posizionamento e latenza del dispositivo;
  - c) inizio del trasferimento della pagina in un frame libero;

6. durante l'attesa, allocazione della CPU a un altro processo utente (scheduling della CPU, facoltativo);
7. ricezione di un'interruzione dal controllore del disco (I/O completato);
8. salvataggio dei registri e dello stato dell'altro processo utente (se è stato eseguito il passo 6);
9. verifica della provenienza dell'interruzione dal disco;
10. aggiornamento della tabella delle pagine e di altre tabelle per segnalare che la pagina richiesta è attualmente presente in memoria;
11. attesa che la CPU sia nuovamente assegnata a questo processo;
12. recupero dei registri utente, dello stato del processo e della nuova tabella delle pagine, quindi ripresa dell'istruzione interrotta.

Non sempre sono necessari tutti i passi sopra elencati. Nel passo 6, ad esempio, si ipotizza che la CPU sia assegnata a un altro processo durante un'operazione di I/O. Tale possibilità permette la multiprogrammazione per mantenere occupata la CPU, ma una volta completato il trasferimento di I/O implica un dispendio di tempo per riprendere la procedura di servizio dell'eccezione di pagina mancante.

In ogni caso, il tempo di servizio dell'eccezione di pagina mancante comporta tre operazioni principali:

1. servizio del segnale di eccezione di pagina mancante;
2. lettura della pagina;
3. riavvio del processo.

La prima e la terza operazione si possono realizzare, per mezzo di un'accurata codifica, in parecchie centinaia di istruzioni. Ciascuna di queste operazioni può richiedere da 1 a 100 microsecondi. D'altra parte, il tempo di cambio di pagina è probabilmente vicino a 8 millisecondi. Un disco ha in genere un tempo di latenza di 3 millisecondi, un tempo di posizionamento di 5 millisecondi e un tempo di trasferimento di 0,05 millisecondi, quindi il tempo totale della paginazione è dell'ordine di 8 millisecondi, compresi i tempi d'esecuzione del codice relativo e delle operazioni dei dispositivi fisici coinvolti. Nel calcolo si è considerato solo il tempo di servizio del dispositivo. Se una coda di processi è in attesa del dispositivo (altri processi che hanno accusato un'assenza di pagina) è necessario considerare anche il tempo di accodamento del dispositivo, poiché occorre attendere che il dispositivo di paginazione sia libero per servire la richiesta, quindi il tempo d'avvicendamento aumenta ulteriormente.

Considerando un tempo medio di servizio dell'eccezione di pagina mancante di 8 millisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$\begin{aligned}
 \text{tempo d'accesso effettivo} &= (1 - p) \times 200 + p \text{ (8 millisecondi)} \\
 &= (1 - p) \times 200 + p \times 8.000.000 \\
 &= 200 + 7.999.800 \times p
 \end{aligned}$$

Il tempo d'accesso effettivo è direttamente proporzionale alla frequenza delle assenze di pagina (*page-fault rate*). Se un accesso su 1000 accusa un'assenza di pagina, il tempo d'accesso effettivo è di 8,2 microsecondi. Impiegando la paginazione su richiesta, il calcolatore è ral-

lentato di un fattore pari a 40. Se si desidera un rallentamento inferiore al 10 per cento, occorre che valgano le seguenti condizioni:

$$\begin{aligned} 220 &> 200 + 7.999.800 \times p \\ 20 &> 7.999.800 \times p \\ p &< 0,00000025 \end{aligned}$$

Quindi, per mantenere a un livello ragionevole il rallentamento dovuto alla paginazione, si può permettere meno di un'assenza di pagina ogni 399.990 accessi alla memoria. In un sistema con paginazione su richiesta, è cioè importante tenere bassa la frequenza delle assenze di pagine, altrimenti il tempo effettivo d'accesso aumenta, rallentando molto l'esecuzione del processo.

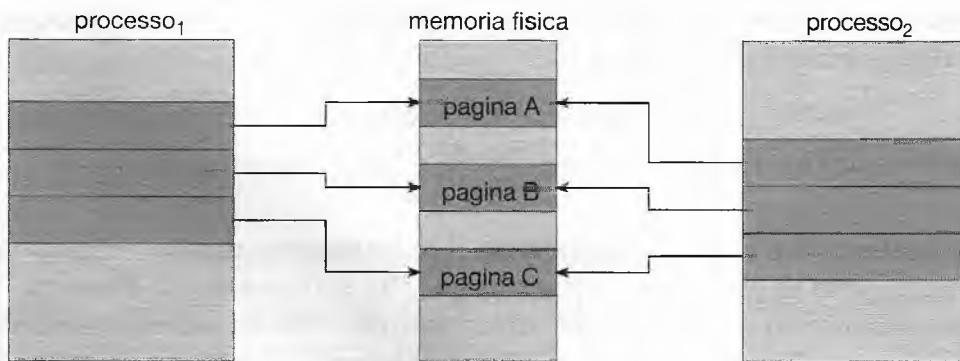
Un altro aspetto della paginazione su richiesta è la gestione e l'uso generale dell'area d'avvicendamento. L'I/O di un disco relativo all'area d'avvicendamento è generalmente più rapido di quello relativo al file system, poiché essa è composta di blocchi assai più grandi e non s'impiegano ricerche di file e metodi d'allocazione indiretta (Capitolo 12). Perciò il sistema può migliorare l'efficienza della paginazione copiando tutta l'immagine di un file nell'area d'avvicendamento all'avvio del processo e di lì eseguire la paginazione su richiesta. Un'altra possibilità consiste nel richiedere inizialmente le pagine al file system, ma scrivere le pagine nell'area d'avvicendamento al momento della sostituzione. Questo metodo assicura che si leggano sempre dal file system solo le pagine necessarie, ma che tutta la paginazione successiva sia fatta dall'area d'avvicendamento.

Quando s'impiegano file binari, alcuni sistemi tentano di limitare tale area: le pagine richieste per questi file si prelevano direttamente dal file system; tuttavia, quando è richiesta una sostituzione di pagine, i frame possono semplicemente essere sovrascritti, dato che non sono mai stati modificati, e le pagine, se è necessario, possono ancora essere lette dal file system. Seguendo questo criterio, lo stesso file system funziona da memoria ausiliaria (*backing store*). L'area d'avvicendamento si deve in ogni caso usare per le pagine che non sono relative ai file; queste comprendono la **pila** (*stack*) e lo **heap** di un processo. Questa tecnica che sembra essere un buon compromesso si usa in diversi sistemi tra cui Solaris e UNIX BSD.

## 9.3 Copiatura su scrittura

Nel Paragrafo 9.2 si è visto come un processo possa cominciare rapidamente l'esecuzione richiedendo solo la pagina contenente la prima istruzione. La generazione dei processi tramite **fork()**, però, può inizialmente evitare la paginazione su richiesta per mezzo di una tecnica simile alla condivisione delle pagine (Paragrafo 8.4.4), che garantisce la celere generazione dei processi riuscendo anche a minimizzare il numero di pagine allocate al nuovo processo.

Si ricordi che la chiamata di sistema **fork()** crea un processo figlio come duplicato del genitore. Nella sua versione originale la **fork()** creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la loro creazione la chiamata di sistema **exec()**, questa operazione di copiatura risulta inutile. In alternativa, si può impiegare una tecnica nota come **copiatura su scrittura** (*copy-on-write*), il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrassegnano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una co-

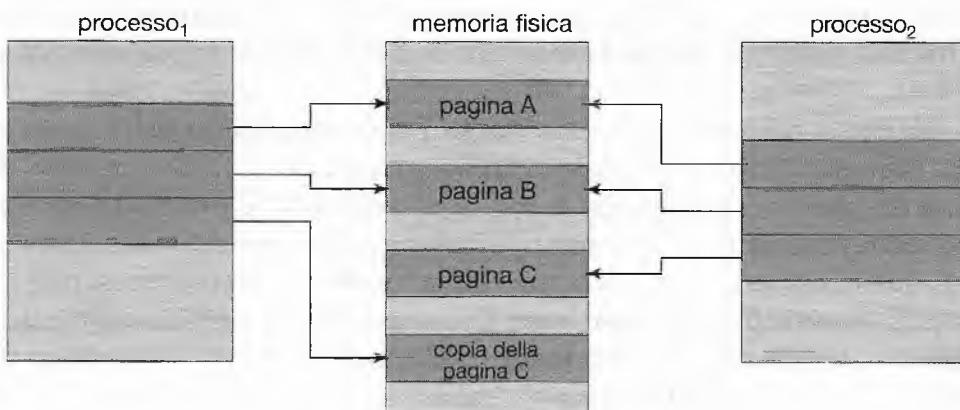


**Figura 9.7** Prima della modifica alla pagina C da parte del processo<sub>1</sub>.

pia di tale pagina. La copia su scrittura è illustrata rispettivamente nella Figura 9.7 e nella Figura 9.8, che mostrano il contenuto della memoria fisica prima e dopo che il processo 1 abbia modificato la pagina C.

Si consideri ad esempio un processo figlio che cerchi di modificare una pagina contenente parti della pila; il sistema operativo considera questa una pagina da copiare su scrittura e ne crea una copia nello spazio degli indirizzi del processo figlio. Il processo figlio modifica la sua copia della pagina e non la pagina appartenente al processo genitore. È chiaro che, adoperando la tecnica di copiatura su scrittura, si copiano soltanto le pagine modificate da uno dei due processi, mentre tutte le altre sono condivisibili dai processi genitore e figlio. Si noti inoltre che soltanto le pagine modificabili si devono contrassegnare come da copiare su scrittura, mentre quelle che non si possono modificare (ad esempio, le pagine contenenti codice eseguibile) sono condivisibili dai processi genitore e figlio. La tecnica di copiatura su scrittura è piuttosto comune e si usa in diversi sistemi operativi, tra i quali Windows XP, Linux e Solaris.

Quando è necessaria la duplicazione di una pagina secondo la tecnica di copiatura su scrittura, è importante capire da dove si attingerà la pagina libera necessaria. Molti sistemi operativi forniscono, per queste richieste, un gruppo (*pool*) di pagine libere, che di solito si assegnano quando la pila o il cosiddetto *heap* di un processo devono espandersi, oppure proprio per gestire pagine da copiare su scrittura. L'allocazione di queste pagine di solito avviene secondo una tecnica nota come **azzeramento su richiesta** (*zero-fill-on-demand*); prima dell'allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.



**Figura 9.8** Dopo la modifica alla pagina C da parte del processo<sub>1</sub>.

Diverse versioni di UNIX (compreso Solaris e Linux) offrono anche una variante della chiamata di sistema `fork()` – detta `vfork()` (per *virtual memory fork*). La `vfork()` offre un’alternativa all’uso della `fork()` con copiatura su scrittura. Con la `vfork()` il processo genitore viene sospeso e il processo figlio usa lo spazio d’indirizzi del genitore. Poiché la `vfork()` non usa la copiatura su scrittura, se il processo figlio modifica qualche pagina dello spazio d’indirizzi del genitore, le pagine modificate saranno visibili al processo genitore non appena riprenderà il controllo. Per assicurarsi che il processo figlio non modifichi lo spazio d’indirizzi del genitore è quindi necessaria molta attenzione nell’uso di `vfork()`. La chiamata di sistema `vfork()` è adatta al caso in cui il processo figlio esegua una `exec()` immediatamente dopo la sua creazione. Poiché non richiede alcuna copiatura delle pagine, la `vfork()` è un metodo di creazione dei processi molto efficiente, in alcuni casi impiegato per realizzare le interfacce degli interpreti dei comandi in UNIX.

## 9.4 Sostituzione delle pagine

Nelle descrizioni fatte finora, la frequenza (*rate*) delle assenze di pagine non è stata un problema grave, giacché ogni pagina poteva essere assente al massimo una volta, e precisamente la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l’I/O necessario per caricare le cinque pagine che non sono mai usate. Il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

Aumentando il grado di multiprogrammazione, si **sovrassegna** la memoria. Eseguendo sei processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s’incrementerebbero l’utilizzo e la produttività della CPU e si risparmierebbero 10 frame. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l’I/O impegnano una rilevante quantità di memoria. Ciò può aumentare le difficoltà agli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all’I/O e quanta alle pagine dei programmi è un problema rilevante. Alcuni sistemi riservano una quota fissa di memoria per l’I/O, altri permettono sia ai processi utenti sia al sottosistema di I/O di competere per tutta la memoria del sistema.

La **sovrallocazione** (*over-allocation*) si può illustrare come segue. Durante l’esecuzione di un processo utente si verifica un’assenza di pagina. Il sistema operativo determina la localizzazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è *vuota*: tutta la memoria è in uso; si veda a questo proposito la Figura 9.9.

A questo punto il sistema operativo può scegliere tra diverse possibilità, ad esempio può terminare il processo utente. Tuttavia, la paginazione su richiesta è un tentativo che il sistema operativo fa per migliorare l’utilizzo e la produttività del sistema di calcolo. Gli utenti non devono sapere che i loro processi sono eseguiti su un sistema paginato. La paginazione deve essere logicamente trasparente per l’utente, quindi la terminazione del processo non costituisce la scelta migliore.

na causa un'eccezione di pagina mancante. In quel momento, la pagina viene riportata in memoria e può sostituire un'altra pagina del processo.

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un **algoritmo di allocazione dei frame** e un **algoritmo di sostituzione delle pagine**. Se sono presenti più processi in memoria, occorre decidere quanti frame vanno assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire. La progettazione di algoritmi idonei a risolvere questi problemi è un compito importante, poiché l'I/O nei dischi è piuttosto oneroso. Anche miglioramenti minimi ai metodi di paginazione su richiesta ne apportano notevoli alle prestazioni del sistema.

Esistono molti algoritmi di sostituzione delle pagine; probabilmente ogni sistema operativo ha il proprio schema di sostituzione. È quindi necessario stabilire un criterio per selezionare un algoritmo di sostituzione particolare; generalmente si sceglie quello con la minima frequenza delle assenze di pagine (*page-fault rate*).

Un algoritmo si valuta effettuandone l'esecuzione su una particolare successione di riferimenti alla memoria e calcolando il numero di assenze di pagine. La successione dei riferimenti alla memoria è detta, appunto, **successione dei riferimenti**. Queste successioni si possono generare artificialmente (ad esempio con un generatore di numeri casuali), oppure analizzando un dato sistema e registrando l'indirizzo di ciascun riferimento alla memoria. Quest'ultima opzione permette di ottenere un numero elevato di dati, dell'ordine di un milione di indirizzi al secondo. Per ridurre questa quantità di dati occorre notare due fatti.

Innanzitutto, di una pagina di date dimensioni, generalmente fissate dall'architettura del sistema, si considera solo il numero della pagina anziché l'intero indirizzo. In secondo luogo, quando si fa riferimento a una pagina  $p$ , i riferimenti alla stessa pagina *immediatamente* successivi al primo non accusano assenze di pagine: dopo il primo riferimento, la pagina  $p$  è presente in memoria.

Esaminando un processo si potrebbe ad esempio registrare la seguente successione di indirizzi:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,  
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

che, a 100 byte per pagina, si riduce alla seguente successione di riferimenti:

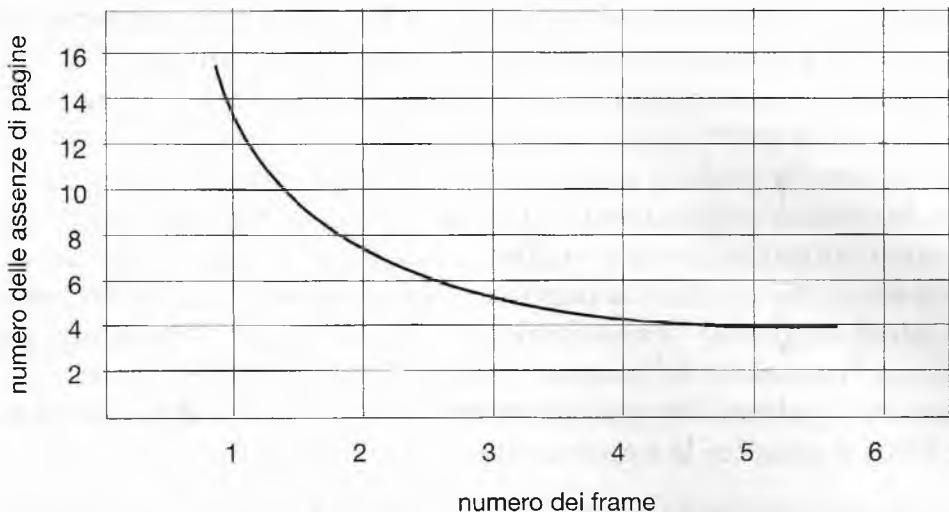
1, 4, 1, 6, 1, 6, 1, 6, 1

Per stabilire il numero di assenze di pagine relativo a una particolare successione di riferimenti e a un particolare algoritmo di sostituzione delle pagine, occorre conoscere anche il numero dei frame disponibili. Naturalmente, aumentando il numero di quest'ultimi diminuisce il numero di assenze di pagine. Per la successione dei riferimenti precedentemente esaminata, ad esempio, dati tre o più blocchi di memoria si possono verificare tre sole assenze di pagine: una per il primo riferimento di ogni pagina. D'altra parte, se si dispone di un solo frame è necessaria una sostituzione per ogni riferimento, con il risultato di 11 assenze di pagine. In generale è prevista una curva simile a quella della Figura 9.11. Aumentando il numero dei frame, il numero di assenze di pagine diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

Per illustrare gli algoritmi di sostituzione delle pagine s'impiega la seguente successione di riferimenti

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

per una memoria con tre frame.

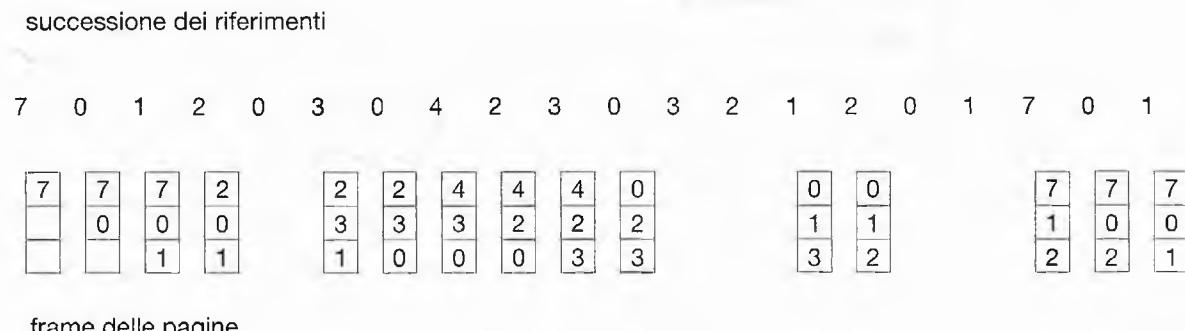


**Figura 9.11** Grafico che illustra il numero di assenze di pagine rispetto al numero dei frame.

## 9.4.2 Sostituzione delle pagine secondo l'ordine d'arrivo (FIFO)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo. Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si possono strutturare secondo una coda FIFO tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova nel primo elemento della coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

Nella successione di riferimenti adottata, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7, 0, 1) accusano ciascuno un'assenza di pagina con conseguente caricamento delle relative pagine nei frame vuoti. Il riferimento successivo (2) causa la sostituzione della pagina 7, perché essa è stata caricata per prima in memoria. Siccome 0 è il riferimento successivo e si trova già in memoria, per questo riferimento non ha luogo alcuna assenza di pagina. Il primo riferimento a 3 causa la sostituzione della pagina 0, che era la prima fra le tre pagine in memoria (0, 1, e 2) da caricare. A causa di questa sostituzione il riferimento successivo, a 0, accuserà un'assenza di pagina. La pagina 1 è allora sostituita dalla pagina 0. Questo processo prosegue come è illustrato nella Figura 9.12. Le pagine presenti nei tre frame sono indicate ogni volta che si verifica un'assenza di pagina. Complessivamente si hanno 15 assenze di pagine.



**Figura 9.12** Algoritmo di sostituzione delle pagine FIFO.

L'algoritmo FIFO di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un'eccezione di pagina mancante per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, scegliendo una sostituzione errata, aumenta la frequenza di pagine mancanti che rallenta l'esecuzione del processo, ma non vengono causati errori.

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine FIFO, si consideri la seguente successione di riferimenti:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

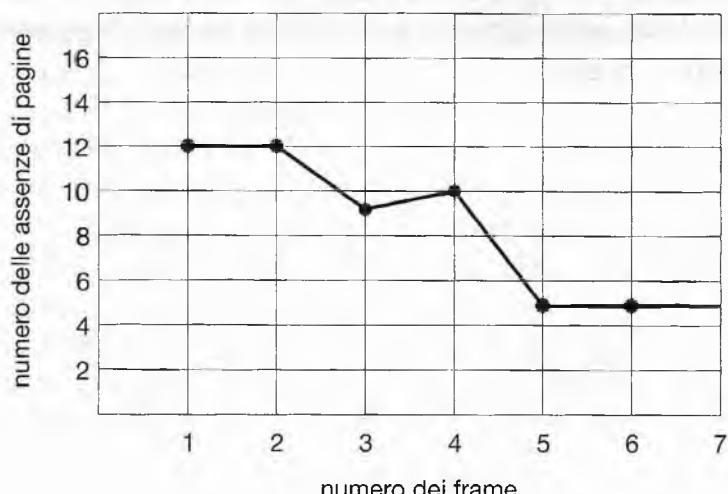
Nella Figura 9.13 è illustrata la curva delle assenze di pagine in funzione del numero dei frame disponibili. Occorre notare che il numero delle assenze di pagine (10) per quattro frame è *maggiore* del numero delle assenze di pagine (9) per tre frame. Questo inatteso risultato è noto col nome di **anomalia di Belady**, e riflette il fatto che, con alcuni algoritmi di sostituzione delle pagine, la frequenza delle assenze di pagine può *aumentare* con l'aumentare del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. Si è invece notato che questo presupposto non sempre è vero; l'anomalia di Belady ne è la prova.

### 9.4.3 Sostituzione ottimale delle pagine

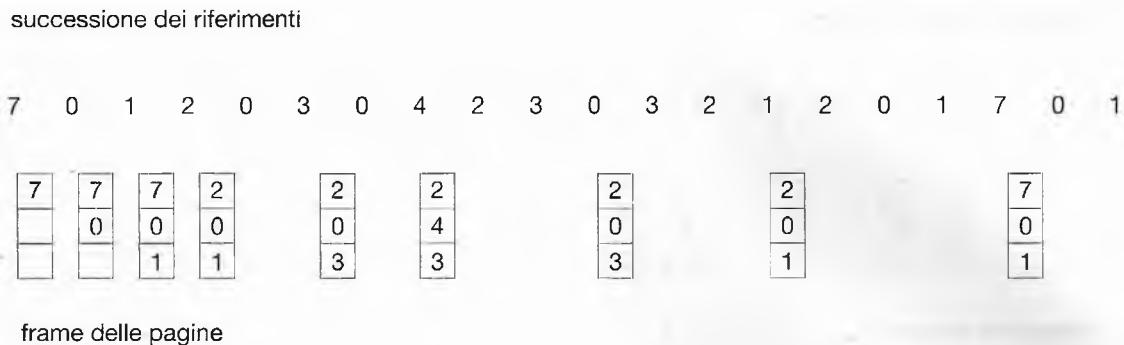
In seguito alla scoperta dell'anomalia di Belady, la ricerca si è diretta verso un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo è quello che fra tutti gli algoritmi presenta la minima frequenza di assenze di pagine e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato OPT o MIN. È semplicemente:

*si sostituisce la pagina che non si userà per il periodo di tempo più lungo.*

L'uso di quest'algoritmo di sostituzione delle pagine assicura la frequenza di assenze di pagine più bassa possibile per un numero fisso di frame.



**Figura 9.13** Curva delle assenze di pagine per sostituzione FIFO su una successione di riferimenti.



**Figura 9.14** Algoritmo ottimale di sostituzione delle pagine.

Ad esempio, nella successione dei riferimenti considerata, l'algoritmo ottimale di sostituzione delle pagine produce nove assenze di pagine, come è mostrato nella Figura 9.14. I primi tre riferimenti causano assenze di pagine che riempiono i tre blocchi di memoria vuoti. Il riferimento alla pagina 2 determina la sostituzione della pagina 7, perché la 7 non è usata fino al riferimento 18, mentre la pagina 0 viene usata al 5 e la pagina 1 al 14. Il riferimento alla pagina 3 causa la sostituzione della pagina, poiché la pagina 1 è l'ultima delle tre pagine in memoria cui si fa nuovamente riferimento. Con sole nove assenze di pagine, la sostituzione ottimale risulta assai migliore di quella ottenuta con un algoritmo FIFO, dove le assenze di pagine erano 15. Ignorando le prime tre assenze di pagine, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore rispetto all'algoritmo FIFO; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove assenze di pagine.

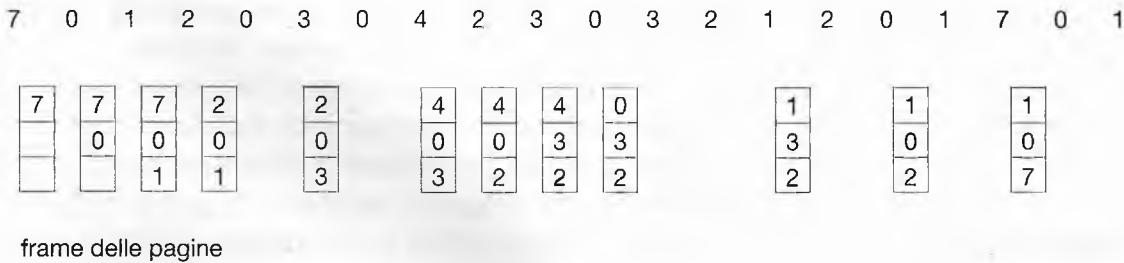
Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti. Una situazione analoga si è riscontrata con l'algoritmo SJF di scheduling della CPU, nel Paragrafo 5.3.2. Quindi, l'algoritmo ottimale si impiega soprattutto per studi comparativi. Ad esempio, può risultare abbastanza utile sapere che, sebbene un algoritmo nuovo non sia ottimale, nel peggiore dei casi le sue prestazioni sono inferiori del 12,3 per cento rispetto a quelle dell'algoritmo ottimale, e mediamente questa percentuale è del 4,7 per cento.

#### 9.4.4 Sostituzione delle pagine usate meno recentemente (LRU)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina è *usata*. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che *non è stata usata* per il periodo più lungo. Il metodo appena descritto è noto come **algoritmo LRU** (*least recently used*).

La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Questa strategia costituisce l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Infatti, supponendo che  $S^R$  sia la successione inversa di una successione di riferimenti  $S$ , la frequenza di assenze di pagine per l'algoritmo OPT su  $S$  è uguale a quella per l'algoritmo LRU su  $S^R$ . Allo stesso modo, la frequenza di assenze di pagine per l'algoritmo LRU su  $S$  è uguale a quella per l'algoritmo OPT su  $S^R$ .

successione dei riferimenti



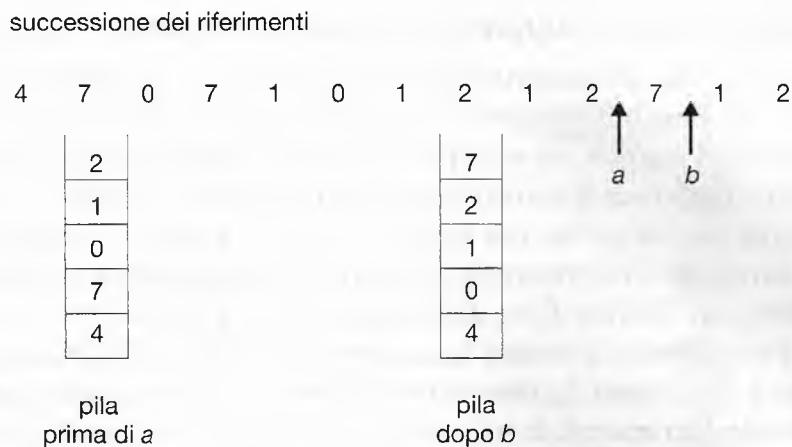
frame delle pagine

**Figura 9.15** Algoritmo di sostituzione delle pagine LRU.

Il risultato dell'applicazione dell'algoritmo LRU alla successione dei riferimenti dell'esempio è illustrato nella Figura 9.15. L'algoritmo LRU produce 12 assenze di pagine. Occorre notare che le prime cinque assenze di pagine sono le stesse della sostituzione ottimale. Quando si presenta il riferimento alla pagina 4, però, l'algoritmo LRU trova che, fra i tre blocchi di memoria, quello usato meno recentemente è della pagina 2. Quindi, l'algoritmo LRU sostituisce la pagina 2 senza sapere che sta per essere usata. Quando si verifica l'assenza della pagina 2, l'algoritmo LRU sostituisce la pagina 3, poiché, fra le tre pagine in memoria (0, 3, 4), la pagina 3 è quella usata meno recentemente. Nonostante questi problemi, la sostituzione LRU, con 12 assenze di pagine, è in ogni modo migliore della sostituzione FIFO, con 15 assenze di pagine.

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la realizzazione della sostituzione stessa. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'architettura del sistema di calcolo. Il problema consiste nel determinare un ordine per i frame definito secondo il momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni.

- ◆ **Contatori.** Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo del momento d'uso, e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. Ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo del momento d'uso nella tabella relativa a quella specifica pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina usata meno recentemente (LRU), e una scrittura in memoria (nel campo del momento d'uso della tabella delle pagine) per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della CPU, si modificano le tabelle delle pagine. Occorre infine considerare il superamento della capacità del contatore (*overflow*).
- ◆ **Pila.** Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede la presenza di una pila dei numeri delle pagine. Ogni volta che si fa un riferimento a una pagina, la si estrae dalla pila e la si colloca in cima a quest'ultima. In questo modo, in cima alla pila si trova sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente, com'è illustrato dalla Figura 9.16. Poiché alcuni elementi si devono estrarre dal centro della pila, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dalla pila e collocarla in cima, nel caso peggiore è necessario modificare sei puntatori. Ogni aggiornamento è un po' più costoso, ma per una sostituzione non si deve compiere alcuna ricerca; il puntatore dell'elemento di co-



**Figura 9.16** Uso di una pila per registrare i più recenti riferimenti alle pagine.

da punta al fondo della pila, vale a dire la pagina usata meno recentemente. Questo metodo è adatto soprattutto alle realizzazioni programmate (o microprogrammate) della sostituzione delle pagine LRU.

Né la sostituzione ottimale né quella LRU sono soggette all'anomalia di Belady. Esiste una classe di algoritmi di sostituzione delle pagine, chiamati **algoritmi a pila**, che non presenta l'anomalia di Belady. Un algoritmo a pila è un algoritmo per il quale è possibile mostrare che l'insieme delle pagine in memoria per  $n$  frame è sempre un *sottoinsieme* dell'insieme delle pagine che dovrebbero essere in memoria per  $n + 1$  frame. Per la sostituzione LRU, l'insieme di pagine in memoria è costituito delle  $n$  pagine cui si è fatto riferimento più recentemente. Se il numero dei frame è aumentato, queste  $n$  pagine continuano a essere quelle cui si è fatto riferimento più recentemente e quindi restano in memoria.

Oltre i registri TLB standard, senza l'ausilio dell'architettura sarebbe inconcepibile anche la realizzazione della sostituzione LRU. L'aggiornamento dei campi del contatore o della pila si deve effettuare per *ogni* riferimento alla memoria. Se per ogni riferimento si dovesse adoperare un segnale d'interruzione per permettere alle procedure del sistema operativo di modificare tali strutture dati, tutti i riferimenti alla memoria sarebbero rallentati di un fattore almeno pari a 10, quindi anche tutti i processi utenti sarebbero rallentati di un uguale fattore. Pochi sistemi possono permettersi un tale sovraccarico per la gestione della memoria.

#### 9.4.5 Sostituzione delle pagine per approssimazione a LRU

Sono pochi i sistemi di calcolo che dispongono di un'architettura adatta a una vera sostituzione LRU delle pagine. Nei sistemi che non offrono tali caratteristiche specifiche si devono impiegare altri algoritmi di sostituzione delle pagine, ad esempio l'algoritmo FIFO. Molti sistemi tuttavia possono fornire un aiuto: un **bit di riferimento**. Il bit di riferimento a una pagina è impostato automaticamente dall'architettura del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

Inizialmente, il sistema operativo azzerà tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'architettura del sistema imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'*ordine* d'uso. È questa l'informazione alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

#### 9.4.5.1 Algoritmo con bit supplementari di riferimento

Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria una serie di bit per ogni pagina. A intervalli regolari, ad esempio di 100 millisecondi, un segnale d'interruzione del timer del sistema trasferisce il controllo al sistema operativo. Questo sposta il bit di riferimento per ciascuna pagina nel bit più significativo della sequenza, traslando gli altri bit a destra di 1 bit e scartando il bit meno significativo. Questi registri a scorrimento, ad esempio di 8 bit, contengono l'ordine d'uso delle pagine relativo agli ultimi otto periodi di tempo. Se il registro a scorrimento contiene la successione di bit 00000000, significa che la pagina associata non è stata usata da otto periodi di tempo; a una pagina usata almeno una volta per ogni periodo corrisponde la successione 11111111 nel registro a scorrimento. Una pagina cui corrisponde la successione 11000100 nel relativo registro, è stata usata più recentemente di quanto non lo sia stata una cui è associata la successione 01110111. Interpretando queste successioni di bit come interi senza segno, la pagina cui è associato il numero minore è la pagina LRU, e può essere sostituita. In ogni caso l'unicità dei numeri non è garantita. Si possono sostituire (o scaricare dalla memoria all'area d'avvicendamento) tutte le pagine con il valore minore, oppure si può ricorrere a una selezione FIFO.

Il numero dei bit può ovviamente essere variato: si stabilisce secondo l'architettura disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento e definendo un algoritmo noto come **algoritmo di sostituzione delle pagine con seconda chance**.

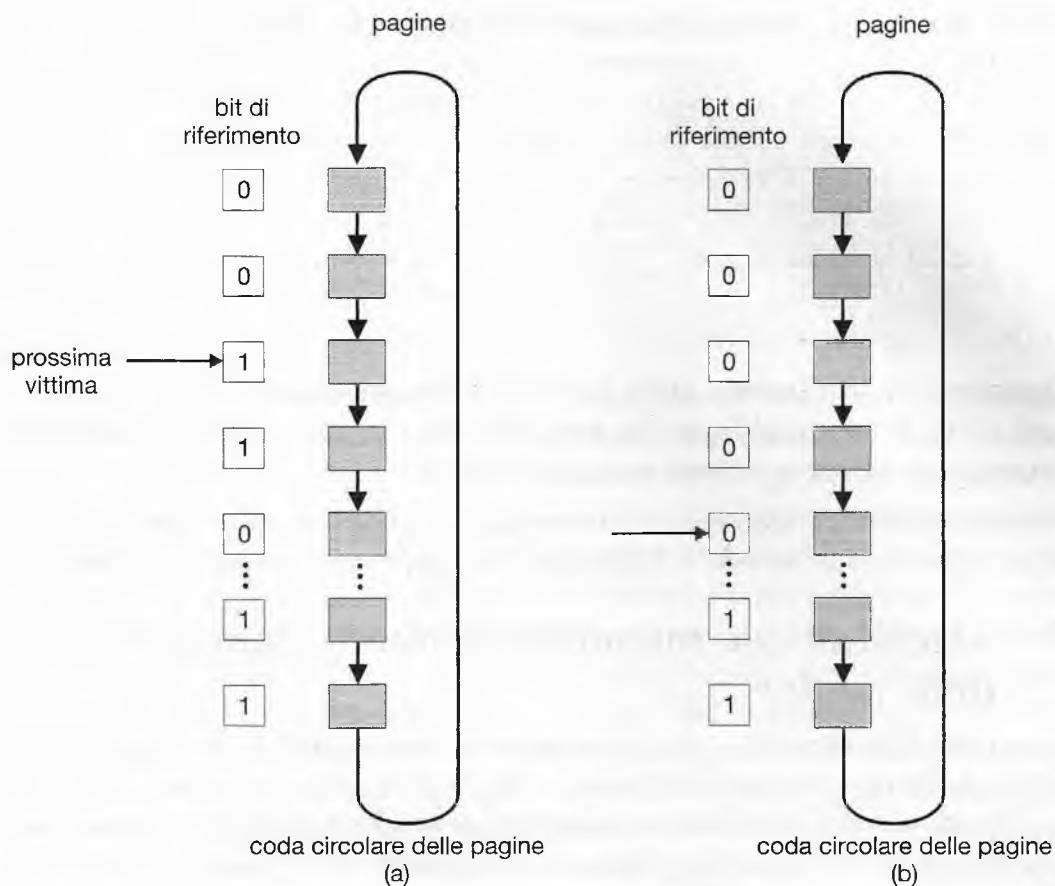
#### 9.4.5.2 Algoritmo con seconda chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Dopo aver selezionato una pagina si controlla il bit di riferimento, se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e la selezione passa alla successiva pagina FIFO. Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita fisché tutte le altre pagine non siano state sostituite, oppure non sia stata offerta loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, in modo che il suo bit di riferimento sia sempre impostato a 1, non viene mai sostituita.

Un metodo per realizzare l'algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull'uso di una coda circolare, in cui un puntatore indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato (Figura 9.17). Una volta trovata una pagina "vittima", la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.

#### 9.4.5.3 Algoritmo con seconda chance migliorato

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica (si veda il Paragrafo 9.4.1) come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:



**Figura 9.17** Algoritmo di sostituzione delle pagine con seconda chance (orologio).

1. (0, 0) né recentemente usato né modificato – migliore pagina da sostituire;
2. (0, 1) non usato recentemente, ma modificato – la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
3. (1, 0) usato recentemente ma non modificato – probabilmente la pagina sarà presto ancora usata;
4. (1, 1) usato recentemente e modificato – probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esaminano le classi cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che la coda circolare deve essere scandita più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo a orologio è che nel primo si dà la preferenza alle pagine modificate, al fine di ridurre il numero di I/O richiesti.

#### 9.4.6 Sostituzione delle pagine basata su conteggio

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Ad esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi.

- ◆ **Algoritmo di sostituzione delle pagine meno frequentemente usate (*least frequently used*, LFU)**; richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Il punto debole di questo algoritmo è rappresentato dai casi in cui una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Poiché è stata usata intensamente il suo conteggio è alto, quindi rimane in memoria anche se non è più necessaria. Una soluzione può essere quella di spostare i conteggi a destra di un bit a intervalli regolari, formando un conteggio per l'uso medio con esponente decrescente.
- ◆ **Algoritmo di sostituzione delle pagine più frequentemente usate (*most frequently used*, MFU)**; è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Le sostituzioni MFU e LFU non sono molto comuni, poiché la realizzazione di questi algoritmi è abbastanza onerosa; inoltre, tali algoritmi non approssimano bene la sostituzione OPT.

#### 9.4.7 Algoritmi con memorizzazione transitoria delle pagine

Oltre a uno specifico algoritmo, per la sostituzione delle pagine si usano spesso anche altre procedure; ad esempio, i sistemi hanno generalmente un gruppo di frame liberi (*pool of free frames*). Quando si verifica un'assenza di pagina, si sceglie innanzi tutto un frame vittima, ma prima che sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame al gruppo dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogniqualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si reimposta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Anche un'altra modifica prevede l'uso di un gruppo di frame liberi ma, in questo caso, per ricordare quale pagina era contenuta in ciascun frame. Poiché quando si scrive il contenuto di un frame in un disco tale contenuto non cambia, se è necessaria, la vecchia pagina è ancora utilizzabile direttamente dal gruppo dei frame liberi, prima che quel sia riusato quel frame. In questo caso non è necessario alcun I/O. Se si verifica un'assenza di pagina occorre controllare se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Questa tecnica, insieme con l'algoritmo di sostituzione FIFO, è usata dal sistema VAX/VMS. Quando l'algoritmo FIFO sostituisce per errore una pagina ancora in uso, la si riprova rapidamente dal gruppo dei frame liberi senza ricorrere a operazioni di I/O. Il gruppo dei frame liberi offre protezione contro l'algoritmo di sostituzione FIFO, relativamente povero, ma semplice. Questo metodo è necessario poiché le prime versioni del VAX non disponevano del bit di riferimento correttamente realizzato.

Alcune versioni di UNIX adottano questo algoritmo insieme a quello con seconda chance. In effetti, si tratta di un'utile integrazione a qualunque algoritmo di sostituzione, al fine di ridurre il prezzo pagato per l'eventuale errata esclusione di una pagina.

### 9.4.8 Applicazioni e sostituzione della pagina

In taluni casi, le applicazioni che accedono ai dati tramite la memoria virtuale del sistema operativo non conseguono prestazioni migliori di quelle che il sistema, senza impiegare alcun buffer, potrebbe offrire. Si pensi, quale esempio tipico, a una base di dati che gestisce la memoria e il buffer dell'I/O in modo autonomo. Applicazioni come questa capiscono il funzionamento della memoria e del disco che occupano meglio di quanto possa fare un sistema operativo, che applica algoritmi adatti a un uso generale. Se il sistema operativo adotta un buffer per l'I/O, e così pure fa l'applicazione, la quantità di memoria necessaria per l'I/O sarà inutilmente raddoppiata.

Un altro esempio proviene dagli archivi di dati, che effettuano spesso impegnative letture sequenziali del disco, seguite da calcoli e scritture. L'algoritmo LRU eliminerebbe le pagine vecchie per conservare le nuove, mentre in questo caso è ragionevole attendersi la lettura delle pagine vecchie in luogo di quelle nuove (dato che l'applicazione esegue periodicamente la lettura sequenziale). In queste circostanze l'algoritmo MFU sarebbe più efficiente di LRU.

Per risolvere tali problemi, alcuni sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto **disco di basso livello** (*raw disk*), e il relativo I/O è denominato **I/O di basso livello** (*raw I/O*). Il disco di basso livello salta tutti i servizi del file system, come la paginazione su richiesta dei file in ingresso e in uscita, i lock dei file, il prefetching, l'allocazione dello spazio, i nomi dei file e le directory. Si noti come, sebbene alcune applicazioni siano più efficienti nel gestire i propri servizi specifici di memorizzazione sul disco di basso livello, quasi tutte hanno una resa migliore quando operano con i servizi regolari del file system.

## 9.5 Allocazione dei frame

Torniamo al concetto di allocazione. A questo punto occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, è possibile considerare un caso in cui 93 frame liberi si debbano assegnare a due processi.

Il caso più semplice di memoria virtuale è il sistema con utente singolo. Si consideri un sistema monoutente che disponga di 128 KB di memoria, con pagine di 1 KB. Complessivamente sono presenti 128 frame. Il sistema operativo può occupare 35 KB, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 blocchi di memoria sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di eccezioni di pagine mancanti. Le prime 93 pagine assenti ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la novantaquattresima, si può usare un algoritmo di sostituzione delle pagine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Questa strategia è semplice, ma può subire molte variazioni. Si può richiedere che il sistema operativo assegni tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un'assenza di pagina sia disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento della pagina, si può fare una sostituzione, la pagina coinvolta viene poi scritta nel disco mentre il processo utente

continua l'esecuzione. Sono possibili anche altre varianti, ma la strategia di base è chiara: al processo utente si assegna qualsiasi frame libero.

### 9.5.1 Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Naturalmente, col diminuire del numero dei frame allocati a ciascun processo aumenta la frequenza delle assenze di pagine, con conseguente rallentamento dell'esecuzione del processo. Esaminiamo quest'ultimo requisito in maggiore dettaglio.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati a ciascun processo aumenta la frequenza di mancanza di pagina, con conseguente ritardo dell'esecuzione dei processi. Inoltre va ricordato che, quando si verifica un'assenza di pagina prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata. Di conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.

Si consideri, ad esempio, un calcolatore in cui tutte le istruzioni di riferimento alla memoria hanno solo un indirizzo di memoria; in questo caso occorre almeno un frame per l'istruzione e uno per il riferimento alla memoria. Inoltre se è ammesso un indirizzamento indiretto a un livello (come nel caso di un'istruzione `Load` presente nella pagina 16 che può far riferimento a un indirizzo della pagina 0, che costituisce a sua volta un riferimento indiretto alla pagina 23) la paginazione richiede allora almeno tre blocchi di memoria per ogni processo. Si consideri che cosa accadrebbe nel caso di un processo che disponga di due soli frame.

Il numero minimo di frame è definito dall'hardware del calcolatore. Ad esempio, l'istruzione di spostamento del PDP-11, per alcune modalità di indirizzamento, è costituita di più di una parola, quindi la stessa istruzione può stare a cavallo tra due pagine. Inoltre, ciascuno dei suoi due operandi può essere un riferimento indiretto, per un totale di sei frame. Un altro esempio è dato dall'istruzione `MVC` di IBM 370. Poiché l'istruzione è da memoria a memoria, può occupare 6 byte e stare a cavallo tra due pagine. Anche la sequenza di caratteri da spostare e l'area su cui effettuare lo spostamento possono essere a cavallo tra due pagine; questa situazione richiede sei frame. In effetti, la situazione peggiore si presenta quando l'istruzione `MVC` è l'operando di un'istruzione `EXECUTE` che sta a cavallo di un limite di pagina; in questo caso occorrono otto frame.

Il caso peggiore si può presentare nelle architetture di calcolatori che permettono riferimenti indiretti a più livelli (ad esempio quando ogni parola di 16 bit può contenere un indirizzo di 15 bit più un indicatore indiretto di 1 bit). In teoria, una semplice istruzione di caricamento può far riferimento a un indirizzo indiretto che a sua volta può far riferimento a un indirizzo indiretto (su un'altra pagina) anch'esso facente riferimento a un indirizzo indiretto su un'altra pagina ancora, e così via, finché tutte le pagine della memoria virtuale siano state chiamate in causa. Quindi, nel caso peggiore, tutta la memoria virtuale si deve trovare in memoria fisica. Per superare questa difficoltà occorre porre un limite al livello dei riferimenti indiretti, ad esempio limitando un'istruzione a un massimo di 16 livelli. Quando si verifica il riferimento indiretto di primo livello, si imposta un contatore al valore 16, per decrementarlo a ciascun livello successivo relativo a questa istruzione. Se il contatore si riduce a 0 si verifica un segnale di eccezione (livello di riferimenti indiretti eccessivo). Tale limite riduce a 17 il numero massimo dei riferimenti alla memoria per ogni istruzione, richiedendo un pari numero di frame.

Il numero minimo di frame per ciascun processo è definito dall'architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile. Rimane ancora aperta la questione della scelta dell'allocazione dei frame.

### 9.5.2 Algoritmi di allocazione

Il modo più semplice per suddividere  $m$  frame tra  $n$  processi è quello per cui a ciascuno si dà una parte uguale,  $m/n$  frame. Dati 93 frame e cinque processi, ogni processo riceve 18 frame. I tre frame lasciati liberi si potrebbero usare come gruppo dei frame liberi. Questo schema è chiamato **allocazione uniforme**.

Un'alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 KB. Se un piccolo processo utente di 10 KB e una base di dati interattiva di 127 KB sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati.

Per risolvere questo problema è possibile ricorrere all'**allocazione proporzionale**, secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Si supponga che  $s_i$  sia la dimensione della memoria virtuale per il processo  $p_i$ . Si definisce la seguente quantità:

$$S = \sum s_i.$$

Quindi, se il numero totale dei frame disponibili è  $m$ , al processo  $p_i$  si assegnano  $a_i$  frame, dove  $a_i$  è approssimativamente

$$a_i = s_i / S \times m.$$

Naturalmente è necessario scegliere ciascun  $a_i$  in modo che sia un intero maggiore del numero minimo di frame richiesti dalla struttura della serie di istruzioni di macchina e in modo che la somma di tutti gli  $a_i$  non sia maggiore di  $m$ .

Usando l'allocazione proporzionale, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti:

$$\begin{aligned} 10/137 \times 62 &\approx 4 \\ 127/137 \times 62 &\approx 57. \end{aligned}$$

In questo modo entrambi i processi condividono i frame disponibili secondo le rispettive necessità, e non in modo uniforme.

Sia nell'allocazione uniforme sia in quella proporzionale, l'allocazione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo allontanato si possono distribuire tra quelli che restano.

Occorre notare che sia con l'allocazione uniforme sia con l'allocazione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l'uso di uno schema di allocazione proporzionale in cui il rapporto dei frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità.

### 9.5.3 Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: **sostituzione globale** e **sostituzione locale**. La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall'insieme di tutti i frame, anche se quel frame è correntemente allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri ad esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Per un processo si può stabilire una sostituzione che attinga tra i suoi frame oppure tra quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito del processo a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale, invece, può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché per altri non si scelgano per la sostituzione i *propri* frame.

L'algoritmo di sostituzione globale risente di un problema: un processo non può controllare la propria frequenza di assenze di pagine (*page-fault rate*), infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso processo può comportarsi in modi piuttosto diversi, ad esempio impiegando 0,5 secondi per un'esecuzione e 10,3 secondi per quella successiva, a causa di circostanze esterne. Con l'algoritmo di sostituzione locale questo problema non si presenta. Infatti l'insieme di pagine in memoria per un processo subisce l'effetto del comportamento di paginazione di quel solo processo. Dal canto suo, la sostituzione locale può limitare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

### 9.5.4 Accesso non uniforme alla memoria

Fino a questo momento trattando il tema della memoria virtuale abbiamo assunto che le diverse parti della memoria centrale funzionassero tutte allo stesso modo, o almeno che vi si potesse accedere con le stesse modalità. In molti sistemi informatici non è così. Spesso in sistemi con processori multipli (Paragrafo 1.3.2) un certo processore può accedere ad alcune regioni della memoria più rapidamente rispetto ad altre. Tali differenze nelle prestazioni sono causate dalla modalità di interconnessione tra processori e memoria all'interno del sistema. Frequentemente un tale sistema è costituito da diverse schede madri, ognuna contenente più processori e una parte di memoria. Le schede sono connesse in vari modi, a partire dai bus di sistema fino a connessioni di rete ad alta velocità come InfiniBand. Come forse ci si aspetta, i processori di una particolare scheda possono accedere alla memoria della scheda stessa in meno tempo rispetto a quello necessario per accedere ad altre schede del sistema. I sistemi nei quali i tempi di accesso alla memoria variano in modo significativo sono generalmente detti **sistemi con accesso non uniforme alla memoria** (*non-uniform memory access*, NUMA) e, senza eccezioni, sono più lenti dei sistemi nei quali memoria e processori risiedono sulla stessa scheda madre.

Le decisioni su quali frame di pagina memorizzare e dove memorizzarli possono condizionare in modo significativo le prestazioni nei sistemi NUMA. Se, in sistemi del genere,

ignorassimo le diversità nei tempi di accesso alla memoria, i processori potrebbero dover aspettare molto più a lungo per accedere alla memoria rispetto al caso in cui gli algoritmi di allocazione della memoria siano modificati per tenere in conto il NUMA. Analoghe modifiche devono essere apportate anche al sistema di scheduling. L'obiettivo di questi cambiamenti è quello di allocare i frame di memoria “il più vicino possibile” al processore sul quale il processo è in esecuzione, dove per “vicino” si intende “con latenza minima”, ovvero, di solito, sulla stessa scheda della CPU.

I cambiamenti negli algoritmi consistono nel fatto che lo scheduler tiene traccia dell'ultimo processore sul quale ciascun processo è stato eseguito. Se lo scheduler prova a pianificare ciascun processo sul suo processore precedente, e se il sistema di gestione della memoria prova ad allocare frame per il processo vicino al processore sul quale sta per essere mandato in esecuzione, si otterrà un incremento dei successi di cache e una diminuzione del tempo di accesso alla memoria.

La questione diventa ancora più complicata con l'aggiunta dei thread. Ad esempio, un processo con molti thread in esecuzione potrebbe vedere quei thread pianificati su differenti schede del sistema. Come viene allocata la memoria in questo caso? Solaris risolve il problema creando una entità **lgroup** nel kernel. Ogni lgroup raccoglie i processori e la memoria vicini fra loro. In realtà gli lgroup sono ordinati gerarchicamente sulla base del periodo di latenza tra i gruppi. Solaris tenta di pianificare tutti i thread di un processo e di allocare tutta la memoria di un processo nell'ambito di un solo lgroup. Se una tale soluzione non è possibile, per il resto delle risorse necessarie vengono utilizzati gli lgroup più vicini, in modo da minimizzare la latenza complessiva della memoria e massimizzare il grado di successo della cache del processore.

## 9.6 Paginazione degenere (thrashing)

Se il numero dei frame allocati a un processo con priorità bassa diviene inferiore al numero minimo richiesto dall'architettura del calcolatore, occorre sospendere l'esecuzione del processo, e quindi togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello intermedio di scheduling per la gestione dell'entrata e dell'uscita dei processi in memoria centrale.

Infatti, si consideri un qualsiasi processo che non disponga di un numero di frame “sufficiente”. Anche se tecnicamente si può ridurre al valore minimo il numero dei frame allocati, esiste un certo (in generale grande) numero di pagine in uso attivo. Se non dispone di questo numero di frame, il processo accusa immediatamente un'assenza di pagina. A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono in uso attivo, si deve sostituire una pagina che sarà immediatamente necessaria, e di conseguenza si verificano subito parecchie assenze di pagine. Il processo continua a subire assenze di pagine, facendo sostituire pagine che saranno immediatamente trattate come assenti e dovranno essere riprese.

Questa intensa quanto degenere paginazione (nota come *thrashing*) si verifica quando si spende più tempo per la paginazione che per l'esecuzione dei processi.

### 9.6.1 Cause della paginazione degenere

La degenerazione dell'attività di paginazione causa parecchi problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione.

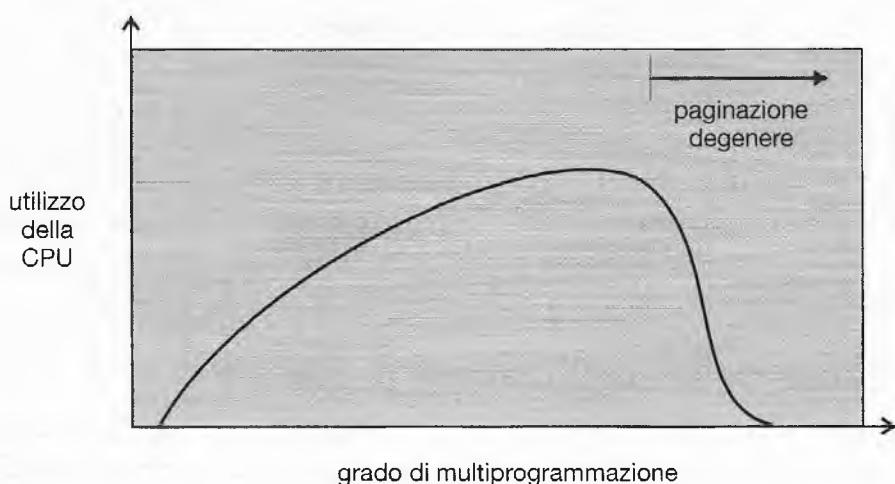
Il sistema operativo vigila sull'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione

delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Per ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di assenze di pagine, cui segue la sottrazione di nuove pagine ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi delle assenze di pagine, con conseguente sottrazione di pagine ad altri processi. Per effettuare il caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce.

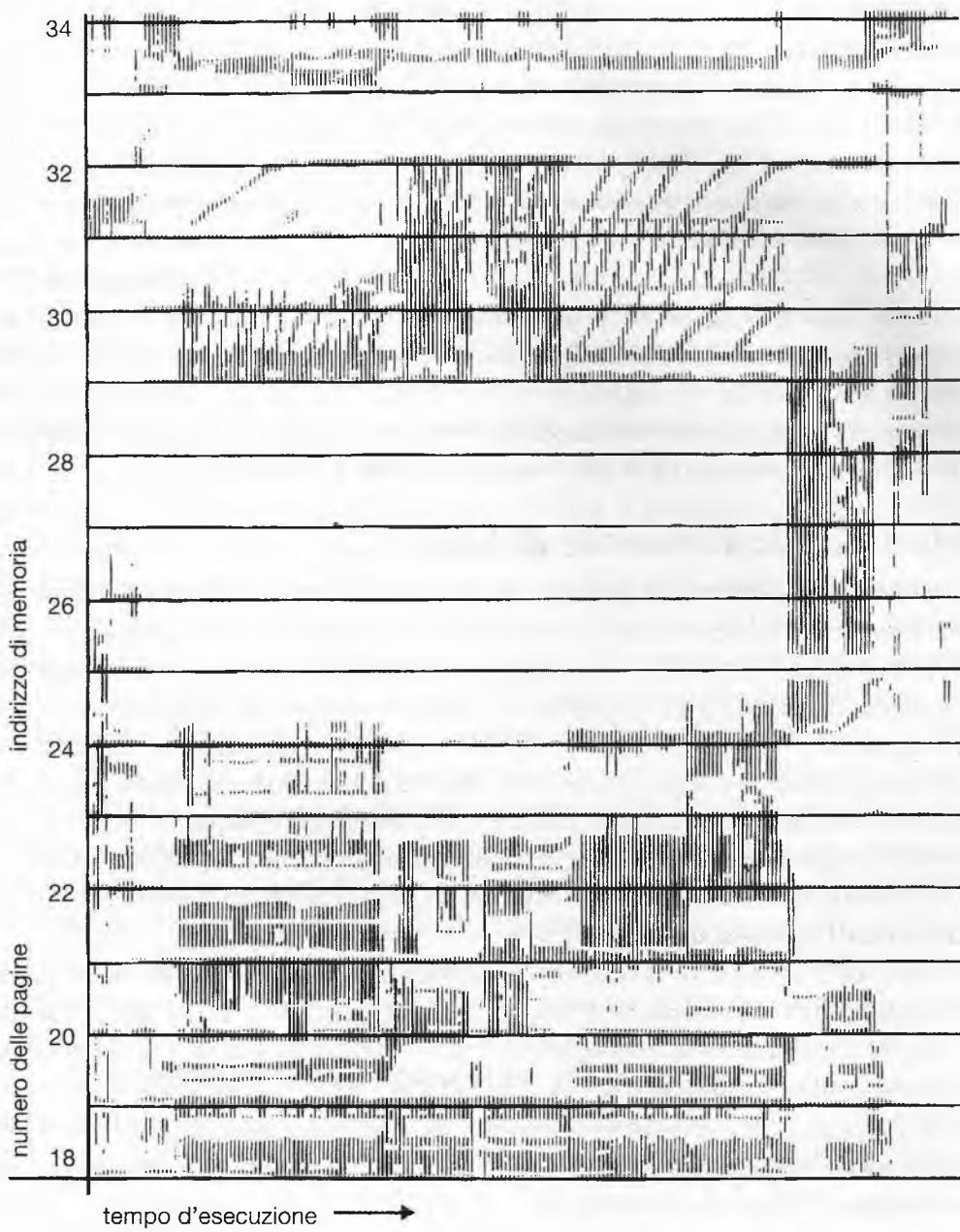
Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e *aumenta* il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori assenze di pagine e allungando la coda per il dispositivo di paginazione. L'utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. L'attività di paginazione è degenerata in una situazione patologica che fa precipitare la produttività del sistema. La frequenza delle assenze di pagine aumenta in modo impressionante, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell'attività di paginazione.

Questo fenomeno è illustrato nella Figura 9.18, in cui si riporta l'utilizzo della CPU in funzione del grado di multiprogrammazione. Aumentando il grado di multiprogrammazione aumenta anche l'utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU occorre *ridurre* il grado di multiprogrammazione.

Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale**, o **algoritmo di sostituzione per priorità**. Con la sostituzione locale, se un processo ricade nell'attività di paginazione degenera, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Le pagine si sostituiscono tenendo conto del processo di cui fanno parte. Tuttavia, se i processi la cui attività di paginazione degenera rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un'eccezione di pagina mancante aumenta a causa dell'allungamento medio della coda d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso al dispositivo di paginazione aumenta anche per gli altri processi.



**Figura 9.18** Paginazione degenera (*thrashing*).



**Figura 9.19** Località dei riferimenti alla memoria.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame “servano” a un processo si impiegano diverse tecniche. Il modello dell’insieme di lavoro (*working-set*), trattato nel Paragrafo 9.6.2, comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo metodo definisce il **modello di località** d’esecuzione del processo.

Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente, com’è illustrato nella Figura 9.19. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili.

Ad esempio, quando s’invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali. Quando la procedura termina, il

processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente. Ritorneremo più avanti al concetto di località.

Quindi, le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all'analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano le assenze delle pagine relative a tali località; quindi, finché le località non vengono modificate, non hanno luogo altre assenze di pagine. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

## 9.6.2 Modello dell'insieme di lavoro

Come già accennato, il modello dell'insieme di lavoro (*working-set model*) è basato sull'ipotesi di località. Questo modello usa un parametro,  $\Delta$ , per definire la finestra dell'insieme di lavoro. L'idea consiste nell'esaminare i più recenti  $\Delta$  riferimenti alle pagine. L'insieme di pagine nei più recenti  $\Delta$  riferimenti è l'*insieme di lavoro*; si veda a questo proposito la Figura 9.20. Se una pagina è in uso attivo si trova nell'insieme di lavoro; se non è più usata esce dall'insieme di lavoro  $\Delta$  unità di tempo dopo il suo ultimo riferimento. Quindi, l'insieme di lavoro non è altro che un'approssimazione della località del programma.

Ad esempio, data la successione di riferimenti alla memoria mostrata nella Figura 9.20, se  $\Delta = 10$  riferimenti alla memoria, l'insieme di lavoro all'istante  $t_1$  è  $\{1, 2, 5, 6, 7\}$ . All'istante  $t_2$  l'insieme di lavoro è diventato  $\{3, 4\}$ .

La precisione dell'insieme di lavoro dipende dalla scelta del valore di  $\Delta$ . Se  $\Delta$  è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se  $\Delta$  è infinito l'insieme di lavoro coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione.

La caratteristica più importante dell'insieme di lavoro è la sua dimensione. Calcolando la dimensione dell'insieme di lavoro,  $WSS_i$ , per ciascun processo  $p_i$  del sistema, si può determinare la richiesta totale di frame, cioè  $D$ :

$$D = \sum WSS_i.$$

Ogni processo usa attivamente le pagine del proprio insieme di lavoro. Quindi, il processo  $i$  necessita di  $WSS_i$  frame. Se la richiesta totale è maggiore del numero totale di frame liberi ( $D > m$ ), la paginazione degenera, poiché alcuni processi non dispongono di un numero sufficiente di frame.

riferimenti alle pagine

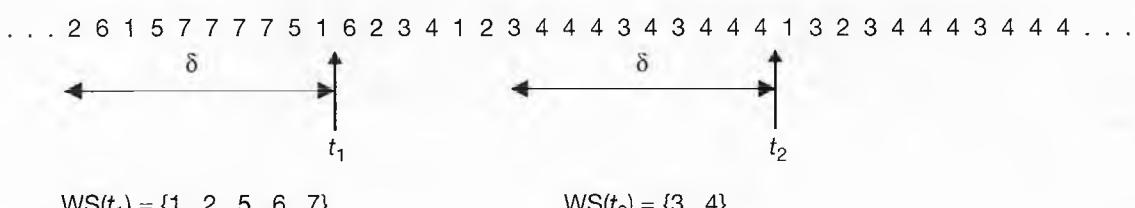


Figura 9.20 Modello dell'insieme di lavoro.

Una volta scelto  $D$ , l'uso del modello dell'insieme di lavoro è abbastanza semplice. Il sistema operativo controlla l'insieme di lavoro di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo insieme di lavoro. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un altro processo. Se la somma delle dimensioni degli insiemi di lavoro aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i propri frame ad altri processi. Il processo sospeso può essere ripreso successivamente.

Questa strategia impedisce la paginazione degenere, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

Poiché la finestra dell'insieme di lavoro è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono l'insieme di lavoro stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nell'insieme di lavoro se esiste un riferimento a essa in qualsiasi punto della finestra dell'insieme di lavoro.

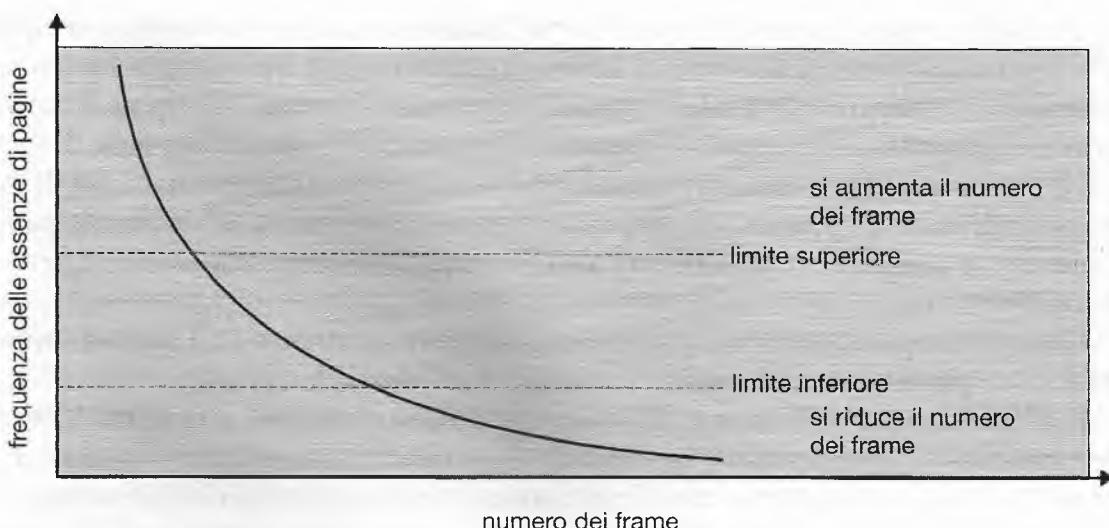
Si supponga, ad esempio, che  $\Delta$  sia pari a 10.000 riferimenti e che sia possibile ottenere un segnale d'interruzione dal timer ogni 5000 riferimenti. Quando si verifica uno di tali segnali d'interruzione, i valori dei bit di riferimento di ciascuna pagina vengono copiati e poi azzerati. Così, quando si verifica un'assenza di pagina è possibile esaminare il bit di riferimento corrente e i 2 bit in memoria per stabilire se una pagina sia stata usata entro gli ultimi 10.000-15.000 riferimenti. Se lo è stata, almeno uno di questi bit è attivo. Se non lo è stata, questi bit sono tutti inattivi. Le pagine con almeno un bit attivo si considerano appartenenti all'insieme di lavoro. Occorre notare che questo schema non è del tutto preciso, poiché non è possibile stabilire dove si è verificato un riferimento entro un intervallo di 5000. L'incertezza si può ridurre aumentando il numero dei bit cronologici e la frequenza dei segnali d'interruzione, ad esempio, 10 bit e un'interruzione ogni 1000 riferimenti. Tuttavia, il costo per servire questi segnali d'interruzione più frequenti aumenta in modo corrispondente.

### 9.6.3 Frequenza delle assenze di pagine

Il modello dell'insieme di lavoro riscuote un discreto successo, e la sua conoscenza può servire per la prepaginazione (Paragrafo 9.9.1), ma appare un modo alquanto goffo per controllare la degenerazione della paginazione. La strategia basata sulla **frequenza delle assenze di pagine** (*page fault frequency*, PFF) è più diretta.

Il problema specifico è la prevenzione della paginazione degenere. La frequenza delle assenze di pagine in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza delle assenze di pagine è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza delle assenze di pagine è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata delle assenze di pagine, com'è illustrato nella Figura 9.21. Se la frequenza effettiva delle assenze di pagine per un processo oltrepassa il limite superiore, occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore, si sottrae un frame a quel processo. Quindi, per prevenire la paginazione degenere, si può misurare e controllare direttamente la frequenza delle assenze di pagine.

Come nel caso dell'insieme di lavoro, può essere necessaria la sospensione di un processo. Se la frequenza delle assenze di pagine aumenta e non ci sono frame disponibili, occorre selezionare un processo e sosponderlo. I frame liberati si distribuiscono ai processi con elevate frequenze di assenze di pagine.



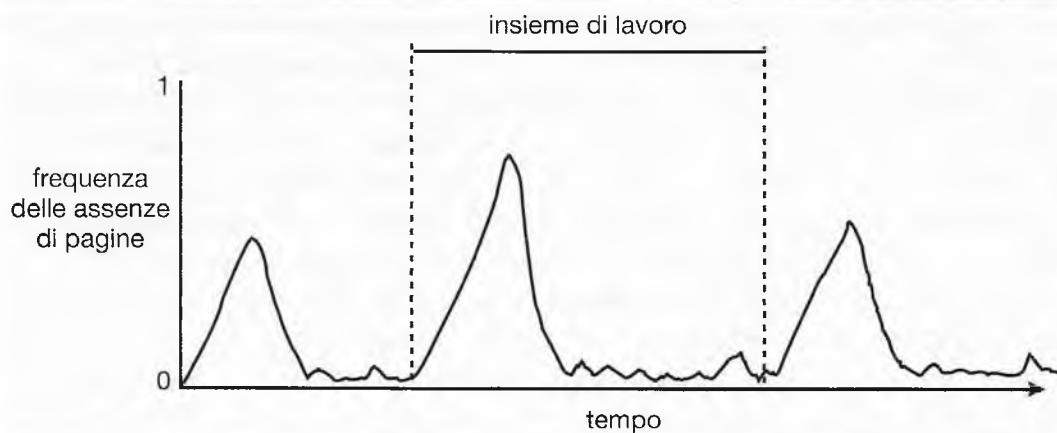
**Figura 9.21** Frequenza delle assenze di pagine.

### INSIEME DI LAVORO E FREQUENZA DELLE PAGINE MANCANTI

Vi è una relazione diretta tra l'insieme di lavoro di un processo e la frequenza di pagine mancanti. Come mostra la Figura 9.20, l'insieme di lavoro di un processo cambia nel corso del tempo, mentre i riferimenti ai dati e alcune parti del codice sono dislocati dall'una all'altra posizione.

Assumendo memoria sufficiente a contenere l'insieme di lavoro di un processo (ossia, a evitare che la paginazione di un processo degeneri), l'andamento delle pagine mancanti oscillerà, in un dato periodo di tempo, tra picchi e valli. Questa tendenza generale è illustrata dalla Figura 9.22.

Un picco nella frequenza di pagine mancanti si verifica alla richiesta di paginazione relativa a una nuova località. Tuttavia, una volta che l'insieme di lavoro interessato sia in memoria, la frequenza di pagine mancanti precipita. Quando il processo entra in un nuovo insieme di lavoro, la frequenza si impenna ancora una volta verso un picco; quando il nuovo insieme di lavoro è caricato in memoria, la frequenza crolla nuovamente. L'intervallo di tempo tra l'inizio di un picco e quello del picco successivo descrive la transizione da un insieme di lavoro a un altro.



**Figura 9.22** Frequenza delle assenze di pagine nel periodo di tempo considerato.

## 9.7 File mappati in memoria

Si consideri la lettura sequenziale di un file sul disco per mezzo delle consuete chiamate di sistema: `open()`, `read()` e `write()`. Ciascun accesso al file richiede una chiamata di sistema e un accesso al disco. In alternativa, possiamo avvalerci delle tecniche di memoria virtuale analizzate sin qui per equiparare l'I/O dei file all'accesso ordinario alla memoria. Grazie a questa soluzione, nota come **mappatura dei file in memoria**, una parte dello spazio degli indirizzi virtuali può essere associata logicamente al file. Come vedremo, ciò comporta un incremento significativo delle prestazioni in fase di I/O.

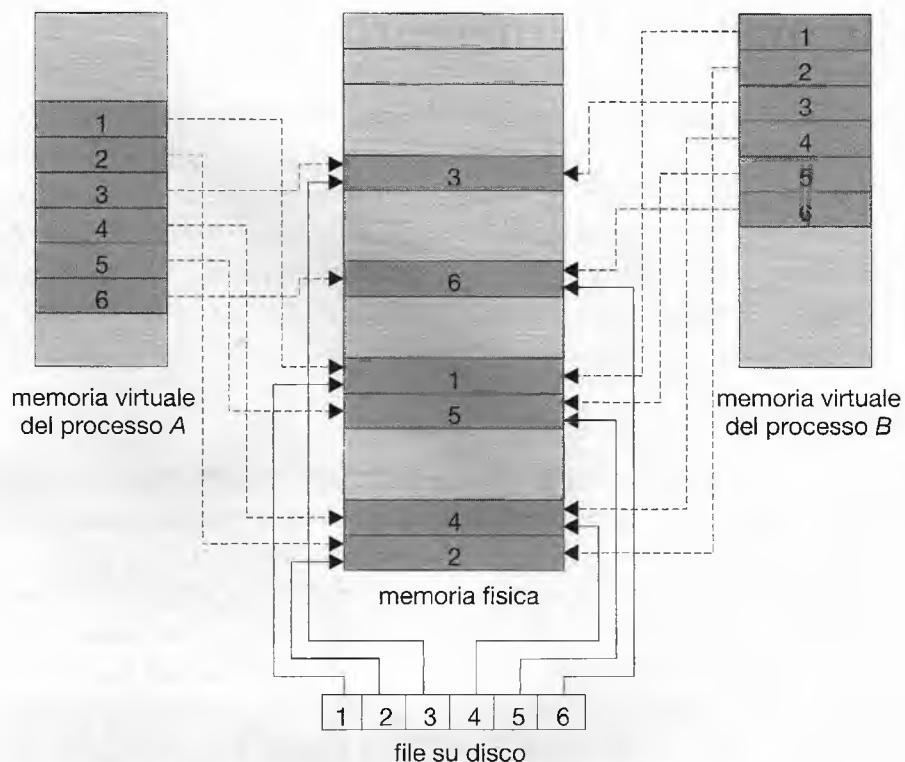
### 9.7.1 Meccanismo di base

La mappatura di un file in memoria si realizza associando un blocco del disco a una o più pagine residenti in memoria. L'accesso iniziale al file avviene tramite una normale richiesta di paginazione, che causa un errore di pagina mancante. Tuttavia, una porzione del file, che è pari a una pagina, è caricata dal file system in una pagina fisica (alcuni sistemi possono decidere di caricare porzioni più grandi di memoria). Ogni successiva lettura e scrittura del file è gestita come accesso ordinario alla memoria, semplificando così l'accesso al file e il suo utilizzo, in quanto si permette al sistema di manipolare i file attraverso la memoria anziché appesantire il sistema stesso con le chiamate `read()` e `write()`. Allo stesso modo, dal momento che il file I/O è creato nella memoria (invece di utilizzare chiamate di sistema che fanno uso di I/O da disco) anche l'accesso al file è molto più veloce.

Si osservi come le scritture sul file mappato in memoria non si traducano necessariamente e immediatamente in scritture sincrone sul file del disco. Alcuni sistemi scelgono di aggiornare il file fisico quando il sistema operativo esegue un controllo periodico per l'accertamento di eventuali modifiche alle pagine. Quando il file viene chiuso, tutti i dati mappati in memoria sono scritti nuovamente su disco e rimossi dalla memoria virtuale del processo.

Alcuni sistemi operativi prevedono un'apposita chiamata di sistema per la mappatura dei dati; le chiamate ordinarie sono riservate a tutte le altre operazioni di I/O. Altri sistemi possono mappare un file in memoria, anche in assenza di un'esplicita richiesta in tal senso. Prendiamo per esempio Solaris. Se si dichiara che un file deve essere mappato in memoria tramite la chiamata di sistema `mmap()`, Solaris opera la mappatura del file nello spazio degli indirizzi del processo. Se si apre un file, e vi si accede con le chiamate di sistema ordinarie, quali `open()`, `read()` e `write()`, Solaris continua a mappare in memoria il file; tuttavia, il file è mappato nello spazio degli indirizzi del kernel. A prescindere da come si apre il file, dunque, Solaris considera tutto l'I/O relativo ai file come mappato in memoria, sfruttando per l'accesso ai file l'efficiente sottosistema della memoria.

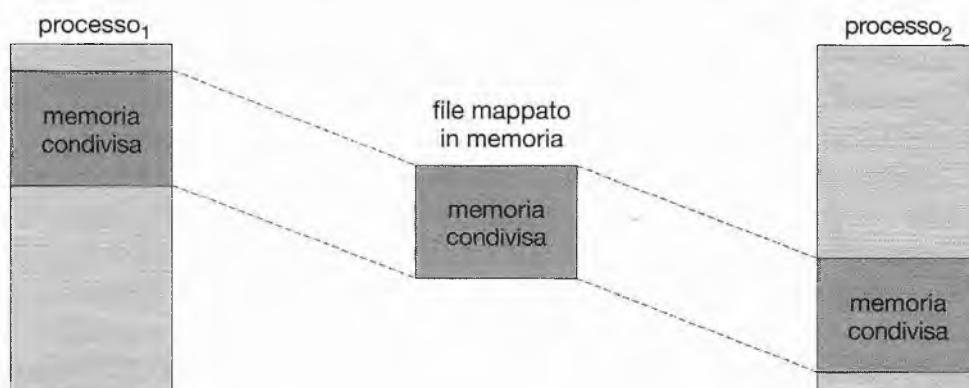
Per consentire la condivisione dei dati, più processi possono essere autorizzati a mappare contemporaneamente un file in memoria. Quanto scritto da uno di questi processi modifica i dati nella memoria virtuale e risulta visibile a tutti gli altri processi che pure mappano il file. L'analisi sulla memoria virtuale svolta fin qui dovrebbe aver chiarito come la condivisione delle sezioni di memoria interessate dalla mappatura abbia luogo: la memoria virtuale di ciascun processo che partecipa alla condivisione punta alla stessa pagina della memoria fisica – la pagina che ospita una copia del blocco del disco. Tale situazione è illustrata nella Figura 9.23. Le chiamate di sistema per la mappatura in memoria possono inoltre offrire la funzionalità della copia su scrittura, che consente ai processi di condividere un file in modalità di sola lettura trattenendo una copia dei dati che modificano. Affinché l'accesso ai



**Figura 9.23** File mappati in memoria.

dati condivisi sia coordinato, i processi interessati potrebbero usare uno dei meccanismi per la mutua esclusione, descritti nel Capitolo 6.

Per molti versi, la condivisione dei file mappati in memoria mostra analogie con la memoria condivisa, trattata nel Paragrafo 3.4.1. Non tutti i sistemi adottano il medesimo meccanismo in entrambi i casi; UNIX e Linux, per esempio, gestiscono la mappatura della memoria con la chiamata di sistema `mmap()`, mentre, per la memoria condivisa, ricorrono alle chiamate `shmget()` e `shmat()` dello standard POSIX (Paragrafo 3.5.1). Nei sistemi Windows NT, 2000 e XP, però, la condivisione della memoria trova concreta applicazione mediante la mappatura dei file in memoria. La comunicazione fra processi si ottiene in questi sistemi mappando in memoria uno stesso file negli spazi degli indirizzi virtuali dei processi coinvolti. Il file mappato in memoria funge da area di memoria condivisa tra i processi comunicanti (Figura 9.24). Nel paragrafo successivo vedremo come la API Win32 fornisca gli strumenti per condividere memoria tramite mappatura dei file in memoria.



**Figura 9.24** Condivisione della memoria in Windows tramite I/O mappato in memoria.

## 9.7.2 Memoria condivisa nella API Win32

La prassi generale per la configurazione di una regione di memoria condivisa nella API Win32 consiste, dapprima, nel **mappare** il file interessato dall'operazione, per poi stabilire una vista (*view*) del file mappato nello spazio degli indirizzi virtuali di un processo. Un secondo processo, a questo punto, può aprire e creare una sua vista del file mappato nel proprio spazio degli indirizzi virtuali. Il file mappato è l'oggetto condiviso tramite il quale può svolgersi la comunicazione tra i processi.

Analizziamo ora queste fasi più da vicino. Nel nostro esempio, il processo produttore crea dapprima un oggetto condiviso, sfruttando le funzionalità per la mappatura in memoria disponibili nella API Win32. Il produttore scrive quindi un messaggio nella memoria condivisa. Il processo consumatore accede a sua volta alla mappatura del file, e legge il messaggio scritto dal produttore.

Per costruire un file mappato in memoria, il processo apre in primo luogo il file da mappare con la funzione `CreateFile()`, che restituisce un riferimento `HANDLE` al file. Il processo allora crea una mappatura di questo riferimento, usando la funzione `CreateFileMapping()`. Una volta stabilita la mappatura del file, il processo genera nel proprio spazio degli indirizzi virtuali una vista del file mappato, con la funzione `MapViewOfFile()`. La vista costituisce la porzione di file che risiederà nello spazio degli indirizzi virtuali del processo; il file può essere mappato solo in parte o per intero. Il programma mostrato nella Figura 9.25 illustra questa procedura (molti controlli sugli errori sono stati omessi per esigenze di brevità del codice).

L'invocazione a `CreateFileMapping()` genera un **oggetto condiviso con nome**, chiamato `SharedObject`. Il processo consumatore utilizzerà questo segmento di memoria condivisa per comunicare, creando una mappatura del medesimo oggetto con nome. Il produttore, quindi, crea nel proprio spazio degli indirizzi virtuali una vista del file mappato in memoria. Passando agli ultimi tre parametri il valore 0, si richiede che la porzione mappata sia l'intero file; si potrebbero tuttavia anche passare valori che specifichino l'inizio e la dimensione della porzione da mappare. (Si osservi, a questo proposito, che non è detto che l'intero file sia caricato in memoria al momento in cui se ne richiede la mappatura: è possibile che il caricamento avvenga tramite paginazione su richiesta, trasferendo perciò di volta in volta le pagine a cui si accede.) La funzione `MapViewOfFile()` restituisce un puntatore all'oggetto condiviso; ogni accesso a questa locazione di memoria è dunque un accesso al file mappato in memoria. In questo caso, nella memoria condivisa il processo produttore scriverà il messaggio **"Messaggio nella memoria condivisa"**.

La Figura 9.26 contiene un programma che dimostra come il processo consumatore stabilisca una vista dell'oggetto condiviso con nome. Il codice è un po' più semplice di quello della Figura 9.25, poiché il processo non ha che da mappare l'oggetto condiviso con nome già esistente. Come il processo produttore, anche il processo consumatore deve creare una vista del file mappato. Il consumatore, quindi, legge dalla memoria condivisa il messaggio scritto dal processo produttore.

Infine, entrambi i processi eliminano la vista del file mappato chiamando `UnmapViewOfFile()`. Alla fine del capitolo il lettore troverà un esercizio di programmazione per la API Win32 incentrato sulla condivisione della memoria tramite mappatura dei file.

```

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

int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", // nome del file
        GENERIC_READ | GENERIC_WRITE, // accesso in lettura/scrittura
        0, // nessuna condivisione del file
        NULL, // sicurezza di default
        OPEN_ALWAYS, // apre il file sia se esistente sia se nuovo
        FILE_ATTRIBUTE_NORMAL, // attributi del file ordinari
        NULL); // niente template del file

    hMapFile = CreateFileMapping(hFile, // riferimento al file
        NULL, // sicurezza di default
        PAGE_READWRITE, // accesso in lettura/scrittura alle pagine mappate
        0, // mappa l'intero file
        0,
        TEXT("OggettoCondiviso")); // oggetto condiviso con nome

    lpMapAddress = MapViewOfFile(hMapFile, // rif. all'oggetto mappato
        FILE_MAP_ALL_ACCESS, // accesso in lettura/scrittura
        0, // vista dell'intero file
        0,
        0);

    // scrive nella memoria condivisa
    sprintf(lpMapAddress, "Messaggio nella memoria condivisa");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);
}

```

**Figura 9.25** Produttore che scrive nella memoria condivisa tramite la API Win32.

### 9.7.3 Mappatura in memoria dell'I/O

Per quanto riguarda l'I/O, come descritto nel Paragrafo 1.2.1, ogni controllore è dotato di registri contenenti istruzioni e i dati in via di trasferimento. Solitamente esistono istruzioni espressamente dedicate all'I/O che consentono il trasferimento dei dati fra questi registri e la memoria del sistema. Le architetture di molti elaboratori, per rendere più agevole l'accesso ai dispositivi dell'I/O, forniscono la **mappatura in memoria dell'I/O**. In questo caso, gli in-

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

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, // accesso R/W
        FALSE, // nessuna ereditarietà
        TEXT("OggettoCondiviso")); // nome dell'oggetto file mappato

    lpMapAddress = MapViewOfFile(hMapFile, // rif. all'oggetto mappato
        FILE_MAP_ALL_ACCESS, // accesso R/W
        0, // vista dell'intero file
        0,
        0);

    // lettura dalla memoria condivisa
    printf("Messaggio letto: %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}
```

**Figura 9.26** Consumatore che legge dalla memoria condivisa tramite la API Win32.

dirizzi di memoria compresi in certi intervalli devono essere riservati alla mappatura dei registri dei dispositivi. Le operazioni in lettura e scrittura su questi indirizzi di memoria fanno sì che i dati siano trasferiti fra i registri dei dispositivi e la memoria, o viceversa. Questo metodo è adatto a dispositivi che abbiano rapidi tempi di risposta, quali i controllori video. Nel PC IBM, a ciascuna posizione sullo schermo corrisponde una locazione mappata in memoria; visualizzare un testo sullo schermo è quasi altrettanto facile che scrivere il testo nelle relative locazioni mappate in memoria.

La mappatura in memoria dell'I/O, inoltre, è conveniente per altri dispositivi, come le porte seriali e parallele usate per connettere i modem e le stampanti all'elaboratore. La CPU invia dati tramite tali dispositivi leggendo e scrivendo alcuni registri all'interno del dispositivo, detti **porte di I/O**. Per inviare una lunga sequenza di byte attraverso una porta seriale mappata in memoria, la CPU scrive un byte nel registro dei dati e imposta un bit nel registro di controllo per indicare che il byte è disponibile. Il dispositivo preleva il byte di dati e cancella il bit del registro di controllo per segnalare che è pronto a ricevere un nuovo byte; a questo punto, la CPU può trasferire il byte successivo. Se la CPU sottopone il bit di controllo a *polling* e cioè esegue costantemente un ciclo per rilevare quando il dispositivo divenga pronto, si parla di **I/O programmato** (*programmed I/O*, PIO). Se la CPU, invece, riceve dal dispositivo un'interruzione non appena è pronto per il byte successivo, si parla di **trasferimento dati guidato dalle interruzioni**.

## 9.8 Allocazione di memoria del kernel

Quando un processo eseguito in modalità utente necessita di memoria aggiuntiva, le pagine sono allocate dalla lista dei frame disponibili che il kernel mantiene. Per formare questa lista, si applica in genere uno degli algoritmi di sostituzione delle pagine esaminati nel Paragrafo 9.4; molto verosimilmente, la lista conterrà pagine non utilizzate sparse per tutta la memoria, come abbiamo visto in precedenza. Va inoltre ricordato che, se un processo utente richiede un solo byte di memoria, si ottiene frammentazione interna, poiché al processo viene garantito un intero frame.

Il kernel, tuttavia, per allocare la propria memoria, attinge spesso a una riserva di memoria libera differente dalla lista usata per soddisfare i processi ordinari in modalità utente. Questo avviene principalmente per due motivi.

1. Il kernel richiede memoria per strutture dati dalle dimensioni variabili; alcune di loro corrispondono a meno di una pagina. Deve quindi fare un uso oculato della memoria, tentando di contenere al minimo gli sprechi dovuti alla frammentazione. Questo fattore è di particolare rilevanza, se si considera che, in molti sistemi operativi, il codice e i dati del kernel non sono soggetti a paginazione.
2. Le pagine allocate ai processi in modalità utente non devono necessariamente essere contigue nella memoria fisica. Alcuni dispositivi, però, interagiscono direttamente con la memoria fisica, senza il vantaggio dell'interfaccia della memoria virtuale; di conseguenza, possono richiedere memoria che risieda in pagine fisicamente contigue.

Nei paragrafi successivi esamineremo due strategie per la gestione della memoria libera assegnata ai processi del kernel: il cosiddetto “sistema buddy” e l’allocazione a lastre.

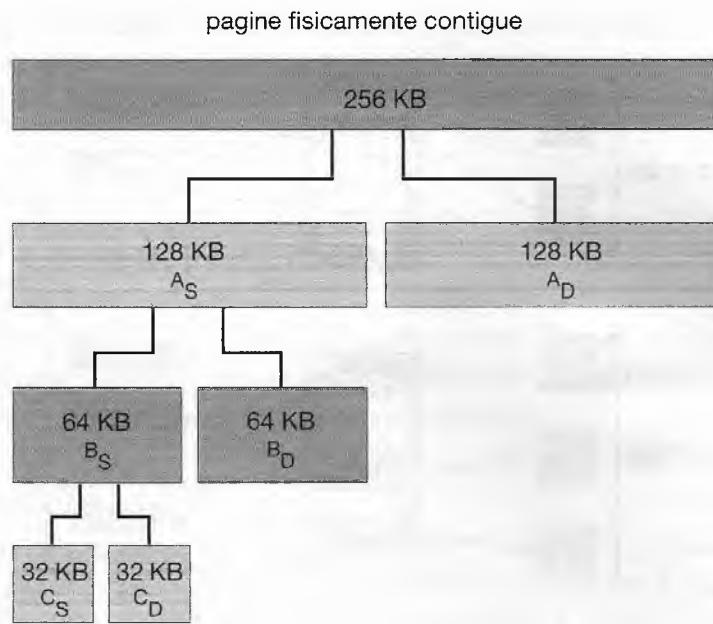
### 9.8.1 Sistema buddy

Il “sistema buddy” (*sistema gemellare*) utilizza un segmento di grandezza fissa per l’allocazione della memoria, contenente pagine fisicamente contigue. La memoria è assegnata mediante un cosiddetto **allocatore-potenza-di-2**, che alloca memoria in unità di dimensioni pari a potenze di 2 (4 KB, 8 KB, 16 KB, e via di seguito). Le differenti quantità richieste sono arrotondate alla successiva potenza di 2. Ad esempio, una richiesta di 11 KB viene soddisfatta con un segmento di 16 KB.

Consideriamo un esempio semplice e ipotizziamo che la grandezza di un segmento di memoria sia inizialmente di 256 KB e che il kernel richieda 21 KB di memoria. In primo luogo il segmento è suddiviso in due *buddy* (“gemelli”), che chiameremo  $A_S$  e  $A_D$ , ciascuno dei quali misura 128 KB. Uno di questi è ulteriormente dimezzato in 2 *buddy* da 64 KB, diciamo  $B_S$  e  $B_D$ . Poiché la minima potenza di 2 che superi 21 KB è pari a 32 KB, occorre suddividere ancora  $B_S$ , oppure  $B_D$ , in due *buddy* la cui dimensione è 32 KB; chiamiamoli  $C_S$  e  $C_D$ . Uno di loro è il segmento scelto per soddisfare la richiesta di 21 KB. Lo schema è illustrati nella Figura 9.27, dove  $C_S$  è il segmento allocato per la richiesta di 21 KB.

Questo sistema offre il vantaggio di poter congiungere rapidamente *buddy* adiacenti per formare segmenti più grandi tramite una tecnica nota come **fusion** (*coalescing*). Nella Figura 9.27, per esempio, quando il kernel rilascia l’unità  $C_S$  che gli era stata allocata, il sistema può fondere  $C_S$  e  $C_D$  in un segmento di 64 KB. Questo segmento,  $B_S$ , può a sua volta fondersi con il proprio gemello  $B_D$ , costituendo un segmento di 128 KB. Con l’ultima operazione di fusione si può ritornare al segmento originale di 256 KB.

L’ovvio inconveniente di questo sistema è che l’arrotondamento per eccesso a una potenza di 2 può facilmente generare frammentazione all’interno dei segmenti allocati. Una ri-



**Figura 9.27** Sistema di allocazione buddy.

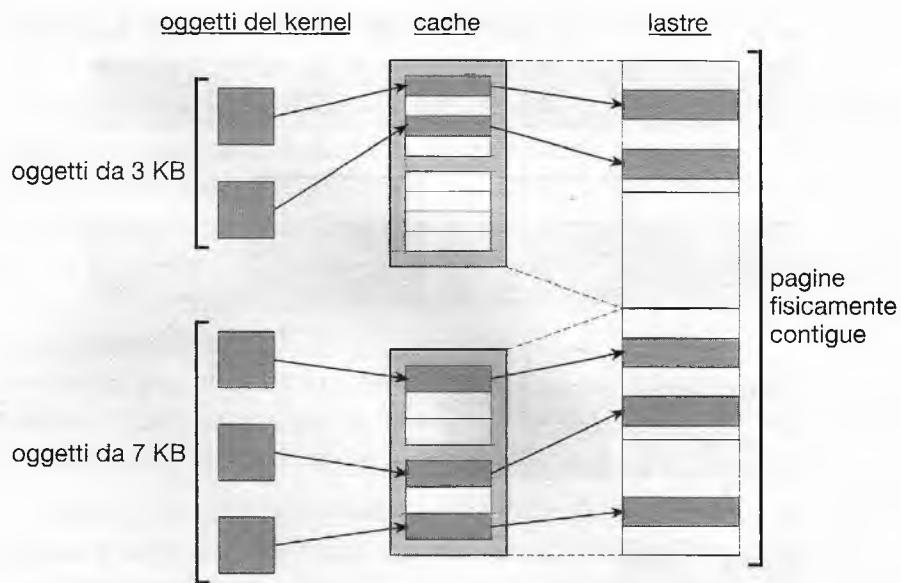
chiesta di 33 KB, per esempio, può essere soddisfatta solo con un segmento di 64 KB. Proprio per effetto della frammentazione interna, dunque, risulta impossibile garantire che lo spreco dell'unità allocata resterà al di sotto del 50%. Nel paragrafo successivo presentiamo una tecnica di allocazione della memoria priva dello spreco dovuto alla frammentazione.

### 9.8.2 Allocazione a lastre

Una seconda strategia per assegnare la memoria del kernel è detta **allocazione a lastre** (*slab allocation*). Una **lastra** è composta da una o più pagine fisicamente contigue. Una **cache** consiste di una o più lastre. Vi è una sola cache per ciascuna categoria di struttura dati del kernel: una cache dedicata alla struttura dati che rappresenta i descrittori dei processi, una dedicata agli oggetti che rappresentano i file, un'altra per i semafori, e così via. Ogni cache è popolata da **oggetti**, istanze della struttura dati del kernel rappresentata dalla cache. La cache che rappresenta i semafori, per esempio, memorizza istanze di oggetti dei semafori; quella che rappresenta i descrittori dei processi memorizza istanze di descrittori dei processi, e così via. La relazione fra lastre, cache e oggetti è illustrata nella Figura 9.28; essa mostra due oggetti del kernel che misurano 3 KB e tre oggetti che misurano 7 KB, memorizzati nelle rispettive cache.

L'algoritmo di allocazione delle lastre utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, un certo numero di oggetti, inizialmente dichiarati liberi, viene assegnato alla cache. Questo numero dipende dalla grandezza della lastra associata alla cache. Per esempio, una lastra di 12 KB (formata da tre pagine contigue di 4 KB) potrebbe contenere sei oggetti di 2 KB ciascuno. Al principio, tutti gli oggetti nella cache sono contrassegnati come **liberi**. Quando una struttura dati del kernel ha bisogno di un oggetto, per soddisfare la richiesta l'allocatore può selezionare dalla cache qualunque oggetto libero; l'oggetto tratto dalla cache è quindi contrassegnato come **usato**.

Consideriamo una situazione in cui il kernel richieda all'allocatore delle lastre la memoria per un oggetto rappresentante un descrittore dei processi. Nei sistemi Linux, un descrittore dei processi ha tipo **struct task\_struct**, che richiede circa 1,7 KB di memo-



**Figura 9.28** Allocazione a lastre (*slab*).

ria. Quando il kernel di Linux crea un nuovo task, richiede alla propria cache la memoria necessaria per l'oggetto di tipo `struct task_struct`. La cache darà corso alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in una lastra e rechi il contrassegno “libero”.

In Linux una lastra può essere in uno dei seguenti stati.

1. **Piena.** Tutti gli oggetti della lastra sono contrassegnati come usati.
2. **Vuota.** Tutti gli oggetti della lastra sono contrassegnati come liberi.
3. **Parzialmente occupata.** La lastra contiene oggetti sia usati sia liberi.

L'allocatore delle lastre, per soddisfare una richiesta, tenta in primo luogo di estrarre un oggetto libero da una lastra parzialmente occupata; se non ne esistono, assegna un oggetto libero da una lastra vuota; in mancanza di lastre vuote disponibili, crea una nuova lastra da pagine fisiche contigue e la alloca a una cache; da tale lastra si attinge la memoria da allocare all'oggetto.

L'allocatore delle lastre offre, essenzialmente, due vantaggi.

1. Annulla lo spreco di memoria derivante da frammentazione. La frammentazione cessa di costituire un problema, poiché ogni struttura dati del kernel ha una cache associata; ciascuna delle cache è composta da un numero variabile di lastre, suddivise in spezzoni di grandezza pari a quella degli oggetti rappresentati. Pertanto, quando il kernel esige memoria per un oggetto, l'allocatore delle lastre restituisce la quantità esatta di memoria necessaria per rappresentare l'oggetto.
2. Le richieste di memoria possono essere soddisfatte rapidamente. La tecnica di allocazione delle lastre si rivela particolarmente efficace quando, nella gestione della memoria, gli oggetti sono frequentemente allocati e deallocati, come spesso accade con le richieste del kernel. In termini di tempo, allocare e deallocare memoria può essere un processo dispendioso. Tuttavia, gli oggetti sono creati in anticipo e possono dunque essere allocati rapidamente dalla cache. Inoltre, quando il kernel rilascia un oggetto di cui non ha più bisogno, questo è dichiarato libero e restituito alla propria cache, rendendosi così immediatamente disponibile ad altre richieste del kernel.

L'allocatore delle lastre ha fatto la sua prima apparizione nel kernel di Solaris 2.4. Per la sua natura generale è ora applicato da Solaris anche ad alcune richieste di memoria in modalità utente. Linux adottava, originariamente, il sistema buddy; tuttavia, a partire dalla versione 2.2, il kernel di Linux include l'allocazione a lastre.

## 9.9 Altre considerazioni

Le due scelte fondamentali nella progettazione dei sistemi di paginazione sono la definizione dell'algoritmo di sostituzione e della politica di allocazione, già analizzate in questo capitolo. Si devono però fare anche molte altre considerazioni.

### 9.9.1 Prepaginazione

Una caratteristica ovvia per un sistema di paginazione su richiesta pura consiste nell'alto numero di assenze di pagine che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La stessa situazione si può presentare anche in altri momenti. Ad esempio, quando si riavvia un processo scaricato nell'area d'avvicendamento, tutte le sue pagine si trovano nel disco e ognuna di loro deve essere reinserita in memoria tramite la gestione di un'eccezione di pagina mancante. La prepaginazione rappresenta un tentativo di prevenire un così alto livello di paginazione iniziale. Alcuni sistemi operativi, e in particolare Solaris, applicano la prepaginazione ai frame dei file di dimensioni modeste.

In un sistema che usa il modello dell'insieme di lavoro, ad esempio, a ogni processo si associa una lista delle pagine contenute nel suo insieme di lavoro. Se occorre sospendere un processo a causa di un'attesa di I/O oppure dell'assenza di frame liberi, si memorizza il suo insieme di lavoro. Al momento di riprendere l'esecuzione del processo (perché l'I/O è terminato o un numero sufficiente di frame liberi è divenuto disponibile), al completamento dell'I/O oppure quando si raggiunge il numero di frame sufficiente, prima di riavviare il processo, si riporta in memoria il suo intero insieme di lavoro.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per l'assistenza delle corrispondenti mancanze di pagina. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepaginate  $s$  pagine e sia effettivamente usata una frazione  $\alpha$  di queste  $s$  pagine ( $0 \leq \alpha \leq 1$ ). Occorre sapere se il costo delle  $\alpha s$  eccezioni di pagina mancanti risparmiate sia maggiore o minore del costo di prepaginazione di  $(1 - \alpha)s$  pagine non necessarie. Se il parametro  $\alpha$  è prossimo allo 0, la prepaginazione non è conveniente; se  $\alpha$  è prossimo a 1, la prepaginazione certamente lo è.

### 9.9.2 Dimensione delle pagine

È raro che chi progetta un sistema operativo per un calcolatore esistente possa scegliere le dimensioni delle pagine. Tuttavia, se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine. Come s'intuisce non esiste un'unica dimensione migliore, ma più fattori sono a sostegno delle diverse dimensioni. Le dimensioni delle pagine sono invariabilmente potenze di 2, in genere comprese tra 4096 ( $2^{12}$ ) e 4.194.304 ( $2^{22}$ ) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la dimensione della tabella delle pagine. Per un dato spazio di memoria virtuale, diminuendo la dimensio-

ne delle pagine aumenta il numero delle stesse e quindi la dimensione della tabella delle pagine. Per una memoria virtuale di 4 MB ( $2^{22}$ ), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano ampie.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione 00000 e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata (le pagine sono unità di allocazione) una parte della pagina finale (frammentazione interna). Supponendo che le dimensioni del processo e delle pagine siano indipendenti è probabile che, in media, metà dell'ultima pagina di ogni processo sia sprecata. Questa perdita è di soli 256 byte per una pagina di 512 byte, ma di 4096 byte per una pagina di 8192 byte. Quindi, per ridurre la frammentazione interna occorrono pagine di piccole dimensioni.

Un altro problema è dovuto al tempo richiesto per leggere o scrivere una pagina. Il tempo di I/O è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che siano preferibili pagine piccole. Come però vedremo nel Paragrafo 12.1.1, il tempo di trasferimento è piccolo se è confrontato con il tempo di latenza e il tempo di posizionamento. A una velocità di trasferimento di 2 MB al secondo, per trasferire 512 byte s'impiegano 0,2 millisecondi. D'altra parte, il tempo di latenza è di circa 8 millisecondi e quello di posizionamento 20 millisecondi. Perciò, del tempo totale di I/O (28,2 millisecondi), l'1 per cento è attribuibile al trasferimento effettivo. Raddoppiando le dimensioni delle pagine, il tempo di I/O aumenta solo fino a 28,4 millisecondi. S'impiegano 28,4 millisecondi per leggere una sola pagina di 1024 byte, ma 56,4 millisecondi per leggere la stessa quantità di byte su due pagine di 512 byte l'una. Quindi, per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori.

Tuttavia, con pagine di piccole dimensioni si riduce l'I/O totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di corrispondere con maggior precisione alla località del programma. Si consideri, ad esempio, un processo di 200 KB, dei quali solo la metà (100 KB) sono effettivamente usati durante l'esecuzione. Se si dispone di una sola ampia pagina, occorre inserirla tutta, sicché vengono trasferiti e assegnati 200 KB. Disponendo di pagine di 1 byte, si possono invece inserire i soli 100 KB effettivamente usati, con trasferimento e allocazione di quei soli 100 KB. Con pagine di piccole dimensioni è possibile avere una migliore **risoluzione**, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre assegnare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole danno come risultato una minore attività di I/O e una minore memoria totale assegnata.

D'altra parte occorre notare che con pagine di 1 byte si verifica un'assenza di pagina per *ciascun* byte. Un processo di 200 KB che usa solo metà di tale memoria accusa una sola assenza di pagina con una pagina di 200 KB, ma 102.400 assenze di pagine con le pagine di 1 byte. Ciascuna assenza di pagina causa un rilevante sovraccarico necessario a elaborare il segnale di eccezione, salvare i registri, sostituire una pagina, mettere in coda nell'attesa del dispositivo di paginazione e aggiornare le tabelle. Per ridurre il numero delle assenze di pagine al minimo sono necessarie pagine di grandi dimensioni.

Occorre considerare altri fattori, come la relazione tra la dimensione delle pagine e quella dei settori del mezzo di paginazione. Non esiste una risposta ottimale al problema considerato. Alcuni fattori (frammentazione interna, località) sono a favore delle piccole dimensioni, mentre altri (dimensione delle tabelle, tempo di I/O) sono a favore delle grandi

dimensioni. La tendenza è storicamente verso l'incremento delle dimensioni delle pagine. Nella prima edizione di questo testo (1983) si considerava un valore di 4096 byte come limite superiore alla dimensione delle pagine. Nel 1990 tale dimensione delle pagine era la più comune. I sistemi moderni possono impiegare pagine di dimensioni assai maggiori, come vedremo nel paragrafo successivo.

### 9.9.3 Portata della TLB

Il tasso di successi (*hit ratio*) di una TLB – si veda in proposito anche il Capitolo 8 – si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dalla TLB anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi della TLB. Tuttavia, la memoria associativa che si usa per costruire le TLB è costosa e consuma molta energia.

Un parametro simile, detto **portata della TLB** (*TLB reach*), esprime la quantità di memoria accessibile dalla TLB, ed è dato semplicemente dal numero di elementi (quindi è correlato al tasso di successi) moltiplicato per la dimensione delle pagine. Idealmente, la TLB dovrebbe contenere l'insieme di lavoro di un processo; altrimenti, la necessità di ricorrere alla tabella delle pagine per la traduzione dei riferimenti alla memoria farà sì che il processo impieghi in quest'operazione assai più tempo di quello richiesto dalla sola TLB. Se si raddoppia il numero di elementi della TLB, se ne raddoppia la portata; per alcune applicazioni che comportano un uso intensivo della memoria ciò potrebbe rivelarsi ancora insufficiente per la memorizzazione dell'insieme di lavoro.

Un altro metodo per aumentare la portata della TLB consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare diverse dimensioni delle pagine. Se si aumenta la dimensione delle pagine, per esempio da 8 KB a 32 KB, la portata della TLB si quadruplica. Quest'aumento potrebbe però condurre a una maggiore frammentazione della memoria relativamente alle applicazioni che non richiedono pagine così grandi. UltraSPARC II è un esempio di architettura che consente diverse dimensioni delle pagine: 8 KB, 64 KB, 512 KB e 4 MB. Di queste possibili dimensioni il sistema operativo Solaris impiega sia quella di 8 KB sia quella di 4 MB. Con una TLB a 64 elementi, la portata della TLB per Solaris varia da 512 KB, con tutte le pagine di 8 KB, a 256 MB, con tutte le pagine di 4 MB. Per la maggior parte delle applicazioni una dimensione delle pagine di 8 KB è sufficiente, sebbene Solaris associa i primi 4 MB del codice e dei dati del kernel a due pagine di 4 MB. Il sistema operativo Solaris permette anche alle applicazioni, come ad esempio i sistemi di gestione delle basi di dati, di trarre vantaggio dalle grandi pagine di 4 MB. Per la maggior parte delle applicazioni è sufficiente la dimensione di 8 KB, sebbene Solaris mappi i primi 4 MB del codice del kernel e dei dati in due pagine da 4 MB. Solaris inoltre consente alle applicazioni – come le basi di dati – di trarre vantaggio dalla dimensione ampia della pagina da 4 MB.

L'uso di diverse dimensioni delle pagine richiede però che la gestione della TLB sia svolta dal sistema operativo e non direttamente dall'architettura. Ad esempio, uno dei campi degli elementi della TLB deve indicare la dimensione della pagina fisica cui il contenuto di ciascun elemento fa riferimento. La gestione della TLB svolta dal sistema operativo e non esclusivamente dall'architettura comporta una penalizzazione delle prestazioni. Tuttavia, i vantaggi dovuti all'aumento del tasso di successi e della portata della TLB superano gli svantaggi che derivano dalla riduzione della rapidità di traduzione degli indirizzi. I recenti sviluppi indicano infatti un'evoluzione verso TLB gestite dal sistema operativo e verso l'uso di pagine di diverse dimensioni. Le architetture UltraSPARC, MIPS, e Alpha adottano TLB che si gestiscono tramite il sistema operativo, mentre le architetture PowerPC e Pentium gestiscono le TLB direttamente, senza l'intervento del sistema operativo.

## 9.9.4 Tabella delle pagine invertita

Nel Paragrafo 8.5.3 si è introdotto il concetto di tabella delle pagine invertita come sistema di gestione delle pagine che consente di ridurre la quantità di memoria fisica necessaria per tener traccia della corrispondenza tra gli indirizzi virtuali e gli indirizzi fisici. Tale riduzione si ottiene tramite una tabella con un elemento per pagina fisica, indicizzato dalla coppia *<id-processo, numero di pagina>*.

Poiché contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, richieste se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di pagine mancanti. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

Contrariamente a quel che potrebbe apparire, l'uso delle tabelle esterne delle pagine non è in contrasto con l'utilità della tabella delle pagine invertita; infatti a loro si fa riferimento solo nel caso di un'assenza di pagina; quindi non è necessario che siano immediatamente disponibili ed esse stesse sono paginate dentro e fuori dalla memoria come è necessario. Sfortunatamente, in questo modo si può verificare un'assenza di qualche pagina dello stesso gestore della memoria virtuale; in tal caso si verifica un'altra assenza di pagina quando esso carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale in memoria ausiliaria (*backing store*). Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e un ritardo nell'elaborazione della ricerca della pagina.

## 9.9.5 Struttura dei programmi

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l'utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono addirittura migliorare se l'utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di  $128 \times 128$  elementi. È tipico il seguente codice:

```

int i, j;
int[128][128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;

```

Occorre notare che l'array è memorizzato per riga, vale a dire che è disposto in memoria secondo l'ordine `data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]`. In pagine di 128 parole, ogni riga occupa una pagina, quindi il frammento di codice precedente azzera una parola per pagina, poi un'altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa  $128 \times 128 = 16.384$  assenze di pagine. D'altra parte, cambiando il codice in

```

int i, j;
int[128][128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;

```

si azzerano tutte le parole di una pagina prima che si inizi la pagina successiva, riducendo a 128 il numero di assenze di pagine.

Un'attenta scelta delle strutture dati e delle strutture di programmazione può aumentare la località e quindi ridurre la frequenza delle assenze di pagine e il numero di pagine dell'insieme di lavoro. Una buona località è quella di una pila, poiché l'accesso avviene sempre alla sua parte superiore. Una tabella hash, invece, è progettata proprio per diffondere i riferimenti, causando una località non buona. Naturalmente, la località dei riferimenti rappresenta soltanto una misura dell'efficienza d'uso di una struttura dati. Altri fattori rilevanti sono rapidità di ricerca, numero totale dei riferimenti alla memoria e numero totale delle pagine coinvolte.

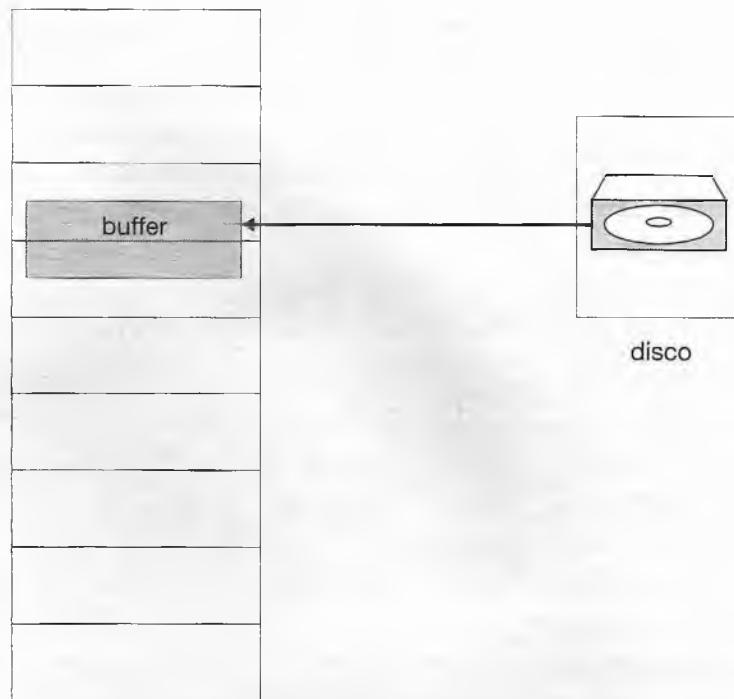
In uno stadio successivo, anche il compilatore e il caricatore possono avere un effetto notevole sulla paginazione. La separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate. Nel sostituire le pagine non modificate, non occorre scriverle in memoria ausiliaria. Il caricatore può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono "impaccare" nella stessa pagina. Questa forma di impaccamento è una variante del problema del *bin-packing* della ricerca operativa: cercare di impaccare i segmenti di dimensione variabile in pagine di dimensione fissa, in modo da ridurre al minimo i riferimenti tra pagine diverse. Un metodo di questo tipo è utile soprattutto per pagine di grandi dimensioni.

Anche la scelta del linguaggio di programmazione può influire sulla paginazione. Con il linguaggio C e il C++, ad esempio, si fa spesso uso dei puntatori, che tendono a distribuire in modo casuale gli accessi alla memoria, diminuendo così la località di un processo. Alcuni studi hanno mostrato che anche i programmi scritti in linguaggi orientati agli oggetti tendono ad avere una scarsa località dei riferimenti.

## 9.9.6 Vincolo di I/O

Quando si usa la paginazione su richiesta, talvolta occorre permettere che alcune pagine si possano **vincolare** alla memoria (*locked in memory*). Una situazione di questo tipo si presenta quando l'I/O si esegue verso o dalla memoria utente (virtuale). Spesso il sistema di I/O comprende una specifica unità d'elaborazione; al controllore di un dispositivo di memorizzazione USB, ad esempio, generalmente si indica il numero di byte da trasferire e un indirizzo di memoria per il buffer (si veda a questo proposito la Figura 9.29). Completato il trasferimento, la CPU riceve un segnale d'interruzione.

Occorre essere certi che non si verifichi la seguente successione di eventi: un processo emette una richiesta di I/O ed è messo in coda per il relativo dispositivo. Nel frattempo si assegna la CPU ad altri processi che accusano assenze di pagine e, usando un algoritmo di sostituzione globale per uno di questi, si sostituisce la pagina contenente l'indirizzo di memoria per l'operazione di I/O attesa dal primo processo; la pagina viene scaricata dalla memoria. Qualche tempo dopo, quando la richiesta di I/O raggiunge la prima posizione della coda d'attesa per il dispositivo, l'operazione di I/O avviene all'indirizzo specificato, ma questo frame è ora impiegato per una pagina appartenente a un altro processo.



**Figura 9.29** Ragione per i cui i frame usati per l'I/O devono essere presenti in memoria.

Questo problema si può risolvere in due modi. Una soluzione prevede di non eseguire operazioni di I/O in memoria utente, ma di copiare i dati sempre tra la memoria di sistema e la memoria utente. In questo modo l'I/O avviene solo tra la memoria di sistema e il dispositivo di I/O. Per scrivere dati in un nastro, occorre prima copiarli in memoria di sistema, quindi trasferirli all'unità a nastro. Tale copia supplementare può causare un sovraccarico inaccettabile.

Un'altra soluzione consiste nel permettere che le pagine siano vincolate alla memoria. A ogni frame si associa un bit di vincolo (*lock bit*); se tale bit è attivato, la pagina contenuta in tale frame non può essere selezionata per la sostituzione. Seguendo questo metodo, per scrivere dati in un nastro occorre vincolare alla memoria le pagine contenenti tali dati, quindi il sistema può continuare come di consueto. Le pagine vincolate non si possono sostituire. Completato l'I/O, si rimuove il vincolo.

I bit di vincolo sono usati in varie situazioni. Spesso il kernel del sistema operativo, o una sua parte, è vincolato alla memoria. La maggior parte dei sistemi non può tollerare l'assenza di una pagina relativa al kernel.

Un altro uso del bit di vincolo riguarda la normale sostituzione di pagine. Si consideri la seguente successione d'eventi: un processo a bassa priorità subisce un'assenza di pagina. Selezionando un frame per la sostituzione, il sistema di paginazione carica in memoria la pagina necessaria. Pronto per continuare, il processo con priorità bassa entra nella coda dei processi pronti per l'esecuzione e attende l'allocazione della CPU. Giacché si tratta di un processo con bassa priorità, può non essere selezionato dallo scheduler della CPU per un certo tempo. Mentre il processo con priorità bassa attende, un processo ad alta priorità accusa un'assenza di pagina. Durante la ricerca per la sostituzione, il sistema di paginazione individua una pagina in memoria alla quale non sono stati fatti riferimenti o modifiche; si tratta della pagina che il processo con bassa priorità ha appena caricato. Questa pagina sembra una sostituzione perfetta: non è stata modificata, non è necessario scriverla in memoria secondaria e, apparentemente, non è stata usata da molto tempo.

Stabilire se la pagina del processo con bassa priorità si debba sostituire a vantaggio del processo con alta priorità è una questione di scelta di un criterio. Semplicemente, si ritarda il processo con bassa priorità a vantaggio di quello con priorità alta. D'altra parte, però, si spreca il lavoro fatto per trasferire in memoria la pagina del processo con bassa priorità. Per evitare che una pagina appena caricata sia sostituita prima che sia usata almeno una volta si può usare il bit di vincolo. Se una pagina è selezionata per la sostituzione, si attiva il suo bit di vincolo: tale bit rimane attivato finché si esegue nuovamente il processo che ha accusato l'assenza di pagina.

Tuttavia, l'uso dei bit di vincolo può essere pericoloso: se un bit non viene mai disattivato, ad esempio a causa di un baco del sistema operativo, il frame relativo alla pagina vincolata diventa inutilizzabile. Su un sistema a singolo utente, l'abuso di tale meccanismo può causare danni soltanto allo stesso utente. Ciò non si può consentire nei sistemi multiutente. Il sistema operativo Solaris, ad esempio, consente l'impiego di "suggerimenti" (*hint*) di vincolo delle pagine, che si possono però trascurare se l'insieme delle pagine libere diviene troppo piccolo o se un singolo processo richiede che troppe pagine siano vincolate alla memoria.

## 9.10 Esempi tra i sistemi operativi

In questo paragrafo si descrive la realizzazione della memoria virtuale nei sistemi operativi Windows XP e Solaris.

### 9.10.1 Windows XP

Il sistema operativo Windows XP realizza la memoria virtuale impiegando la paginazione su richiesta per gruppi di pagine (*demand paging with clustering*). Tale tecnica consiste nel gestire le assenze di pagine caricando in memoria, non solo la pagina richiesta, ma più pagine a essa adiacenti. Alla sua creazione, un processo riceve i valori dell'insieme di lavoro minimo e dell'insieme di lavoro massimo. L'**insieme di lavoro minimo** è il minimo numero di pagine caricate in memoria di un processo che il sistema garantisce di assegnare; se la memoria è sufficiente, però, il sistema potrebbe assegnare un numero di pagine pari al suo **insieme di lavoro massimo**. Per la maggior parte delle applicazioni la misura dell'insieme di lavoro minimo e di quello massimo va, rispettivamente, da 50 a 345 pagine. (In alcuni casi sarebbe anche possibile superare quest'ultimo valore). Il gestore della memoria virtuale mantiene una lista di pagine fisiche libere, con associato un valore di soglia che indica se è disponibile una quantità sufficiente di memoria libera oppure no. Se si verifica un'assenza di pagina per un processo che è sotto il suo insieme di lavoro massimo, il gestore della memoria virtuale assegna una pagina dalla lista delle pagine libere; se invece un processo è già al suo massimo e si verifica un'assenza di pagina, il gestore deve scegliere una pagina da sostituire usando un criterio di sostituzione locale.

Nel caso in cui la quantità di memoria libera scenda sotto la soglia, il gestore della memoria virtuale usa un metodo noto come **regolazione automatica dell'insieme di lavoro** (*automatic working-set trimming*) per riportare il valore sopra la soglia. Si tratta sostanzialmente di valutare il numero di pagine assegnate a ciascun processo; se a un processo sono state assegnate più pagine del suo insieme di lavoro minimo, il gestore della memoria virtuale rimuove pagine fino a raggiungere quel valore; a un processo che è al suo insieme di lavoro minimo, può assegnare altre pagine prendendole dalla lista delle pagine fisiche libere, non appena è disponibile una quantità sufficiente di memoria libera.

L'algoritmo impiegato per stabilire quale pagina rimuovere da un insieme di lavoro dipende dal tipo di unità d'elaborazione disponibile: nei sistemi monoprocessori 80x86, il si-

stema operativo Windows XP usa una variante dell'algoritmo a orologio, presentato nel Paragrafo 9.4.5.2; nei sistemi basati su CPU Alpha e nei sistemi multiprocessore *x86*, l'azzeramento del bit di riferimento può richiedere l'invalidamento dell'elemento corrispondente nella TLB delle altre unità d'elaborazione. Perciò anziché accettare quest'onere, nel sistema Windows XP si usa una variante dell'algoritmo FIFO illustrato nel Paragrafo 9.4.2.

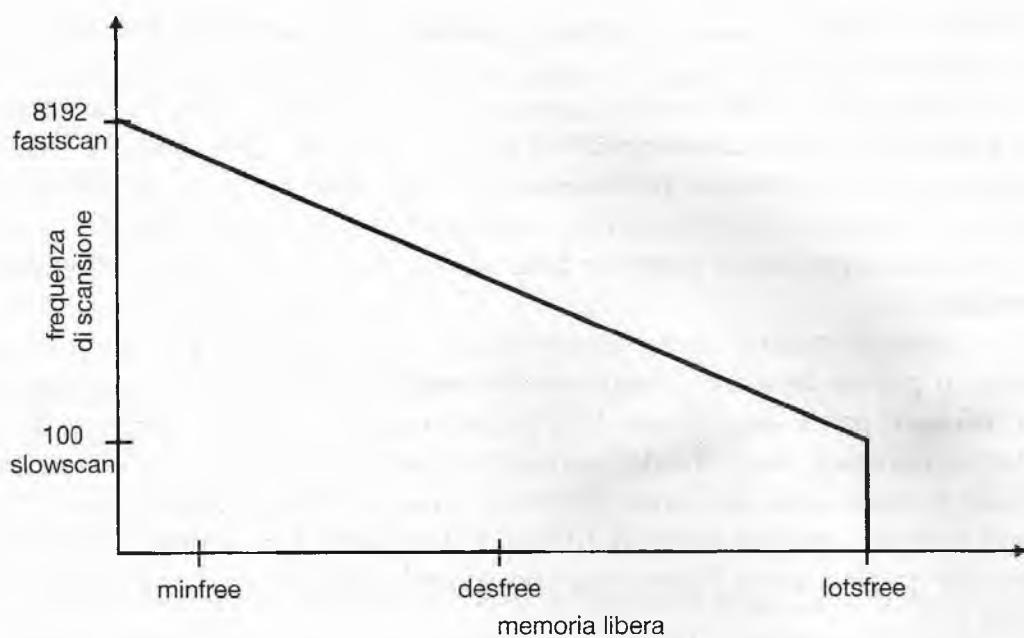
## 9.10.2 Solaris

Il kernel del sistema operativo Solaris assegna una pagina a un thread ogni volta che si verifica un'assenza di pagina, prendendola dalla lista delle pagine libere che il kernel stesso mantiene. È quindi essenziale che il kernel riesca a mantenere una quantità sufficiente di memoria libera. Un parametro, *lotsfree*, associato alla lista delle pagine libere, rappresenta una soglia per l'inizio del processo di paginazione, e il suo è di solito fissato a 1/64 della dimensione della memoria fisica. Il kernel verifica, quattro volte al secondo, se la quantità di memoria libera è inferiore a *lotsfree*. Se il numero di pagine libere scende sotto *lotsfree*, si avvia un processo noto come *pageout*. Questo processo è simile all'algoritmo con seconda chance descritto nel Paragrafo 9.4.5.2, tranne per il fatto che usa due lancette per scorrere le pagine. Il suo funzionamento prevede che la prima lancetta scorra lungo tutte le pagine della memoria, azzerandone il bit di riferimento; in seguito, la seconda lancetta esamina il bit di riferimento delle pagine in memoria, ponendo le pagine con bit nullo in coda alla lista delle pagine libere, e scrivendone i contenuti su disco in caso di modifica. Solaris mantiene una lista cache di pagine liberate, ma non ancora soprascritte. La lista delle pagine libere contiene frame dal contenuto non valido. Le pagine possono essere **richiamate** dalla lista cache nel caso in cui occorra accedervi prima che siano trasferite nella lista delle pagine libere.

Per controllare la frequenza di scansione delle pagine (chiamata anche *scanrate*) l'algoritmo *pageout* si serve di diversi parametri. Questa frequenza è espressa in pagine al secondo ed è compresa tra i valori *slowscan* e *fastscan*. Quando la memoria libera scende sotto *lotsfree*, la scansione delle pagine avviene alla frequenza *slowscan*, e sale fino a *fastscan* secondo la quantità di memoria libera disponibile. Il valore predefinito di *slowscan* è 100, mentre *fastscan* è di solito fissato a (*numero totale delle pagine fisiche*)/2 con un massimo di 8192 pagine al secondo. Questa variazione di frequenza è illustrata nella Figura 9.30 (con *fastscan* fissato al massimo).

La distanza (in pagine) tra le lancette dell'orologio è determinata dal parametro di sistema *handspread*. L'intervallo tra l'azzeramento di un bit da parte della lancetta anteriore e l'esame del suo valore da parte della lancetta posteriore dipende sia da *scanrate* sia da *handspread*. Se il valore di *scanrate* è pari a 100 pagine al secondo e quello di *handspread* è pari a 1024 pagine, possono passare 10 secondi tra la scrittura di un bit della lancetta anteriore e la sua verifica da parte di quella posteriore. Tuttavia, visti i requisiti imposti a un sistema di memoria, non sono rari valori di *scanrate* di diverse migliaia di pagine al secondo. Ciò significa che l'intervallo tra l'azzeramento e il controllo di un bit è spesso di pochi secondi.

Come si è descritto sopra, il processo *pageout* controlla la memoria quattro volte al secondo. Tuttavia, se la memoria libera scende sotto *desfree* (Figura 9.30) *pageout* sarà eseguito 100 volte al secondo con lo scopo di tenere una quantità di memoria libera almeno pari a *desfree*. Se il processo *pageout* non riesce a mantenere al valore *desfree* la quantità media di memoria libera calcolata in un intervallo di 30 secondi, il kernel intraprende l'avvicendamento dei processi, liberando, in questo caso, tutte le pagine assegnate a un processo. In generale, il kernel cerca i processi che sono rimasti inattivi per lunghi periodi. Infine, se il sistema non riesce a mantenere la quantità di memoria libera a *minfree*, invoca il processo *pageout* a ogni richiesta di una nuova pagina.



**Figura 9.30** Scansione delle pagine in Solaris.

Le versioni recenti del kernel di Solaris hanno portato alcuni miglioramenti all'algoritmo di paginazione. Uno è il riconoscimento delle pagine che appartengono a librerie condivise da più processi: anche se sono potenzialmente richiedibili per la scansione, sono ignorate durante il processo d'esame delle pagine. Un altro miglioramento riguarda la capacità di distinguere le pagine allocate ai processi da quelle allocate ai file ordinari. Si tratta del meccanismo di paginazione con priorità descritto nel Paragrafo 11.6.2.

## 9.11 Sommario

È auspicabile poter eseguire processi il cui spazio degli indirizzi logici superi quello disponibile per gli indirizzi fisici. La memoria virtuale è una tecnica che permette di associare grandi spazi degli indirizzi logici a quantità più ridotte di memoria fisica. La memoria virtuale è una tecnica che consente di eseguire processi molto grandi e di aumentare il grado di multiprogrammazione, incrementando l'utilizzo della CPU. Inoltre, grazie a tale tecnica, i programmatore di applicazioni non devono più preoccuparsi della disponibilità di memoria. In più, grazie alla memoria virtuale, processi distinti possono condividere librerie di sistema e memoria. La memoria virtuale consente anche l'utilizzo di un tipo efficiente di creazione di processo conosciuto con il nome di copiatura su scrittura (*copy-on-write*), in cui i processi genitore e figlio condividono pagine effettive di memoria.

La memoria virtuale è comunemente implementata tramite paginazione su richiesta, che trasferisce in memoria una pagina solo quando si incontra un riferimento alla pagina stessa; il primo riferimento produce un errore di pagina. Il kernel del sistema operativo consulta una tabella interna per stabilire la locazione della pagina in memoria ausiliaria, quindi individua un frame libero e vi trasferisce la pagina prelevandola dalla memoria ausiliaria. La tabella delle pagine viene aggiornata per riflettere tale modifica e si riavvia l'istruzione che aveva causato l'eccezione di pagina mancante. Questo metodo permette l'esecuzione di un processo anche se in memoria centrale non è interamente presente la sua immagine di me-

moria. Finché la frequenza delle assenze di pagine rimane ragionevolmente bassa, le prestazioni si considerano accettabili.

La paginazione su richiesta si può usare per ridurre il numero dei frame assegnati a un processo. Questo metodo può aumentare il grado di multiprogrammazione, permettendo che più processi siano disponibili per l'esecuzione in un dato momento e, almeno in teoria, può migliorare l'utilizzo della CPU. Inoltre, consente l'esecuzione di processi i cui requisiti di spazio di memoria superano la memoria fisica disponibile. Tali processi si eseguono in memoria virtuale.

Se i requisiti di spazio di memoria superano la memoria fisica, può essere necessaria la sostituzione di pagine presenti in memoria allo scopo di liberare frame per nuove pagine. Gli algoritmi usati per la sostituzione delle pagine sono diversi: la sostituzione di tipo FIFO è facile da programmare, ma soffre dell'anomalia di Belady; la sostituzione ottimale delle pagine richiede la conoscenza dei futuri riferimenti alla memoria; la sostituzione delle pagine LRU è quasi ottimale, ma può essere di difficile realizzazione. Quasi tutti gli algoritmi di sostituzione delle pagine, come l'algoritmo con seconda chance, sono approssimazioni della sostituzione LRU.

Oltre un algoritmo di sostituzione delle pagine, occorre un criterio di allocazione dei frame. L'allocazione può essere statica, indicando una sostituzione di pagine locale, oppure dinamica, con una sostituzione di pagine globale. Il modello dell'insieme di lavoro presuppone che i processi siano eseguiti in località. L'insieme di lavoro è l'insieme delle pagine nella località corrente. Di conseguenza, a ogni processo si possono allocare frame sufficienti al suo corrente insieme di lavoro. Se un processo non ha spazio di memoria sufficiente per il proprio insieme di lavoro, si ha una paginazione degenera (*thrashing*). Se a ogni processo si devono fornire frame sufficienti per evitare tale degenerazione, sono necessarie le attività d'avvicendamento (*swapping*) e scheduling dei processi.

La maggior parte dei sistemi operativi mette a disposizione degli strumenti per la mappatura in memoria dei file, ciò che permette di trattare l'I/O alla stregua degli accessi in memoria. La API Win32 implementa la condivisione della memoria tramite la mappatura in memoria di file.

Di solito, i processi del kernel richiedono l'allocazione di pagine fisicamente contigue. Il sistema buddy alloca memoria al kernel in segmenti di dimensioni pari a potenze di 2; ciò conduce facilmente a frammentazione. L'allocazione a lastre assegna le strutture dati del kernel a cache associate a lastre, le quali a loro volta sono costituite da una o più pagine fisiche contigue. Questa strategia non produce frammentazione e permette di servire rapidamente le richieste del kernel.

La corretta progettazione dei sistemi di paginazione non solo richiede la soluzione dei due problemi fondamentali della sostituzione delle pagine e dell'allocazione dei frame, ma porta anche a considerare questioni relative a dimensione delle pagine, I/O, gestione dei lock, prepaginazione, generazione dei processi, struttura dei programmi, e altro ancora.

## Esercizi pratici

- 9.1 In quali circostanze si verifica un'eccezione di pagina mancante? Descrivete le azioni che vengono intraprese dal sistema operativo in caso di eccezione di pagina mancante.
- 9.2 Considerate una successione di riferimenti alle pagine di memoria per un processo con  $m$  frame (inizialmente tutti vuoti). La successione ha lunghezza  $p$ ; in essa vi sono  $n$  distinti numeri di pagina. Rispondete alle seguenti domande relative ad algoritmi di sostituzione delle pagine:

- a. Qual è un limite inferiore del numero di pagine mancanti?  
 b. Qual è un limite superiore del numero di pagine mancanti?
- 9.3 Quale delle seguenti tecniche e strutture impiegate nella programmazione si adatta a un ambiente di paginazione su richiesta? Quali invece non sono indicate? Argomentate le vostre risposte:
- pila;
  - tabella hash dei simboli;
  - ricerca sequenziale;
  - ricerca binaria;
  - codice puro;
  - operazioni vettoriali;
  - indirezione (indirection).
- 9.4 Considerate i seguenti algoritmi di sostituzione delle pagine e valutateli basandovi su di una scala a cinque valori da “pessimo” a “ottimo” a seconda della frequenza delle assenze di pagine. Separate gli algoritmi che soffrono dell’anomalia di Belady da quelli che non ne sono affetti:
- sostituzione delle pagine usate meno recentemente (LRU);
  - sostituzione delle pagine secondo l’ordine d’arrivo (FIFO);
  - sostituzione ottimale;
  - sostituzione alla seconda chance.
- 9.5 L’uso della memoria virtuale in un sistema operativo comporta una serie di costi e di benefici: elencateli. È possibile che i costi superino i benefici? In questo caso, quali misure possono essere introdotte per evitare che ciò accada?
- 9.6 Un sistema operativo offre la memoria virtuale paginata, utilizzando un processore centrale con una durata di ciclo di 1 microsecondo. L’accesso a una pagina diversa da quella corrente richiede 1 ulteriore microsecondo. Le pagine hanno 1.000 parole e lo strumento di paginazione è un cilindro che ruota a 3.000 giri al minuto e trasferisce un milione di parole al secondo. Dal sistema si ottengono le seguenti misurazioni statistiche.
- ◆ L’1 per cento di tutte le istruzioni eseguite hanno avuto accesso a una pagina diversa dalla pagina corrente.
  - ◆ L’80 per cento di queste istruzioni – che hanno cioè avuto accesso a un’altra pagina – hanno avuto accesso a una pagina già in memoria.
  - ◆ Nel caso in cui sia stata richiesta una nuova pagina, la pagina sostituita è stata modificata nel 50 per cento dei casi.
- Calcolate il tempo effettivo di esecuzione delle istruzioni di questo sistema, assumendo che il sistema stia eseguendo un unico processo e che il processore sia fermo durante i trasferimenti dal cilindro.
- 9.7 Considerate un array bidimensionale A:

```
int A [ ] [ ] = new int[100][100];
```

dove A[0][0] si trova nella posizione 200 in un sistema di memoria paginato con pagine di dimensione 200. Un piccolo processo che manipola la matrice risiede alla pagina 0 (posizioni da 0 a 199); ogni prelievo di istruzioni partirà così dalla pagina 0.

Per tre frame di pagina, quanti errori per assenza di pagina vengono generati dai seguenti cicli di inizializzazione dell'array, utilizzando la sostituzione LRU e ipotizzando che un frame contenga il processo e gli altri due frame siamo inizialmente vuoti?

- a. 

```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
        A[i][j] = 0;
```
- b. 

```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
        A[i][j] = 0;
```

9.8 Considerate la seguente successione di riferimenti a pagine di memoria:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

Quante eccezioni di pagina mancante si verificherebbero per i seguenti algoritmi di sostituzione, assumendo uno, due, tre, quattro, cinque, sei e sette frame? Ricordate che tutti i frame sono inizialmente vuoti, per cui le vostre prime pagine uniche costeranno un'eccezione ciascuna.

- ◆ Sostituzione secondo LRU.
- ◆ Sostituzione secondo FIFO.
- ◆ Sostituzione ottimale.

9.9 Supponete di voler utilizzare un algoritmo di paginazione che richiede un bit di riferimento (come nella sostituzione alla seconda chance o nel modello dell'insieme di lavoro) che non viene però fornito dall'hardware. Delineate un'ipotesi di simulazione per un bit di riferimento anche se non fornito dall'hardware, oppure spiegate perché non è possibile mettere in pratica una tale ipotesi. Se possibile, calcolate il costo di questo progetto.

9.10 Avete progettato un nuovo algoritmo per la sostituzione delle pagine che pensate possa essere ottimale. In alcuni complicati test di controllo si verifica l'anomalia di Belady. L'algoritmo può essere considerato ottimale? Argomentate la vostra risposta.

9.11 La segmentazione è simile alla paginazione, ma utilizza "pagine" di dimensione variabile. Definite due algoritmi di sostituzione dei segmenti basati sugli schemi di sostituzione delle pagine FIFO e LRU. Ricordate che, siccome i segmenti non hanno la stessa dimensione, il segmento scelto per essere sostituito può essere troppo piccolo per poter contenere abbastanza locazioni di memoria consecutive per il segmento richiesto. Considerate strategie per sistemi nei quali i segmenti non possono essere rilocati e per sistemi nei quali ciò è invece possibile.

9.12 Considerate un sistema informatico a paginazione su richiesta nel quale il grado di multiprogrammazione sia attualmente fissato a quattro. Il sistema è stato recentemente sottoposto a misurazioni volte a determinare l'utilizzo del processore e del disco di paginazione, ottenendo come risultati una delle seguenti alternative. Per ognuno di questi casi, che cosa sta avvenendo? Il livello di multiprogrammazione può essere incrementato per migliorare l'utilizzo del processore? La paginazione può essere utile?

- a. Utilizzo del processore 13 per cento; utilizzo del disco 97 per cento.
- b. Utilizzo del processore 87 per cento; utilizzo del disco 3 per cento.
- c. Utilizzo del processore 13 per cento; utilizzo del disco 3 per cento.

- 9.13 Considerate un sistema operativo per una macchina che utilizza registri base e limite, ma supponete di aver modificato la macchina di modo che metta a disposizione una tabella delle pagine. Si possono configurare le tabelle in modo da simulare registri base e limite? Come? Oppure, perché la cosa non è possibile?

## Esercizi

- 9.14 Supponete che un programma abbia appena fatto riferimento a un indirizzo nella memoria virtuale. Descrivete uno scenario nel quale si verifichi ognuno dei seguenti eventi. (Se non è possibile che si verifichi un tale scenario, spiegatene il motivo.)

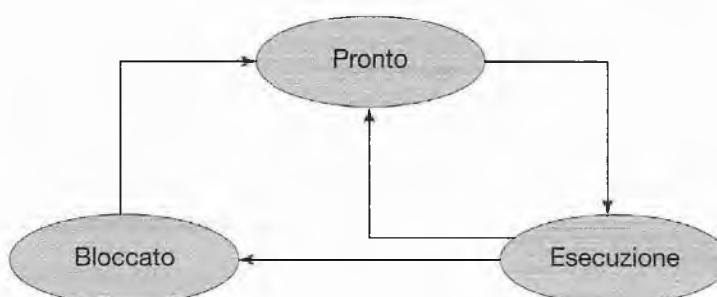
- ◆ Insuccesso della TLB senza assenze di pagina.
- ◆ Insuccesso della TLB con assenze di pagina.
- ◆ Successo della TLB senza assenze di pagina.
- ◆ Successo della TLB con assenze di pagina.

- 9.15 Una visione semplificata degli stati di un thread è *Pronto (Ready)*, *Esecuzione (Running)* e *Bloccato (Blocked)*, dove un thread è pronto e in attesa di essere pianificato, oppure è in esecuzione nel processore, oppure è bloccato (ad esempio è in attesa di I/O), come illustrato nella Figura 9.31. Assumendo che un thread si trovi nello stato di Esecuzione, rispondete alle seguenti domande, argomentando le risposte.

- a. Il thread cambierà di stato se incorrerà in una assenza di pagina? In caso affermativo, quale sarà il nuovo stato?
- b. Il thread cambierà di stato se genererà un insuccesso della TBL che viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?
- c. Il thread cambierà di stato se il riferimento all'indirizzo viene risolto nella tabella delle pagine? In caso affermativo, quale sarà il nuovo stato?

- 9.16 Considerate un sistema che utilizza la paginazione pura su richiesta.

- a. Quando un processo inizia a essere eseguito, come caratterizzereste il tasso di assenze di pagina?
- b. Una volta che l'insieme di lavoro di un processo viene caricato nella memoria, come caratterizzereste il tasso di assenze di pagina?
- c. Supponete che un processo cambi la propria collocazione in memoria e che la dimensione del nuovo insieme di lavoro sia troppo grande per essere caricata nella memoria libera disponibile. Identificate alcune opzioni che i progettisti di sistemi potrebbero scegliere per gestire questa situazione.



**Figura 9.31** Diagramma di stato del thread per l'Esercizio 9.15.

- 9.17 Illustrate con un esempio il problema del riavvio dell'istruzione di rimozione del blocco (MVC) sull'IBM 360/70 quando le regioni di origine e di destinazione si sovrappongono.
- 9.18 Analizzate i dispositivi fisici necessari alla paginazione su richiesta.
- 9.19 Come descrivereste la funzionalità della copiatura su scrittura? A quali condizioni l'uso di tale funzionalità è vantaggioso? Quale dispositivo fisico è richiesto per implementarla?
- 9.20 Un elaboratore fornisce ai propri utenti uno spazio di memoria virtuale di  $2^{32}$  byte. L'elaboratore dispone di  $2^{18}$  byte di memoria fisica. La memoria virtuale è implementata tramite paginazione, e la dimensione delle pagine è di 4096 byte. Un processo utente genera l'indirizzo virtuale 11123456. Spiegate in che modo il sistema determina la corrispondente locazione fisica, distinguendo fra operazioni software e hardware.
- 9.21 Ipotizzate di avere una memoria paginata su richiesta. La tabella delle pagine è conservata in registri. Se un frame vuoto è disponibile o se la pagina sostituita non è modificata, per ovviare alla mancanza di una pagina sono necessari 8 millisecondi, mentre occorrono 20 millisecondi, qualora la pagina sostituita subisca modifiche. Il tempo di accesso alla memoria è pari a 100 nanosecondi.  
Supponete che la pagina da sostituire subisca modifiche in ragione del 70 per cento del tempo. Per un tempo effettivo di accesso non superiore a 200 nanosecondi, qual è la frequenza massima tollerabile di pagine mancanti?
- 9.22 Quando manca una pagina, il processo che ha richiesto la pagina deve bloccarsi mentre aspetta che la pagina venga portata dal disco alla memoria fisica. Posto che esista un processo con cinque thread a livello utente e che la rilocazione dei thread utente ai thread del kernel sia di molti a uno, se un thread utente incorre nell'assenza di pagina quando accede alla sua pila, anche gli altri thread utente appartenenti allo stesso processo sono coinvolti nell'assenza di pagina: devono quindi anch'essi aspettare che la pagina mancante venga portata alla memoria? Motivate la risposta.
- 9.23 Considerate la tabella delle pagine per un sistema con indirizzi virtuali e fisici a 12 bit con pagine di 256 byte. La lista dei frame di pagina liberi è *D, E, F* (dove *D* è in testa alla lista, *E* al secondo posto, ed *F* è in coda).

Pagina	Frame di pagina
0	-
1	2
2	C
3	A
4	-
5	4
6	3
7	-
8	B
9	0

Convertite i seguenti indirizzi virtuali nei corrispondenti indirizzi fisici, in esadecimale. Tutti i numeri dati sono esadecimali. (Il trattino nella colonna dei frame di pagina indica che la pagina non è in memoria.)

- ◆ 9EF
- ◆ 111
- ◆ 700
- ◆ 0FF

- 9.24 Si supponga di monitorare la velocità con cui si muove il puntatore nell'algoritmo a orologio (che indica la pagina candidata per la sostituzione). Che cosa si può inferire sul sistema sapendo che:
- il puntatore si muove velocemente;
  - il puntatore si muove lentamente.
- 9.25 Esaminate a quali condizioni l'algoritmo di sostituzione delle pagine meno frequentemente usate genera un numero inferiore di errori per pagine mancanti rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente. Descrivete anche le circostanze nelle quali è vero il contrario.
- 9.26 Considerate a quali condizioni l'algoritmo di sostituzione delle pagine più frequentemente usate genera un numero inferiore di errori per pagine mancanti rispetto all'algoritmo di sostituzione delle pagine usate meno recentemente. Descrivete anche le circostanze nelle quali è vero il contrario.
- 9.27 Il sistema VAX/VMS utilizza un algoritmo di sostituzione FIFO per le pagine residenti, nonché un gruppo di frame liberi costituito dalle pagine recentemente usate. Per gestire il gruppo di frame, ricorrete al criterio che sostituisce le pagine meno frequentemente usate. Rispondete alle seguenti domande.
- Se viene a mancare una pagina che non si trova nel gruppo di frame, come si genera lo spazio libero per la pagina appena richiesta?
  - Se si verifica la mancanza di una pagina che si trova nel gruppo di frame, come va impostata la pagina residente e in che modo deve essere gestito il gruppo di frame per far spazio alla pagina richiesta?
  - In che cosa degenera il sistema di paginazione se il numero delle pagine residenti è impostato a uno?
  - In che cosa degenera il sistema di paginazione se il numero delle pagine nel gruppo di frame è zero?
- 9.28 Considerate un sistema con paginazione su richiesta con il seguente utilizzo temporale:
- |                          |                |
|--------------------------|----------------|
| utilizzo della CPU       | 20 per cento   |
| disco di paginazione     | 97,7 per cento |
| altri dispositivi di I/O | 5 per cento    |

Indicate, fra le seguenti operazioni, quelle che consentono (o è probabile che consentano) di migliorare l'utilizzo della CPU:

- installazione di una CPU più veloce;
- installazione di un disco di paginazione più grande;
- aumento del grado di multiprogrammazione;

- d. riduzione del grado di multiprogrammazione;
- e. installazione di una maggiore quantità di memoria centrale;
- f. installazione di un disco più veloce o di più controllori di unità con dischi multipli;
- g. aggiunta della prepaginazione agli algoritmi di prelievo delle pagine;
- h. aumento della dimensione delle pagine.

Motivate le risposte.

- 9.29 Supponete che una macchina fornisca istruzioni per l'accesso alle locazioni di memoria attraverso il modello di indirizzamento indiretto a un livello. Quale sequenza di pagine mancanti si riscontra allorché tutte le pagine di un programma siano non residenti e la prima istruzione del programma sia un'operazione di caricamento indiretto dalla memoria? Che cosa succede se il sistema adopera una tecnica di allocazione dei frame per processo e soltanto due pagine siano allocate al processo in questione?
- 9.30 Si supponga che il criterio di sostituzione (in un sistema paginato) consista nel controllo regolare delle pagine, una per volta, eliminando ogni pagina che non sia stata usata dopo l'ultimo controllo. Che cosa offre in più tale criterio, e che cosa in meno, se paragonato all'algoritmo di sostituzione LRU o con seconda chance?
- 9.31 Un algoritmo di sostituzione delle pagine dovrebbe ridurre al minimo il numero delle assenze di pagine. Questa minimizzazione si può ottenere distribuendo in modo uniforme su tutta la memoria le pagine maggiormente usate, anziché lasciarle competere per un piccolo numero di frame. A ogni frame si può associare un contatore del numero delle pagine relative a quel frame. Quindi, per sostituire una pagina, si cerca il frame con il contatore più basso.
- a. Definite un algoritmo di sostituzione delle pagine che si avvalga di questa idea di base. Affrontate in modo specifico i seguenti problemi:
    1. qual è il valore iniziale dei contatori;
    2. quando si incrementano i contatori;
    3. quando si decrementano i contatori;
    4. come si sceglie la pagina da sostituire.
  - b. Se sono disponibili quattro frame, dite quante assenze di pagine avvengono per l'algoritmo indicato, in relazione alla seguente successione di riferimenti:
    - 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.
  - c. Calcolate il numero minimo di assenze di pagine per una strategia di sostituzione delle pagine ottimale per la successione di riferimenti del punto b), con quattro frame.
- 9.32 Considerate un sistema di paginazione su richiesta con un disco di paginazione che abbia un tempo medio d'accesso e di trasferimento di 20 millisecondi. Gli indirizzi sono tradotti per mezzo di una tabella delle pagine che si trova in memoria centrale, con un tempo d'accesso di un microsecondo per ogni accesso alla memoria. Quindi, ogni riferimento alla memoria per mezzo della tabella delle pagine richiede due accessi. Per migliorare questo tempo, è stata aggiunta una memoria associativa che riduce il tempo d'accesso a un riferimento alla memoria, se l'elemento della tabella delle pagine si trova in memoria associativa.

Supponete che, per l'80 per cento degli accessi, l'elemento relativo si trovi in memoria associativa e che il 10 per cento dei restanti (cioè il 2 per cento del totale) causi un'assenza di pagina. Calcolate il tempo effettivo d'accesso alla memoria.

- 9.33 Qual è la causa della paginazione degenere? Come può il sistema accertarla? E, una volta rivelato questo problema, che cosa può fare per eliminarlo?
- 9.34 Chiarite se un processo possa avere due insiemi di lavoro, uno per rappresentare i dati e l'altro per rappresentare il codice.
- 9.35 Considerate il parametro  $\Delta$  usato per definire la finestra dell'insieme di lavoro nell'ambito del modello omonimo. Impostando  $\Delta$  a un valore basso, quale effetto ne deriva per la frequenza degli errori dovuti a pagine mancanti e per il numero di processi attivi (non sospesi) in esecuzione nel sistema? Qual è l'effetto quando  $\Delta$  è impostato a un valore molto alto?
- 9.36 Ipotizzate di avere un segmento iniziale da 1024 KB allocato con il sistema buddy. Seguendo la Figura 9.27 come guida, tracciate l'albero che rappresenta l'allocazione di memoria derivante dalle richieste seguenti:
  - ◆ richiesta di 240 byte;
  - ◆ richiesta di 120 byte;
  - ◆ richiesta di 60 byte;
  - ◆ richiesta di 130 byte.

Modificate adesso l'albero in conformità ai seguenti rilasci di memoria; applicate la fusione ogni qual volta è possibile:

- ◆ rilascio di 240 byte;
- ◆ rilascio di 60 byte;
- ◆ rilascio di 120 byte.

- 9.37 Considerate un sistema in grado di gestire thread sia a livello utente sia a livello kernel. La rilocazione in questo sistema è di uno a uno (a ogni thread del kernel corrisponde un thread utente). Un processo a più thread consiste allora di (a) un insieme di lavoro per l'intero processo, oppure di (b) un insieme di lavoro per ciascun thread?
- 9.38 L'algoritmo di allocazione delle lastre (*slab*) riserva una cache a ciascun oggetto di tipo diverso. Assumendo di avere una cache per tipo di oggetto, spiegate perché il metodo si dimostra scarsamente applicabile a sistemi multiprocessore. Quale potrebbe essere la soluzione a tale problema di scalabilità?
- 9.39 Considerate un sistema che assegna ai propri processi pagine di dimensioni differenti. Quali vantaggi presenta tale schema di paginazione? Quali sono le modifiche da apportare al sistema di memoria virtuale per ottenere questa funzionalità?

## Problemi di programmazione

- 9.40 Scrivete un programma che codifichi gli algoritmi di sostituzione delle pagine FIFO e LRU descritti in questo capitolo. Generate una successione di riferimenti casuale, in cui i numeri delle pagine siano compresi tra 0 e 9. Applicate ciascun algoritmo a tale successione e registrate i rispettivi numeri delle assenze di pagine. Codificate gli algoritmi di sostituzione delle pagine in modo che il numero dei frame sia compreso tra 1 e 7. Supponete l'impiego della paginazione su richiesta.

- 9.41 I numeri di Catalan costituiscono una successione di interi  $C_n$ , che si incontra nei problemi di enumerazione degli alberi. I primi termini della successione, a partire da  $n = 1$ , sono 1, 2, 5, 14, 42, 132, .... Una formula esatta per  $C_n$  è la seguente:

$$C_n = \frac{1}{(n+1)} \left( \frac{2n}{n} \right) = \frac{(2n)!}{(n+1)n!}$$

Progettate due programmi che comunichino condividendo memoria, servendovi della API Win32, come accennato al Paragrafo 9.7.2. Il processo produttore dovrà generare la sequenza di numeri di Catalan e scriverla su un oggetto condiviso. Il processo consumatore, quindi, la leggerà, restituendo in uscita la sequenza dalla memoria condivisa. Al processo produttore sarà passato il parametro intero  $n$  dalla riga di comando; se, per esempio, si passa il valore  $n = 5$ , il processo produttore dovrà generare i primi 5 termini della successione dei numeri di Catalan.

## 9.12 Note bibliografiche

La paginazione su richiesta è stata usata per la prima volta nel sistema operativo Atlas, realizzato per il calcolatore MUSE della Manchester University intorno al 1960 [Kilburn et al. 1961]. Un altro tra i primi sistemi di paginazione su richiesta è stato MULTICS, per il calcolatore GE 645 [Organick 1972].

[Belady et al. 1969] sono stati i primi ricercatori a osservare che la strategia di sostituzione FIFO poteva presentare l'anomalia che porta il nome di Belady. In [Mattson et al. 1970] si dimostra che gli algoritmi a pila non sono soggetti all'anomalia di Belady.

L'algoritmo di sostituzione ottimale è dovuto a [Belady 1966]. In [Mattson et al. 1970] si trova la dimostrazione che esso è ottimale. L'algoritmo ottimale di Belady si usa per l'allocazione statica; [Prieve e Fabry 1976] hanno proposto un algoritmo ottimale per situazioni in cui l'allocazione può variare.

L'algoritmo dell'orologio è trattato in [Carr e Hennessy 1981].

Il modello dell'insieme di lavoro è stato sviluppato da [Denning 1968]. Analisi relative al modello dell'insieme di lavoro sono presenti in [Denning 1980].

Lo schema di controllo della frequenza di assenze di pagine è stato sviluppato da [Wulf 1969], che ha applicato con successo la propria tecnica al calcolatore Burroughs B5500. [Gupta e Franklin 1978] fornisce un confronto delle prestazioni tra lo schema dell'insieme di lavoro e lo schema di sostituzione basato sulla frequenza delle assenze di pagine.

[Wilson et al. 1995] hanno introdotto diversi algoritmi per l'assegnazione dinamica della memoria. Varie problematiche sulla frammentazione della memoria sono esaminate da [Johnstone e Wilson 1998]. Gli allocator di memoria con il sistema buddy sono stati descritti in [Knowlton 1965], [Peterson e Norman 1977], [Purdom Jr. e Stigler 1970]. [Bonwick 1994] ha trattato l'allocator delle lastre e, insieme con Adams [2001] ha esteso la discussione ai processori multipli. Altri algoritmi del genere sono reperibili in [Stephenson 1983], [Bays 1977] e [Brent 1989]. Per una panoramica delle strategie di allocazione della memoria si può consultare [Wilson et al. 1995].

[Solomon e Russinovich 2000] descrivono come il sistema Windows 2000 implementi la memoria virtuale. [Mauro e McDougall 2001] trattano della memoria virtuale di Solaris. Le tecniche di memoria virtuale nei sistemi operativi Linux e BSD UNIX sono state descritte, rispettivamente, in [Bovet e Cesati 2001] e [McKusick et al. 1996]. Le caratteristiche dei sistemi con pagine di dimensioni diverse sono trattate da [Ganapathy e Schimmel 1998], e da [Navarro et al. 2002]. [Ortiz 2001] ha analizzato l'argomento della memoria virtuale integrata nei sistemi operativi in tempo reale.

[Jacob e Mudge 1998b] confrontano le implementazioni dei sistemi di memoria virtuale per le architetture MIPS, PowerPC e Pentium. Un articolo correlato, [Jacob e Mudge 1998a], descrive gli elementi dell'architettura di sistema necessari alla realizzazione della memoria virtuale in sei diversi sistemi, tra i quali l'UltraSPARC.

# Gestione della memoria secondaria

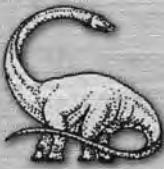
Poiché la memoria centrale è in genere troppo piccola per contenere in modo permanente tutti i dati e tutti i programmi, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale. I calcolatori moderni impiegano i dischi come mezzo principale di registrazione delle informazioni, cioè programmi e dati. Il file system fornisce i meccanismi sia per l'accesso ai dati e ai programmi residenti nei dischi sia per la loro registrazione. Un file è una raccolta d'informazioni tra loro correlate definite dal suo creatore. Il sistema operativo gestisce la corrispondenza tra i file e i dispositivi che li contengono fisicamente; i file sono normalmente organizzati in directory che ne facilitano l'uso.

I dispositivi da collegare a un calcolatore possono variare sotto molti aspetti. Alcuni di loro trasferiscono un carattere o un blocco di caratteri per volta. Alcuni prevedono esclusivamente l'accesso sequenziale, altri solo l'accesso casuale. Talvolta il trasferimento dei dati è sincrono, talaltra è asincrono. Esistono dispositivi dedicati, mentre altri possono essere condivisi. E ancora, a differenza di quelli a sola lettura, alcuni dispositivi ammettono sia la lettura che la scrittura. Pur essendo caratterizzati da notevoli differenze di velocità, i dispositivi rappresentano, nel loro insieme, la componente più lenta e voluminosa dell'elaboratore.

Il sistema operativo, per trattare efficacemente tutte queste varianti, deve offrire alle applicazioni funzionalità che consentano loro un minuzioso controllo dei dispositivi. Uno degli obiettivi cruciali del sottosistema di I/O è fornire la più semplice interfaccia possibile al resto del sistema. Poiché i dispositivi rappresentano un collo di bottiglia per le prestazioni, al fine di sfruttare l'accesso concorrente nel migliore dei modi, l'ottimizzazione dell'I/O è un altro elemento chiave.

## Capitolo 10

# Interfaccia del file system



### OBIETTIVI

- Qual è la funzione dei file system.
- Descrizione delle interfacce dei file system.
- Presentazione dei compromessi di progettazione dei file system, compresi metodi d'accesso, condivisione dei file, uso dei lock e strutture della directory.
- Analisi della protezione dei file system.

Per la maggior parte degli utenti il file system è l'aspetto più visibile di un sistema operativo. Esso fornisce il meccanismo per la registrazione e l'accesso in linea a dati e programmi appartenenti al sistema operativo e a tutti gli utenti del sistema di calcolo. Il file system consiste di due parti distinte: un insieme di *file*, ciascuno dei quali contenente dati correlati, e una *struttura della directory*, che organizza tutti i file nel sistema e fornisce le informazioni relative. I file system – brevemente accennati nel presente capitolo, ma oggetto di approfondimento nei capitoli successivi – basano la loro esistenza su dispositivi. Si analizza inoltre la semantica dei file condivisi da più processi, utenti e calcolatori. In questo capitolo infine si considerano i vari aspetti dei file e i principali tipi di strutture della directory. Si esamina inoltre la gestione della *protezione dei file*, necessaria in un ambiente in cui più utenti hanno accesso ai file, e dove si vuole controllare chi e in che modo vi ha accesso.

## 10.1 Concetto di file

I calcolatori possono memorizzare le informazioni su diversi supporti, come dischi, nastri magnetici e dischi ottici. Per rendere agevole l'uso del calcolatore, il sistema operativo offre una visione logica uniforme della memorizzazione delle informazioni; astrae il *file* dalle caratteristiche fisiche dei propri dispositivi di memoria per definire un'unità di memoria logica. Il sistema operativo associa i file ai dispositivi fisici, di solito non volatili, in modo che il loro contenuto non vada perduto a causa delle interruzioni dell'alimentazione elettrica e dei riavvii del sistema.

Un file è un insieme di informazioni, correlate e registrate in memoria secondaria, cui è stato assegnato un nome. Dal punto di vista dell'utente, un file è la più piccola porzione di memoria secondaria logica; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file. Di solito i file rappresentano programmi, in forma sorgente e oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari, e non possedere

un formato specifico, come i file di testo; oppure essere rigidamente formattati. In genere un file è formato da una sequenza di bit, byte, righe o *record* il cui significato è definito dal creatore e dall'utente del file stesso. Il concetto di file è quindi estremamente generale.

Le informazioni contenute in un file sono definite dal suo creatore e possono essere di molti tipi: programmi sorgente, programmi oggetto, dati numerici, testo, dati contabili, immagini, registrazioni sonore, e così via. Un file ha una struttura definita secondo il tipo: un file di *testo* è formato da una sequenza di caratteri organizzati in righe, e probabilmente pagine; un file *sorgente* è formato da una sequenza di procedure e funzioni, ciascuna delle quali è a sua volta organizzata in dichiarazioni seguite da istruzioni eseguibili; un file *oggetto* è formato da una sequenza di byte, organizzati in blocchi, comprensibile al modulo di collegamento del sistema; un file *eseguibile* consiste di una serie di sezioni di codice che il caricatore può caricare in memoria ed eseguire.

### 10.1.1 Attributi dei file

Per comodità degli utenti, ogni file ha un nome che si usa come riferimento. Un nome, di solito, è una sequenza di caratteri come `esempio.c`. Alcuni sistemi, nella composizione dei nomi, distinguono le lettere maiuscole dalle minuscole, altri le considerano equivalenti. Una volta ricevuto il nome, il file diviene indipendente dal processo, dall'utente, e anche dal sistema da cui è stato creato. Ad esempio, un utente potrebbe creare il file `esempio.c` e un altro utente potrebbe modificarlo specificandone il nome. Il proprietario del file potrebbe registrare il file in un dischetto, inviarlo per posta elettronica, o copiarlo attraverso la rete, ed esso potrebbe ancora chiamarsi `esempio.c` nel sistema di destinazione.

Un file ha altri attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti.

- ◆ **Nome.** Il nome simbolico del file è l'unica informazione in forma umanamente leggibile.
- ◆ **Identificatore.** Si tratta di un'etichetta unica, di solito un numero, che identifica il file all'interno del file system; è il nome impiegato dal sistema per il file.
- ◆ **Tipo.** Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi.
- ◆ **Locazione.** Si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.
- ◆ **Dimensione.** Si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- ◆ **Protezione.** Le informazioni di controllo degli accessi controllano chi può leggere, scrivere o far eseguire il file.
- ◆ **Ora, data e identificazione dell'utente.** Queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione e per controllarne l'uso.

Le informazioni sui file sono conservate nella struttura della directory, che risiede a sua volta in memoria secondaria. Di solito un elemento di directory consiste di un nome di file e di un identificatore unico, che a sua volta individua gli altri attributi del file. Un elemento di directory può richiedere più di un kilobyte per contenere queste informazioni per ciascun file. In un sistema con molti file, la dimensione della stessa directory può essere dell'ordine di megabyte. Poiché le directory, come i file, devono essere non volatili, si devono registrare in memoria secondaria e caricare in memoria centrale un po' per volta, secondo le necessità.

## 10.1.2 Operazioni sui file

Un file è un tipo di dato astratto. Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file. Le successive considerazioni su ciò che deve fare un sistema operativo per ciascuna di queste sei operazioni di base dovrebbero rendere più semplice osservare come si possano realizzare altre operazioni simili, ad esempio la ridenominazione di un file.

- ◆ **Creazione di un file.** Per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system; la discussione sui criteri di allocazione dei file è rimandata al Capitolo 11. Secondariamente, per il file si deve creare un nuovo elemento nella directory in cui registrare il nome del file, la sua posizione nel file system ed eventualmente altre informazioni.
- ◆ **Scrittura di un file.** Per scrivere in un file è indispensabile una chiamata di sistema che specifichi il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema cerca la sua posizione nella directory. Il file system deve mantenere un puntatore di *scrittura* alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogniqualvolta si esegue una scrittura.
- ◆ **Lettura di un file.** Per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il successivo blocco del file. Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un puntatore di *lettura* alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo legge o scrive in un file, e la posizione corrente è mantenuta come un **puntatore alla posizione corrente del file**. Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- ◆ **Riposizionamento in un file.** Si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di I/O. Questa operazione è anche nota come *posizionamento o ricerca (seek)* nel file.
- ◆ **Cancellazione di un file.** Per cancellare un file si cerca l'elemento della directory associato al file designato, si rilascia lo spazio associato al file (in modo che possa essere adoperato per altri) e si elimina l'elemento della directory.
- ◆ **Troncamento di un file.** Si potrebbe voler cancellare il contenuto di un file, ma mantenere i suoi attributi. Invece di forzare gli utenti a cancellare il file e quindi a ricrearlo, questa funzione consente di mantenere immutati gli attributi (a esclusione della lunghezza del file) pur azzerando la lunghezza del file e rilasciando lo spazio occupato.

Queste sei operazioni di base comprendono sicuramente l'insieme minimo delle operazioni richieste per i file. Altre operazioni comuni comprendono l'*aggiunta (Appending)* di nuove informazioni alla fine di un file esistente e la *ridenominazione* di un file esistente. Queste operazioni primitive si possono combinare per compiere altre operazioni. Ad esempio, per creare una *copia* di un file o per copiare il file in un altro dispositivo di I/O, come una stampante o un video, è sufficiente creare un nuovo file, leggere i dati dal file vecchio e scriverli nel nuovo. Sono inoltre necessarie operazioni che consentano a un utente di leggere e impostare i vari attributi di un file.

## APPLICAZIONE DI LOCK NEL LINGUAGGIO JAVA

L'acquisizione del lock di un file tramite la API Java prevede in primo luogo l'acquisizione del `FileChannel` relativo al file; il metodo `lock()` del `FileChannel` permette poi di ottenere il lock. Il prototipo del metodo `lock()` è:

```
FileLock lock(long begin, long end, boolean shared)
```

dove `begin` ed `end` sono il punto iniziale e finale della parte da sottoporre a lock. Se `shared` vale `true`, si ottiene un lock condiviso; altrimenti, un lock esclusivo. Il lock si rilascia tramite il metodo `release()` invocato sull'oggetto restituito da `lock()`.

Il programma della Figura 10.1 illustra questa tecnica. Esso acquisisce due lock del file `file.txt`. La prima metà del file è soggetta a lock esclusivo, la seconda a lock condiviso.

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");
            // acquisisce il canale per il file
            FileChannel ch = raf.getChannel();

            // acquisisce lock esclusivo per la prima metà del file
            exclusiveLock = ch.lock(0, raf.length()/2, EXCLUSIVE);
            /* Modifica i dati . . . */

            // rilascia il lock
            exclusiveLock.release();

            // acquisisce lock condiviso per la seconda metà del file
            sharedLock = ch.lock(raf.length()/2+1, raf.length(), SHARED);
            /* Legge i dati . . . */

            // rilascia il lock
            sharedLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}
```

**Figura 10.1** Esempio di applicazione di lock a un file in Java.

Ad esempio, si potrebbe dover ricorrere a un'operazione che consenta all'utente di determinare lo stato di un file, come la lunghezza, e che consenta di definire gli attributi di un file, come il proprietario.

La maggior parte delle operazioni sopra citate richiede una ricerca dell'elemento associato al file specificato nella directory. Per evitare questa continua ricerca, molti sistemi richiedono l'impiego di una chiamata di sistema `open()` la prima volta che si adopera un file in maniera attiva. Il sistema operativo mantiene una piccola tabella contenente informazioni riguardanti tutti i file aperti (detta, per l'appunto, **tabella dei file aperti**). Quando si richiede un'operazione su un file, questo viene individuato tramite un indice in tale tabella, in questo modo si evita qualsiasi ricerca. Quando il file non è più attivamente usato viene *chiuso* dal processo, e il sistema operativo rimuove l'elemento a esso associato dalla tabella dei file aperti.

Alcuni sistemi aprono implicitamente un file al primo riferimento e lo chiudono automaticamente quando il processo che lo ha aperto termina. A ogni modo, la maggior parte dei sistemi esige che il programmatore richieda l'apertura del file in modo esplicito per mezzo di una chiamata di sistema `open()` prima che sia possibile adoperarlo. L'operazione `open()` riceve il nome del file, lo cerca nella directory e copia l'elemento a esso associato nella tabella dei file aperti. La chiamata di sistema `open()` può accettare anche informazioni sui modi d'accesso: creazione, sola lettura, lettura e scrittura, sola aggiunta, ecc. Si controllano i permessi relativi al file, e se la modalità d'accesso richiesta è consentita, si apre il file. La chiamata di sistema `open()` riporta di solito un puntatore all'elemento nella tabella dei file aperti; questo puntatore si adopera al posto dell'effettivo nome del file in tutte le operazioni di I/O, evitando così successive operazioni di ricerca e semplificando l'interfaccia delle chiamate di sistema.

La realizzazione delle operazioni `open()` e `close()` è più complicata in un ambiente multiutente dove più utenti possono aprire un file contemporaneamente. Di solito il sistema operativo introduce due livelli di tabelle interne: una tabella per ciascun processo e una tabella di sistema. La tabella di un processo contiene i riferimenti a tutti i file aperti da quel processo; il puntatore alla posizione di ciascun file si trova in questa tabella e specifica la posizione nel file. Si possono includere anche i diritti d'accesso al file e informazioni di contabilizzazione.

Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una tabella di sistema dei file aperti, contenente le informazioni indipendenti dai processi come la posizione dei file nei dischi, le date degli accessi e le dimensioni dei file. Quando un file è già stato aperto da un processo, l'apertura di un file da parte di un altro processo comporta l'aggiunta di un elemento relativo al file nella tabella dei file aperti del sistema; una `open()` eseguita da un altro processo comporta solamente l'aggiunta di un nuovo elemento nella tabella dei file aperti associata a quel processo, che punta al corrispondente elemento della tabella di sistema. In genere, la tabella dei file aperti ha anche un *contatore delle aperture* associato a ciascun file, indicante il numero di processi che hanno aperto quel file. Ogni `close()` decremente questo *contatore*; quando raggiunge il valore zero il file non è più in uso e si elimina l'elemento corrispondente dalla tabella dei file aperti. Riassumendo, a ciascun file aperto sono associate le diverse seguenti informazioni.

- ◆ **Puntatore al file.** Nei sistemi che non prevedono lo scostamento come parte delle chiamate di sistema `read()` e `write()`, il sistema deve tener traccia dell'ultima posizione di lettura e scrittura sotto forma di un puntatore alla posizione corrente nel file. Questo puntatore è unico per ogni processo che opera sul file e quindi deve essere tenuto separato dagli attributi del file residenti nel disco.

- ◆ **Contatore dei file aperti.** Man mano che si chiudono i file, per evitare di esaurire lo spazio associato alla propria tabella dei file aperti, il sistema operativo deve riutilizzarne gli elementi. Poiché più processi possono aprire uno stesso file, prima di rimuovere l'elemento corrispondente, il sistema deve attendere l'ultima chiusura del file. Questo contatore tiene traccia del numero di `open()` e `close()`, e raggiunge il valore zero dopo l'ultima chiusura, momento in cui il sistema può rimuovere l'elemento della tabella.
- ◆ **Posizione nel disco del file.** La maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file nel disco è mantenuta in memoria, per evitare di doverla prelevare dal disco a ogni operazione.
- ◆ **Diritti d'accesso.** Ciascun processo apre un file in una delle modalità d'accesso. Questa informazione è contenuta nella tabella del processo in modo che il sistema operativo possa permettere o negare le successive richieste di I/O.

Alcuni sistemi operativi offrono la possibilità di applicare lock a un file aperto (o a parti di esso). Quando un processo intende proteggere un file dall'accesso concorrente di altri processi, si serve dei lock. L'utilità dei lock dei file emerge nel caso di file condivisi da diversi processi: un file di log, per esempio, può subire modifiche da parte di molti processi.

I lock dei file sono basati su una funzionalità simile ai lock di lettura-scrittura (Paragrafo 6.6.2). Un **lock condiviso** è assimilabile, per funzionamento, ai lock di lettura: entrambi consentono a più processi concorrenti di appropriarsene. Un **lock esclusivo** mostra invece analogie con i lock di scrittura, perché un solo processo per volta può acquisire questo tipo di lock. Si noti bene che non in tutti i sistemi operativi è possibile scegliere tra i due tipi di lock; alcuni sistemi forniscono solamente lock esclusivi dei file.

Inoltre, il sistema operativo può fornire meccanismi di protezione dei file **obbligatori**, oppure **consigliati**. Se un lock è obbligatorio, il sistema operativo impedirà a qualunque altro processo di accedere al file interessato una volta che il suo lock sia stato acquisito. Poniamo, per esempio, che un processo si appropri del lock esclusivo del file `system.log`. Se un altro processo – per esempio, un editor – tentasse di aprire `system.log`, il sistema operativo negherebbe l'accesso finché il lock esclusivo ritorni disponibile. Ciò accade anche se l'editor non è esplicitamente programmato per acquisire il lock. Qualora invece il lock sia solo consigliato, il sistema operativo non impedirà l'accesso dell'editor a `system.log`. Tuttavia, per poter accedere al file, l'editor deve essere scritto in modo tale da acquisire esplicitamente il lock. In altri termini, se il lock è obbligatorio, il sistema operativo assicura l'integrità dei dati soggetti a lock; se il lock è solo consigliato, è compito dei programmatore garantire la corretta acquisizione e cessione dei lock. In linea generale, i sistemi operativi Windows adottano i lock obbligatori, mentre i sistemi UNIX impiegano i lock consigliati.

L'uso dei lock dei file richiede l'osservazione delle stesse accortezze per la sincronizzazione dei processi. Per esempio, i programmatore impegnati a sviluppare su sistemi con lock obbligatori devono prestare attenzione a detenere i lock esclusivi solo per l'effettiva durata degli accessi ai file; in caso contrario, bloccheranno anche gli accessi da parte di altri processi. Occorre, inoltre, attuare misure appropriate al fine di evitare che due o più processi entrino in stallo nel tentativo di acquisire i lock per i file.

### 10.1.3 Tipi di file

Nella progettazione di un file system, ma anche dell'intero sistema operativo, si deve sempre considerare la possibilità o meno che quest'ultimo riconosca e gestisca i tipi di file. Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragio-

Tipo di file	Estensione usuale	Funzione
Eseguibile	exe, com, bin, o nessuna	Programma, in linguaggio di macchina, eseguibile
Oggetto	obj, o	Compilato, in linguaggio di macchina, non collegato
Codice sorgente	c, cc, java, pas, asm, a	Codice sorgente in vari linguaggi di programmazione
Batch	bat, sh	Comandi all'interprete dei comandi
Testo	txt, doc	Testi, documenti
Elaboratore di testi	wp, tex, rtf, doc	Vari formati per elaboratori di testi
Libreria	lib, a, so, dll	Librerie di procedure per programmatore
Stampa o visualizzazione	ps, pdf, jpeg	File ASCII o binari in formato per stampa o visione
Archivio	arc, zip, tar	File contenenti più file tra loro correlati, talvolta compressi, per archiviazione o memorizzazione
Multimediali	mpeg, mov, rm, mp3, avi	File binari contenenti informazioni audio o A/V

**Figura 10.2** Comuni tipi di file.

nevole. Ad esempio, un errore abbastanza comune consiste nel tentativo, da parte degli utenti, di stampare un programma oggetto in forma binaria; di solito questo tentativo porta semplicemente a uno spreco di carta, ma si potrebbe impedire se il sistema operativo fosse informato del fatto che il file è un programma oggetto in forma binaria.

Una tecnica comune per realizzare la gestione dei tipi di file consiste nell'includere il tipo nel nome del file. Il nome è suddiviso in due parti, un nome e un'estensione, di solito separate da un punto (Figura 10.2); in questo modo l'utente e il sistema operativo possono risalire al tipo del file semplicemente esaminandone il nome. La maggior parte dei sistemi operativi, per esempio, permette agli utenti di specificare i nomi dei file come sequenze di caratteri seguite da un punto e concluse da un'estensione di caratteri aggiuntivi. Esempi di nomi di file sono *resume.doc*, *Server.java* e *ReaderThread.c*. Il sistema usa l'estensione per stabilire il *tipo* del file e le operazioni che si possono eseguire su tale file. Ad esempio, solamente i file con estensione *.com*, *.exe* o *.bat* sono eseguibili; i file con estensione *.com* e *.exe* sono due formati di file eseguibili, mentre i file con estensione *.bat* sono file (*batch*) contenenti una sequenza di comandi, scritti in formato ASCII, diretti al sistema operativo. L'MS-DOS riconosce un numero limitato di estensioni, che però anche i programmi applicativi possono usare per individuare i tipi di file cui sono interessati: gli assemblatori si aspettano che i file sorgente siano caratterizzati dall'estensione *.asm*; l'elaboratore di testi Microsoft Word presuppone che i propri file terminino con l'estensione *.doc*. Queste estensioni non sono necessarie, ma la loro presenza consente a un utente di ridurre il numero delle battute specificando il nome del file senza estensione e lasciando all'applicazione il compito di cercare il file con il nome impostato e l'estensione attesa. Poiché queste estensioni non sono gestite dal sistema operativo, si possono considerare un suggerimento rivolto alle applicazioni che operano su di loro.

Consideriamo, inoltre, il sistema operativo Mac OS X. In questo sistema ciascun file ha un tipo, come *TEXT* (text file) oppure *APPL* (applicazione). Ciascun file possiede anche un at-

tributo di creazione contenente il nome del programma che lo ha creato. Questo attributo è impostato dal sistema operativo durante la chiamata di sistema `create()`, quindi la sua presenza è forzata e gestita dal sistema operativo. Per esempio, un file prodotto da un elaboratore di testi avrà il nome dell'elaboratore di testi come attributo di creazione. Quando un utente apre il file, con un doppio clic del mouse sull'icona che lo rappresenta, si attiva automaticamente l'elaboratore di testi che apre il file, pronto per essere letto e modificato.

Il sistema operativo UNIX non fornisce una funzione di questo tipo, ma si limita a memorizzare un codice (noto come **magic number**) all'inizio di alcuni tipi di file allo scopo di indicarne in modo generico il tipo: eseguibili, sequenze di comandi (batch file, noti come *shell script*), PostScript e così via. Non tutti i file possiedono tale codice, quindi il sistema non può affidarsi unicamente a questo tipo d'informazione; inoltre, non memorizza il nome del programma che ha creato il file. UNIX consente di sfruttare le estensioni come suggerimento del tipo di file; queste non vengono però imposte né dipendono dal sistema operativo; il loro compito consiste principalmente nell'aiutare gli utenti a riconoscere il tipo di contenuto del file. Un'applicazione può usare o ignorare le estensioni; dipende dalle scelte dei programmati.

### 10.1.4 Struttura dei file

I tipi di file si possono anche adoperare per indicare la struttura interna dei file. Come si è accennato nel Paragrafo 10.1.3, i file sorgente e i file oggetto hanno una struttura corrispondente a ciò che il programma che dovrà leggerli si attende. Inoltre alcuni file devono rispettare una determinata struttura comprensibile al sistema operativo. Ad esempio, il sistema operativo può richiedere che un file eseguibile abbia una struttura specifica che consenta di determinare dove caricare il file in memoria e quale sia la locazione della prima istruzione. Alcuni sistemi operativi estendono questa idea a un insieme di strutture di file gestite dal sistema, con un insieme di operazioni specifiche per la manipolazione dei file con queste strutture. Ad esempio, il sistema operativo DEC VMS è dotato di un file system che gestisce tre strutture di file.

L'analisi precedente porta a considerare uno degli svantaggi dei sistemi operativi che gestiscono più strutture di file: la dimensione risultante del sistema operativo è ingombrante. Se definisce cinque strutture di file differenti, il sistema operativo deve contenere il codice per gestirle tutte; inoltre qualsiasi file potrebbe dover essere definito come uno dei tipi gestiti dal sistema operativo, con la conseguenza di introdurre notevoli problemi per le applicazioni che richiedono una strutturazione dei propri dati in modi non previsti dal sistema operativo.

Ad esempio, si supponga che un sistema operativo preveda due tipi di file: file di testo (composti da caratteri ASCII separati da caratteri di ritorno del carrello e avanzamento di riga) e file binari eseguibili. Un utente che volesse definire un file cifrato per proteggere i propri dati da letture non autorizzate potrebbe scoprire che nessuna delle due strutture si adatta al problema: non è un file di righe di testo ASCII, ma un insieme di bit (apparentemente casuali), e sebbene possa sembrare un file binario, non è eseguibile. Queste limitazioni impongono all'utente di raggirare o usare in modo scorretto il meccanismo dei tipi dei file definito dal sistema operativo, oppure di abbandonare lo schema di codifica.

Alcuni sistemi operativi impongono (e gestiscono) un numero minimo di strutture di file. Questo orientamento è stato seguito da UNIX, dall'MS-DOS e altri. UNIX considera ciascun file come una sequenza di byte, senza alcuna interpretazione. Questo schema garantisce la massima flessibilità, ma il minimo sostegno. Qualsiasi programma applicativo deve contenere il proprio codice per interpretare in modo appropriato la struttura di un file. A ogni modo, per poter caricare ed eseguire i programmi, tutti i sistemi operativi devono prevedere almeno un tipo di struttura, quella dei file eseguibili.

Anche il sistema operativo Macintosh gestisce un numero ridotto di strutture di file. Prevede che i file eseguibili consistano di due parti, **resource fork** e **data fork**. La prima contiene le informazioni che interessano l'utente, ad esempio le etichette dei pulsanti dell'interfaccia del programma. (Per tradurre in un'altra lingua le etichette dei pulsanti, si possono adoperare gli strumenti messi a disposizione dal sistema operativo per la modifica delle informazioni contenute nella prima parte del file.) La seconda contiene il codice del programma o dati: l'usuale contenuto dei file. Per ottenere i medesimi risultati con UNIX o MS-DOS, il programmatore dovrebbe modificare e ricompilare il codice sorgente, a meno che non abbia creato il proprio tipo di file di dati modificabile dall'utente. Evidentemente è utile che un sistema operativo gestisca le strutture che si adoperano spesso, ciò risparmia molto lavoro ai programmatore. Un numero eccessivamente limitato di strutture rende scomoda la programmazione, mentre troppe strutture appesantiscono il sistema operativo e confondono i programmatore.

### 10.1.5 Struttura interna dei file

Per il sistema operativo la localizzazione di uno scostamento all'interno di un file può essere complicata. I dischi hanno una dimensione dei blocchi ben definita, stabilita secondo la dimensione di un settore. Tutti gli I/O su disco si eseguono in unità di un blocco (record fisico), e tutti i blocchi hanno la stessa dimensione. È improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico desiderato, che può anche essere variabile. Una soluzione diffusa per questo tipo di problema consiste nell'**impaccamento** di un certo numero di record logici in blocchi fisici.

Il sistema operativo UNIX, ad esempio, definisce tutti i file semplicemente come un flusso di byte. A ciascun byte si può accedere in modo individuale tramite il suo scostamento a partire dall'inizio, o dalla fine, del file. In questo caso il record logico è un byte. Il file system impacca e deimpacca automaticamente i byte in blocchi fisici (ad esempio 512 byte per blocco) come è necessario.

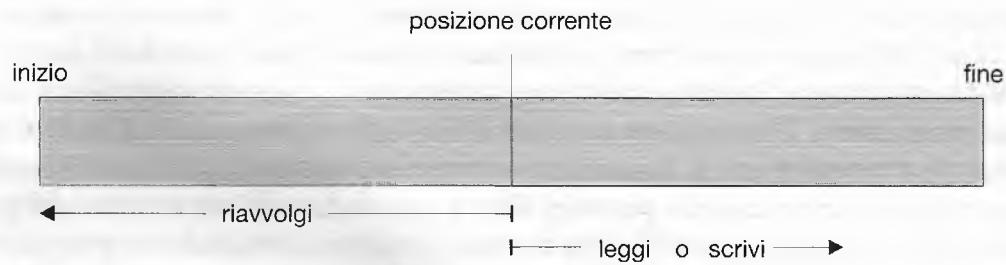
La dimensione dei record logici, quella dei blocchi fisici e la tecnica d'impaccamento determinano il numero dei record logici all'interno di ogni blocco fisico. L'impaccamento può essere fatto dal programma applicativo dell'utente oppure dal sistema operativo.

In entrambi i casi il file si può considerare come una sequenza di blocchi. Tutte le funzioni di I/O di base operano in termini di blocchi. La conversione da record logici a blocchi fisici è un problema di programmazione relativamente semplice.

Poiché lo spazio del disco è sempre assegnato in blocchi, una parte dell'ultimo blocco di ogni file in genere è sprecata. Se ogni blocco è composto di 512 byte, a un file di 1949 byte si assegnano quattro blocchi (2048 byte); gli ultimi 99 byte sono sprecati. I byte sprecati, assegnati per la gestione in multipli di blocchi invece che di byte, costituiscono la **frammentazione interna**. Tutti i file system ne soffrono; maggiore è la dimensione dei blocchi, maggiore sarà la frammentazione interna.

## 10.2 Metodi d'accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria. Esistono molti metodi per accedere alle informazioni dei file; alcuni sistemi consentono un solo metodo d'accesso ai file; altri, come quelli IBM, offrono diversi metodi d'accesso; la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.



**Figura 10.3** File ad accesso sequenziale.

### 10.2.1 Accesso sequenziale

Il più semplice metodo d'accesso è l'**accesso sequenziale**: le informazioni del file si elaborano ordinatamente, un record dopo l'altro; questo metodo d'accesso è di gran lunga il più comune, ed è usato, ad esempio, dagli editor e dai compilatori.

Le più comuni operazioni che si compiono sui file sono le letture e le scritture: un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. Un file siffatto si può reimpostare sull'inizio e, in alcuni sistemi, un programma può riuscire ad andare avanti o indietro di  $n$  record, con  $n$  intero e alcune volte solo per  $n = 1$ . L'accesso sequenziale è illustrato nella Figura 10.3. L'accesso sequenziale è basato su un modello di file che si rifa al nastro, e funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.

### 10.2.2 Accesso diretto

Un altro metodo è l'**accesso diretto** (o **accesso relativo**). Un file è formato da elementi logici (**record**) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifa al disco: i dischi permettono, infatti, l'accesso diretto a ogni blocco di file. Il file si considera come una sequenza numerata di blocchi o record che si possono leggere o scrivere in modo arbitrario: si può ad esempio leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. Non esistono limiti all'ordine di lettura o scrittura di un file ad accesso diretto.

I file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre stabilire quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste.

In un sistema di prenotazione di volo, ad esempio, si possono registrare tutte le informazioni su un particolare volo, ad esempio il volo 713, nel blocco identificato da tale numero di volo. Quindi, il numero di posti disponibili per il volo 713 si memorizza nel blocco 713 del file di prenotazione. Per registrare informazioni riguardanti un gruppo più grande, ad esempio una popolazione, si può eseguire la ricerca calcolando una funzione hash sui nomi delle persone, oppure usando un piccolo indice per determinare il blocco da leggere.

Per il metodo ad accesso diretto, si devono modificare le operazioni sui file per inserire il numero del blocco in forma di parametro. Quindi, si hanno `read n`, dove  $n$  è il numero del blocco, al posto di `read next`, e `write n`, invece che `write next`. Un metodo alternativo prevede di mantenere `read next` e `write next`, come nell'accesso sequen-

ziale, e di aggiungere un'operazione `position to n`, dove  $n$  è il numero del blocco. Quindi a un'operazione `read n` corrispondono una `position to n` e una `read next`.

Il numero del blocco fornito dall'utente al sistema operativo è normalmente un **numero di blocco relativo**. Si tratta di un indice relativo all'inizio del file, quindi il primo blocco relativo del file è 0, il successivo è 1 e così via, anche se l'indirizzo assoluto nel disco del blocco può essere 14703 per il primo blocco e 3192 per il secondo. L'uso dei numeri di blocco relativi permette al sistema operativo di decidere dove posizionare il file (si tratta del *problema dell'allocazione* trattato nel Capitolo 11) e aiuta a impedire che l'utente acceda a porzioni del file system che possono non far parte del suo file. Alcuni sistemi iniziano la numerazione dei blocchi relativi da 0, altri da 1.

Come soddisfa il sistema operativo una richiesta di un record  $n$  in un file? Dato un record logico di lunghezza  $l$ , una richiesta per il record  $n$  determina una richiesta di I/O per  $l$  byte alla locazione  $l \times n$  all'interno del file (assumendo che il primo record sia  $n = 0$ ). Lettura, scrittura e cancellazione di un record sono rese più semplici dalla sua dimensione fissa.

Non tutti i sistemi operativi gestiscono ambedue i tipi di accesso; alcuni permettono il solo accesso sequenziale, altri solo quello diretto. Alcuni sistemi richiedono che si definisca il tipo d'accesso al file al momento della sua creazione; a tale file si può accedere soltanto nel modo definito. Tuttavia si può facilmente simulare l'accesso sequenziale a un file ad accesso diretto mantenendo una variabile `cp` che, come illustra la Figura 10.4, definisce la posizione corrente. D'altra parte è estremamente goffo e inefficiente simulare l'accesso diretto a un file che di per sé è ad accesso sequenziale.

### 10.2.3 Altri metodi d'accesso

Sulla base di un metodo d'accesso diretto se ne possono costruire altri, che implicano generalmente la costruzione di un indice per il file. L'indice contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice, e quindi usare il puntatore per accedere direttamente al file e trovare l'elemento desiderato.

Si consideri, ad esempio, un file contenente prezzi al dettaglio, contenente una lista dei codici universali dei prodotti (*universal product codes*, UPC), a ciascuno dei quali è associato un prezzo. Dato un elemento di 16 byte, questo è composto da un codice UPC a 10 cifre e un prezzo a 6 cifre. Se il disco usato ha 1024 byte per blocco, in ogni blocco si possono memorizzare 64 elementi. Un file di 120.000 elementi occupa circa 2000 blocchi (2 milioni di byte). Ordinando il file secondo il codice UPC si può definire un indice composto dal primo codice UPC di ogni blocco. Tale indice è costituito di 2000 elementi di 10 cifre ciascuno (20.000 byte) e quindi può essere tenuto in memoria. Per trovare il prezzo di un oggetto specifico si può fare una ricerca (binaria) nell'indice, che permette di sapere esattamente quale

Accesso sequenziale	Realizzazione nel caso di accesso diretto
<code>reset</code>	<code>cp = 0;</code>
<code>read next</code>	<code>read cp;</code> <code>cp = cp + 1;</code>
<code>write next</code>	<code>write cp;</code> <code>cp = cp + 1;</code>

**Figura 10.4** Simulazione dell'accesso sequenziale a un file ad accesso diretto.

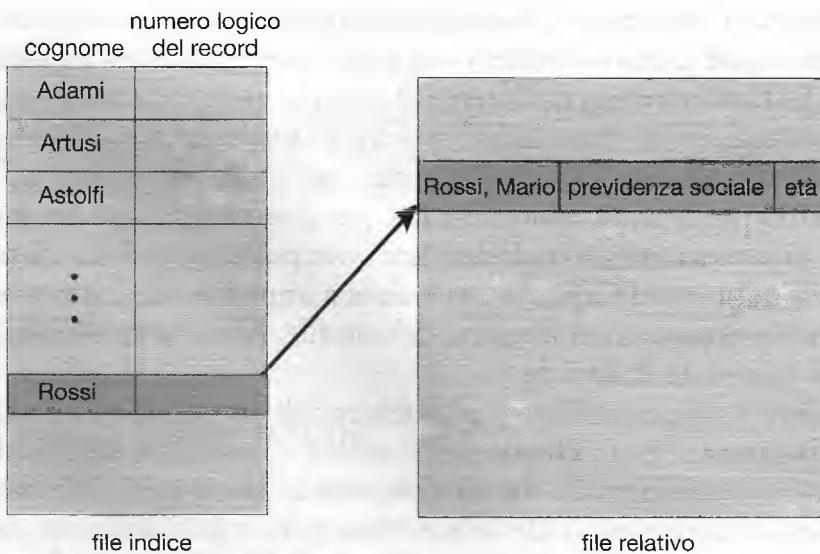


Figura 10.5 Esempio di indice e relativo file.

blocco contiene l'elemento desiderato e quindi accedere a quel blocco. Questa struttura permette di compiere ricerche in file molto lunghi limitando il numero di operazioni di I/O.

Nel caso di file molto lunghi, lo stesso file indice può diventare troppo lungo perché sia tenuto in memoria. Una soluzione a questo problema è data dalla creazione di un indice per il file indice. Il file indice principale contiene puntatori ai file indice secondari, che puntano agli effettivi elementi di dati.

Il metodo ad accesso sequenziale indicizzato di IBM (*indexed sequential access method*, ISAM), ad esempio, usa un piccolo indice principale che punta ai blocchi del disco di un indice secondario, e i blocchi dell'indice secondario puntano ai blocchi del file effettivo. Il file è ordinato rispetto a una chiave definita. Per trovare un particolare elemento, si fa inizialmente una ricerca binaria nell'indice principale, che fornisce il numero del blocco dell'indice secondario. Questo blocco viene letto e sottoposto a una seconda ricerca binaria che individui il blocco contenente l'elemento richiesto. Infine, si fa una ricerca sequenziale sul blocco. In questo modo si può localizzare ogni elemento tramite il suo codice con al massimo due letture ad accesso diretto. La Figura 10.5 mostra uno schema simile, com'è realizzato nel VMS con indici e relativi file.

### 10.3 Struttura della directory e del disco

Si prenderanno ora in considerazione le modalità di memorizzazione dei file. Ovviamente, nessun computer a uso generale contiene un unico file. Vi sono infatti memorizzati spesso migliaia, milioni o persino bilioni di file. I file vengono salvati in dispositivi di memorizzazione ad accesso casuale, come dischi fissi, dischi ottici e dischi a stato solido (basati sulla memoria).

Un dispositivo di memorizzazione può essere interamente utilizzato per un file system, ma può anche essere suddiviso per un controllo più raffinato. Un disco può ad esempio essere **partizionato** e ogni partizione può contenere un file system. I dispositivi di memorizzazione possono essere raccolti in insiemi RAID che proteggono dal fallimento di un singolo disco (come descritto al Paragrafo 12.7). Alcune volte, i dischi sono suddivisi e al contempo raccolti in insiemi RAID.

La suddivisione in partizioni è utile anche per limitare la dimensione dei file system individuali, per mettere sullo stesso dispositivo tipi diversi di file system, oppure per liberare ad altri scopi una parte del dispositivo, come nel caso dello spazio di swap (*avvicendamento*) o dello spazio su disco non formattato (*raw*). Le partizioni sono note anche come **suddivisioni** (*slice*) o **minidischi** (nel mondo IBM). Un file system può essere installato su ciascuna di queste parti del disco. Ogni entità contenente un file system è generalmente nota come **volume**. Il volume può essere un sottoinsieme di dispositivi, un dispositivo intero o dispositivi multipli collegati in RAID. Ogni volume può essere pensato come un disco virtuale. I volumi possono inoltre contenere diversi sistemi operativi, permettendo al sistema di avviare ed eseguire più di un sistema operativo.

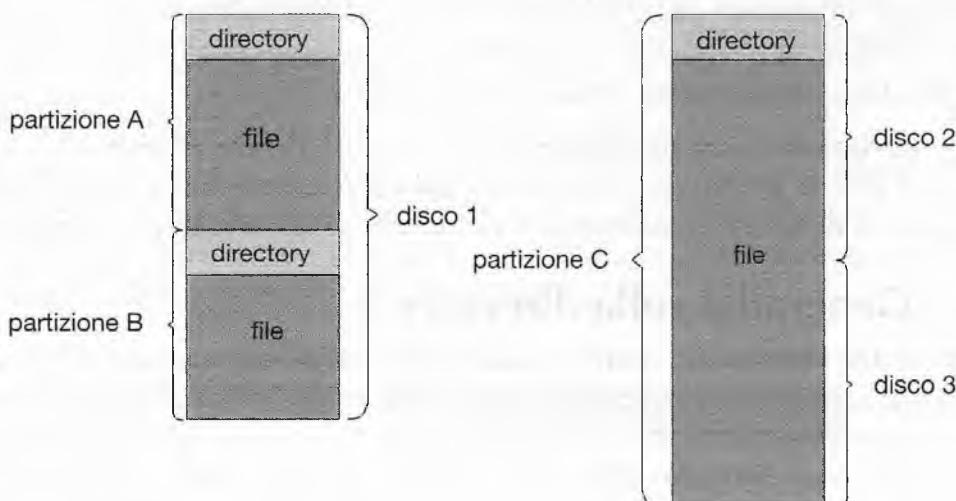
Ogni volume contenente un file system deve anche avere in sé le informazioni sui file presenti nel sistema. Tali informazioni risiedono in una **directory del dispositivo** o **indice del volume**. La directory del dispositivo (in breve **directory**) registra informazioni, quali nome, posizione e tipo, di tutti i file del volume. La Figura 10.6 mostra la tipica organizzazione dei file system.

### 10.3.1 Struttura della memorizzazione di massa

Come discusso, un sistema informatico a uso generale dispone di dispositivi di memorizzazione multipli, che possono essere ripartiti nei volumi che contengono i file system. Il numero di file system in un sistema informatico può variare da zero a molti, e ne esistono di diverso tipo. Per esempio, un tipico sistema Solaris può avere molti file system diversi, come evidenziato nella Figura 10.7.

Nel testo prenderemo in considerazione solo file system a scopo generico. Vale la pena però di notare che esistono molti file system a scopo specifico. Consideriamo i tipi di file system dell'esempio menzionato precedentemente riguardante Solaris:

- ◆ **tmpfs** - un file system “temporaneo” creato nella memoria centrale volatile e i cui contenuti vengono cancellati se il sistema si riavvia o si blocca;
- ◆ **objfs** - un file system “virtuale” (essenzialmente un’interfaccia con il kernel che è simile a un file system) che permette agli strumenti che eseguono il debug di accedere ai simboli del kernel;



**Figura 10.6** Tipica organizzazione di un file system.

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	procfs
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fdfs
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

**Figura 10.7** File system in Solaris.

- ◆ **ctfs** - un file system virtuale che mantiene le informazioni “contrattuali” per gestire quali sono i processi che partono quando il sistema si avvia e quali devono continuare a girare durante il funzionamento del sistema;
- ◆ **lofs** - un file system di tipo “loop back” che permette di accedere a un file system piuttosto che a un altro;
- ◆ **procfs** - un file system virtuale che presenta le informazioni su tutti i processi presenti come se esse risiedessero su un file system;
- ◆ **ufs, zfs** - file system a scopo generico.

I file system dei computer possono quindi essere numerosi. Persino all'interno di un file system è utile dividere i file in gruppi da gestire e sui quali agire. Un'organizzazione di questo tipo richiede l'utilizzo di directory, argomento che esamineremo nel resto di questo paragrafo.

### 10.3.2 Generalità sulla directory

La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Da questo punto di vista, si capisce che la stessa directory si può organizzare in molti modi diversi; deve essere possibile inserire nuovi elementi, cancellarne di esistenti, cercare un elemento ed elencare tutti gli elementi della directory. In questo paragrafo si affronta l'analisi delle appropriate strutture dati utilizzabili per la realizzazione delle directory. Nel considerare una particolare struttura della directory si deve tenere presente l'insieme delle seguenti operazioni che si possono eseguire su una directory.

- ◆ **Ricerca di un file.** Deve esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- ◆ **Creazione di un file.** Deve essere possibile creare nuovi file e aggiungerli alla directory.
- ◆ **Cancellazione di un file.** Quando non serve più, si deve poter rimuovere un file dalla directory.
- ◆ **Elencazione di una directory.** Deve esistere la possibilità di elencare tutti i file di una directory, e il contenuto degli elementi della directory associati ai rispettivi file nell'elenco.
- ◆ **Ridenominazione di un file.** Poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l'uso del file subiscono cambiamenti. La ridefinizione di un file potrebbe comportare la variazione della posizione del file nella directory.
- ◆ **Attraversamento del file system.** Si potrebbe voler accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell'intero file system a intervalli regolari. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (*backup*) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema o più semplicemente se un file non è più in uso. In quest'ultimo caso si può liberare lo spazio da esso occupato nel disco, riutilizzabile quindi per altri file.

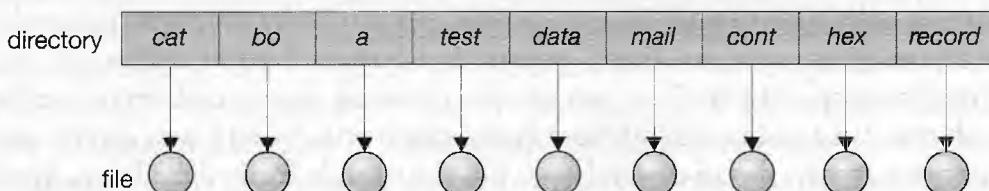
Nei paragrafi seguenti sono descritti gli schemi più comuni per la definizione della struttura logica di una directory.

### 10.3.3 Directory a livello singolo

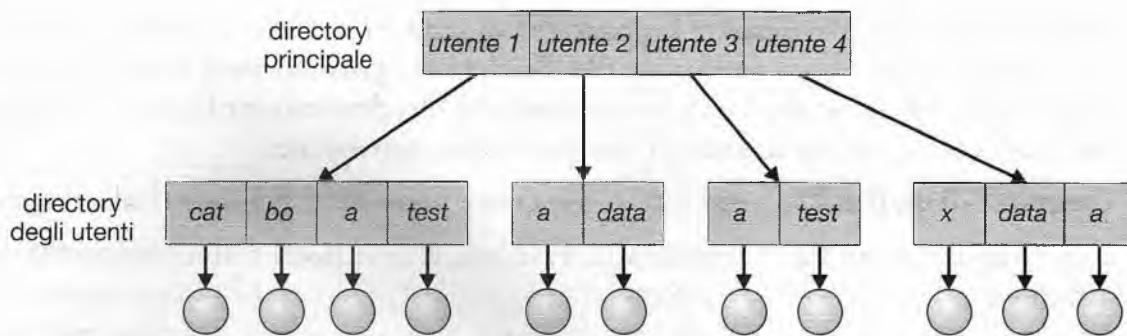
La struttura più semplice per una directory è quella a livello singolo. Tutti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile (Figura 10.8).

Una directory a livello singolo presenta però limiti notevoli che si manifestano all'aumentare del numero dei file in essa contenuti, oppure se il sistema è usato da più utenti. Poiché si trovano tutti nella stessa directory, i file devono avere nomi unici; se due utenti attribuiscono lo stesso nome al loro file di dati, ad esempio *prova*, si viola la regola del nome unico. Anche se i nomi dei file generalmente si scelgono in modo da riflettere il contenuto del file stesso, spesso hanno una lunghezza limitata (il sistema operativo MS-DOS permette nomi di non più di 11 caratteri, lo UNIX permette lunghezze di 255 caratteri).

Anche per un solo utente, con una directory a livello singolo, con l'aumentare del loro numero diventa difficile ricordare i nomi dei file. Non è affatto raro che un utente abbia centinaia di file in un calcolatore e altrettanti file in un altro sistema. In un tale ambiente, sarebbe un compito arduo dover ricordare tanti nomi di file.



**Figura 10.8** Directory a livello singolo.



**Figura 10.9** Struttura della directory a due livelli.

### 10.3.4 Directory a due livelli

Come abbiamo visto, una directory a livello singolo spesso causa confusione dei nomi dei file tra diversi utenti. La soluzione più ovvia prevede la creazione di una directory *separata* per ogni utente.

Nella struttura a due livelli, ogni utente dispone della propria **directory utente** (*user file directory*, UFD). Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. Quando comincia l'elaborazione di un lavoro dell'utente, oppure un utente inizia una sessione di lavoro, si fa una ricerca nella **directory principale** (*master file directory*, MFD) del sistema. La directory principale viene indirizzata con il nome dell'utente o il numero che lo rappresenta, e ogni suo elemento punta alla relativa directory utente (Figura 10.9).

Quando un utente fa un riferimento a un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory utente siano unici. Per creare un file per un utente, il sistema operativo controlla che non ci sia un altro file con lo stesso nome soltanto nella directory di tale utente. Per cancellare un file il sistema operativo limita la propria ricerca alla directory utente locale, quindi non può cancellare per errore un file con lo stesso nome che appartenga a un altro utente.

Le stesse directory utente devono essere create e cancellate come è necessario; a tale scopo si esegue uno speciale programma di sistema con nome dell'utente e dati contabili adeguati. Il programma crea una nuova directory utente e aggiunge l'elemento a essa corrispondente nella directory principale. L'esecuzione di questo programma può essere limitata all'amministratore del sistema. L'allocazione dello spazio nei dischi per le directory utente può essere gestita con le tecniche descritte per i file nel Capitolo 11.

Sebbene risolva la questione delle collisioni dei nomi, la struttura della directory a due livelli presenta ancora dei problemi. In effetti, questa struttura isola un utente dagli altri. Questo isolamento può essere un vantaggio quando gli utenti sono completamente indipendenti, ma è uno svantaggio quando gli utenti *vogliono* cooperare e accedere a file di altri utenti. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti.

Se l'accesso è autorizzato, un utente deve avere la possibilità di riferirsi al nome di un file che si trova nella directory di un altro utente. Per attribuire un nome unico a un particolare file di una directory a due livelli, occorre indicare sia il nome dell'utente sia il nome del file. Una directory a due livelli si può pensare come un albero, o almeno un albero rovesciato, di altezza 2. La radice dell'albero è la directory principale, i suoi diretti discendenti sono le directory utente, da cui discendono i file che sono le foglie dell'albero. Specificando un nome utente e un nome di file si definisce un percorso che parte dalla radice (la directo-

ry principale) e arriva a una specifica foglia (il file specificato). Quindi, un nome utente e un nome di file definiscono un *nome di percorso* (*path name*). Ogni file del sistema ha un nome di percorso. Per attribuire un nome unico a un file, un utente deve conoscere il nome di percorso del file desiderato.

Se, ad esempio, l'utente *A* desidera accedere al proprio file chiamato **prova**, è sufficiente che faccia riferimento a **prova**. Invece, per accedere al file denominato **prova** dell'utente *B*, con nome di elemento della directory **utenteB**, l'utente *A* deve fare riferimento a **/utenteB/prova**. Ogni sistema ha la propria sintassi per riferirsi ai file delle directory diverse da quella dell'utente.

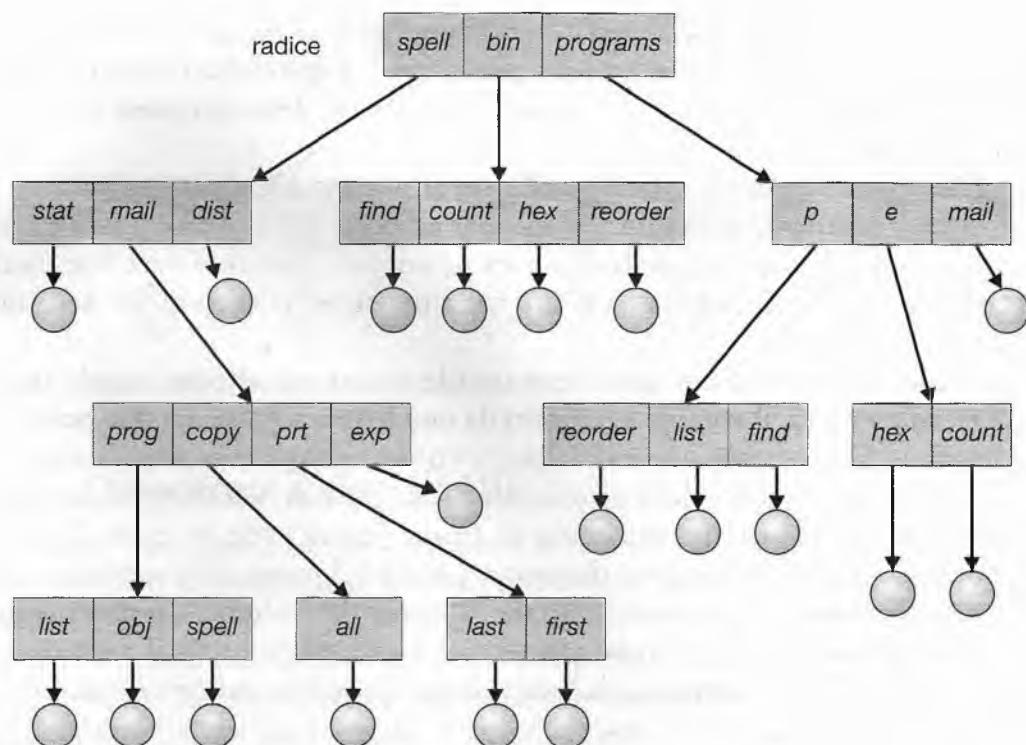
Per specificare il volume cui appartiene un file occorrono ulteriori regole sintattiche. Nell'MS-DOS, ad esempio, il volume è indicato da una lettera seguita dai due punti. Quindi l'indicazione di un file potrebbe essere del tipo **C:\utenteB\prova**. Alcuni sistemi vanno oltre e separano: volume, nome della directory, e nome del file. Nel VMS, ad esempio, il file **login.com** potrebbe essere indicato come **u:[sst.jdeck]login.com;1**, dove **u** è il nome del volume, **sst** è il nome della directory, **jdeck** è il nome della sottodirectory e **1** è il numero della versione. Altri sistemi trattano il nome del volume semplicemente come parte del nome della directory. Il primo elemento è quello del volume, il resto è composto dalla directory e dal file. Ad esempio, **/u/pbg/prova** potrebbe indicare il volume **u**, la directory **pbg** e il file **prova**.

Un caso particolare di questa situazione riguarda i file di sistema. I programmi forniti come elementi integranti del sistema, come caricatori, assemblatori, compilatori, strumenti, librerie e così via, sono infatti definiti come file. Quando si impartiscono al sistema operativo i comandi appropriati, il caricatore legge questi file che poi vengono eseguiti. Molti interpreti di comandi operano semplicemente trattando il comando come il nome di un file da caricare ed eseguire. Poiché il sistema delle directory è già stato definito, questo nome di file viene cercato nella directory utente locale. Una soluzione prevede la copiatura dei file di sistema in ciascuna directory utente. Tuttavia, con la copiatura di tutti i file di sistema si spreca un'enorme quantità di spazio. Se i file di sistema occupano 5 MB, con 12 utenti si avrebbe un'occupazione di spazio pari a  $5 \times 12 = 60$  MB, solo per le copie dei file di sistema.

La soluzione standard prevede una leggera complicazione della procedura di ricerca; si definisce una speciale directory utente contenente i file di sistema, ad esempio la directory utente 0. Ogni volta che si indica un file da caricare, il sistema operativo lo cerca innanzitutto nella directory utente locale, e, se lo trova, lo usa; se non lo trova, il sistema cerca automaticamente nella speciale directory utente contenente i file di sistema. La sequenza delle directory in cui è stata fatta la ricerca avviata dal riferimento a un file è detta **percorso di ricerca** (*search path*). Tale percorso si può estendere in modo da contenere una lista illimitata di directory in cui fare le ricerche quando si dà il nome di un comando. Questo metodo è il più usato in UNIX e MS-DOS. Alcuni sistemi prevedono anche che ogni utente disponga del proprio percorso di ricerca personale.

### 10.3.5 Directory con struttura ad albero

La corrispondenza strutturale tra directory a due livelli e albero a due livelli permette di generalizzare facilmente il concetto, estendendo la struttura della directory a un albero di altezza arbitraria (Figura 10.10). Questa generalizzazione permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. Il file system dell'MS-DOS, ad esempio, è strutturato ad albero; si tratta infatti del più comune tipo di struttura delle directory. L'albero ha una directory radice (*root directory*), e ogni file del sistema ha un unico nome di percorso.



**Figura 10.10** Struttura della directory ad albero.

Una directory, o una sottodirectory, contiene un insieme di file o sottodirectory. Le directory sono semplicemente file, trattati però in modo speciale. Tutte le directory hanno lo stesso formato interno. La distinzione tra file e directory è data dal bit, rispettivamente 0 e 1, di ogni elemento della directory. Per creare e cancellare le directory si adoperano speciali chiamate di sistema.

Normalmente, ogni utente dispone di una directory corrente. La directory corrente deve contenere la maggior parte dei file di interesse corrente per il processo. Quando si fa un riferimento a un file, si esegue una ricerca nella directory corrente; se il file non si trova in tale directory, l'utente deve specificare un nome di percorso oppure cambiare la directory corrente facendo diventare tale la directory contenente il file desiderato. Per cambiare directory corrente si fa uso di una chiamata di sistema che preleva un nome di directory come parametro e lo usa per ridefinire la directory corrente. Quindi, l'utente può cambiare la propria directory corrente ogni volta che lo desidera. Da una chiamata di sistema `change directory` alla successiva, tutte le chiamate di sistema `open()` cercano i file specificati nella directory corrente. Si noti che il percorso di ricerca può contenere o meno uno speciale elemento che rappresenta "la directory corrente".

La directory corrente iniziale di un utente è stabilita all'avvio del lavoro d'elaborazione dell'utente, oppure quando quest'ultimo inizia una sessione di lavoro; il sistema operativo cerca nel file di contabilizzazione, o in qualche altra locazione predefinita, l'elemento relativo a questo utente. Nel file di contabilizzazione è memorizzato un puntatore alla (oppure il nome della) directory iniziale dell'utente. Tale puntatore viene copiato in una variabile locale per l'utente che specifica la sua directory corrente iniziale. Dalla shell dell'utente si possono poi avviare altri processi: la loro directory corrente è solitamente la directory corrente del processo genitore al momento della creazione del figlio.

I nomi di percorso possono essere di due tipi: **nomi di percorso assoluti** e **nomi di percorso relativi**. Un *nome di percorso assoluto* comincia dalla radice dell'albero di directory

e segue un percorso che lo porta fino al file specificato indicando i nomi delle directory che costruiscono le tappe del suo cammino. Un *nome di percorso relativo* definisce un percorso che parte dalla directory corrente. Ad esempio, nel file system con struttura ad albero della Figura 10.10, se la directory corrente è `root/spell/mail`, il nome di percorso relativo `prt/first` si riferisce allo stesso file indicato dal percorso assoluto `root/spell/mail/prt/first`.

Se si permette all'utente di definire le proprie sottodirectory, gli si consente anche di dare una struttura ai suoi file. Questa struttura può presentare directory distinte per file associati a soggetti diversi; ad esempio, si può creare una sottodirectory contenente il testo di questo libro, oppure diversi tipi di informazioni, ad esempio la directory `programmi` può contenere programmi sorgente; la directory `bin` può contenere tutti i programmi eseguibili.

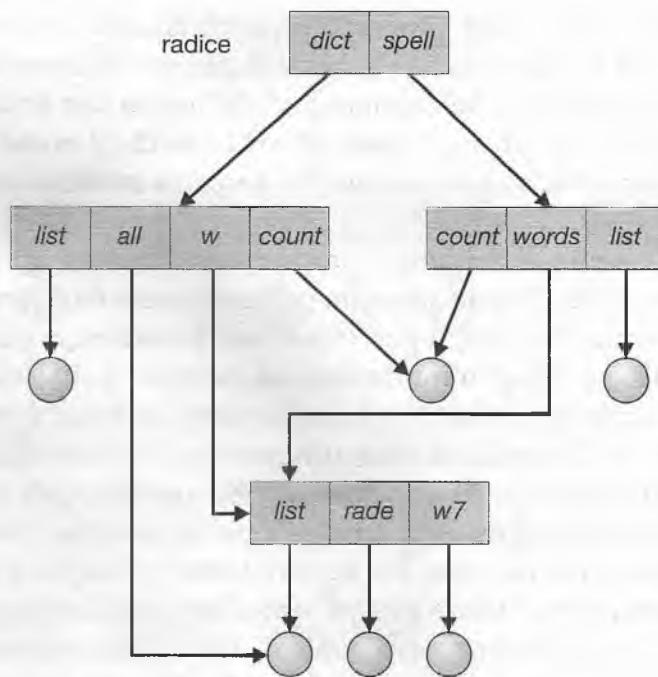
Una decisione importante relativa alla strutturazione ad albero delle directory riguarda il modo di gestire la cancellazione di una directory. Se una directory è vuota, è sufficiente cancellare l'elemento che la designa nella directory che la contiene. Tuttavia se la directory da cancellare non è vuota, ma contiene file oppure sottodirectory, è possibile procedere in due modi. Alcuni sistemi, come l'MS-DOS, non cancellano una directory a meno che non sia vuota; per cancellarla l'utente deve prima cancellare i file in essa contenuti. Se esiste qualche sottodirectory, questa procedura si deve applicare anche alle sottodirectory. Questo metodo può richiedere una discreta quantità di lavoro. In alternativa, come nel comando `rm` di UNIX, si può avere un'opzione che, alla richiesta di cancellazione di una directory, cancelli anche tutti i file e tutte le sottodirectory in essa contenuti. Entrambi i criteri sono abbastanza facili da realizzare; si tratta soltanto di stabilire quale seguire. Il secondo criterio è più comodo, anche se più pericoloso, poiché si può rimuovere un'intera struttura della directory con un solo comando. Se si eseguisse tale comando per sbaglio sarebbe necessario ripristinare un gran numero di file e directory dalle copie di riserva (ipotizzandone l'esistenza).

Con un sistema di directory strutturato ad albero anche l'accesso ai file di altri utenti è di facile realizzazione. Ad esempio, l'utente *B* può accedere ai file dell'utente *A* specificando i nomi di percorso assoluti oppure relativi. In alternativa, l'utente *B* può far sì che la propria directory corrente sia quella dell'utente *A* e accedere ai file usando direttamente i loro nomi.

Un percorso per un file in una directory strutturata ad albero può ovviamente essere più lungo di quello in una directory a due livelli. Per consentire agli utenti di accedere ai programmi senza dover ricordare tali lunghi percorsi, il sistema operativo Macintosh cerca automaticamente i programmi eseguibili attraverso un file detto `Desktop file` contenente il nome e la locazione di tutti i programmi eseguibili a esso noti. Quando si aggiunge al sistema un nuovo disco o si accede a una rete di comunicazione, il sistema operativo attraversa la struttura delle directory del dispositivo alla ricerca dei programmi eseguibili e registra le informazioni relative. Questo meccanismo consente la funzione del doppio clic descritta precedentemente. Un doppio clic sul nome di un file, o sull'icona che lo rappresenta, determina la lettura del suo attributo di creazione e la ricerca dell'elemento corrispondente nel `Desktop file`. Una volta trovato tale elemento, si avvia il programma eseguibile associato che apre il file su cui si è fatto il doppio clic.

### 10.3.6 Directory con struttura a grafo aciclico

Si considerino due programmatorei che lavorano a un progetto comune. I file associati a quel progetto si possono memorizzare in una sottodirectory, separandoli da altri progetti e file dei due programmatorei, ma poiché entrambi i programmatorei hanno le stesse responsabilità sul progetto, ciascuno richiede che la sottodirectory si trovi nelle proprie directory. La sot-



**Figura 10.11** Struttura della directory a grafo aciclico.

La directory comune deve essere *condivisa*. Nel sistema esiste quindi una directory condivisa o un file condiviso in due, o più, posizioni alla volta.

La struttura ad albero non ammette la condivisione di file o directory. Un **grafo aciclico** permette alle directory di avere sottodirectory e file condivisi (Figura 10.11). Lo stesso file o la stessa sottodirectory possono essere in due directory diverse. Un grafo aciclico, cioè senza cicli, rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.

Il fatto che un file sia condiviso, o che lo sia una directory, non significa che ci siano due copie del file: con due copie ciascun programmatore potrebbe vedere la copia presente nella propria directory e non l'originale; se un programmatore modifica il file, le modifiche non appaiono nell'altra copia. Se invece il file è condiviso esiste un solo file effettivo, perciò tutte le modifiche sono immediatamente visibili. La condivisione è di particolare importanza se applicata alle sottodirectory; un nuovo file appare automaticamente in tutte le sottodirectory condivise.

Quando più persone lavorano insieme, tutti i file da condividere si possono inserire in una directory comune. Ciascuna directory utente di tutti i membri del gruppo contiene questa directory di file condivisi in forma di sottodirectory. Anche nel caso di un singolo utente, il metodo di gestione dei file di tale utente può richiedere che alcuni file siano inseriti in più sottodirectory. Ad esempio, un programma scritto per un progetto particolare deve trovarsi sia nella directory di tutti i programmi sia nella directory di quel progetto.

I file e le sottodirectory condivisi si possono realizzare in molti modi. Un metodo diffuso, esemplificato da molti tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory, chiamato collegamento. Un **collegamento (link)** è un puntatore a un altro file o un'altra directory. Ad esempio, un collegamento si può realizzare come un nome di percorso assoluto o relativo. Quando si fa riferimento a un file, si compie una ricerca nella directory, l'elemento cercato è contrassegnato come collegamento e riporta il nome di percorso del file, o della directory, reale. Quindi si *risolve* il collegamento usando il nome di percorso per localizzare il file reale. I collegamenti si identificano facilmente tramite il loro formato nell'e-

mento della directory (o, nei sistemi che gestiscono i tipi, dal tipo speciale), e sono infatti chiamati puntatori indiretti. Durante l'attraversamento degli alberi delle directory il sistema operativo ignora questi collegamenti, preservandone così la struttura aciclica.

Un altro comune metodo per la realizzazione dei file condivisi prevede semplicemente la duplicazione di tutte le informazioni relative ai file in entrambe le directory di condivisione, quindi i due elementi sono identici. Un collegamento è chiaramente diverso dagli altri elementi della directory. Duplicando gli elementi della directory, invece, la copia e l'originale sono resi indistinguibili: sorge allora il problema di mantenere la coerenza se il file viene modificato.

Una struttura della directory a grafo aciclico è più flessibile di una semplice struttura ad albero, ma anche più complessa. Si devono prendere in considerazione parecchi problemi. Un file può avere più nomi di percorso assoluti, quindi nomi diversi possono riferirsi allo stesso file. Questa situazione è simile al problema dell'uso degli *alias* nei linguaggi di programmazione. Quando si percorre tutto il file system – per trovare un file, per raccogliere dati statistici su tutti i file o per fare le copie di riserva di tutti i file – il problema diviene più grave poiché le strutture condivise non si devono attraversare più di una volta.

Un altro problema riguarda la cancellazione, poiché è necessario stabilire in quali casi è possibile allocare e riutilizzare lo spazio allocato a un file condiviso. Una possibilità prevede che a ogni operazione di cancellazione seguva l'immediata rimozione del file; quest'azione può però lasciare puntatori a un file che ormai non esiste più. Sarebbe ancora più grave se i puntatori contenessero gli indirizzi effettivi del disco e lo spazio fosse poi riutilizzato per altri file, poiché i puntatori potrebbero puntare nel mezzo di questi altri file.

In un sistema dove la condivisione è realizzata da collegamenti simbolici la gestione di questa situazione è relativamente semplice. La cancellazione di un collegamento non influisce sul file originale, poiché si rimuove solo il collegamento. Se si cancella il file, si libera lo spazio corrispondente lasciando in sospeso il collegamento; a questo punto è possibile cercare tutti questi collegamenti e rimuoverli, ma se in ogni file non esiste una lista dei collegamenti associati al file stesso questa ricerca può essere abbastanza onerosa. In alternativa, si possono lasciare i collegamenti finché non si tenta di usarli, quindi si “scopre” che il file con il nome dato dal collegamento non esiste e non si riesce a determinare il collegamento rispetto al nome; l'accesso è trattato proprio come qualsiasi altro nome di file irregolare. In questo caso, il progettista del sistema deve decidere attentamente cosa si debba fare quando si cancella un file e si crea un altro file con lo stesso nome, prima che sia stato usato un collegamento simbolico al file originario. In UNIX, quando si cancella un file, i collegamenti simbolici restano, è l'utente che deve rendersi conto che il file originale è scomparso o è stato sostituito. Nella famiglia di sistemi operativi Microsoft Windows si segue lo stesso criterio.

Un altro tipo di approccio prevede la conservazione del file finché non siano stati cancellati tutti i riferimenti a esso. In questo caso è necessario disporre di un meccanismo che determini la cancellazione dell'ultimo riferimento a quel file; è possibile tenere una lista di tutti i riferimenti a un file (elementi di directory o collegamenti simbolici). Quando si crea un collegamento, oppure una copia dell'elemento della directory, si aggiunge un nuovo elemento alla lista dei riferimenti al file; quando si cancella un collegamento oppure un elemento della directory, si elimina dalla lista l'elemento corrispondente. Quando la sua lista di riferimenti è vuota, il file viene cancellato.

Questo metodo presenta, però, un problema: la dimensione della lista dei riferimenti al file può essere variabile e potenzialmente grande. Tuttavia, non è realmente necessario mantenere l'intera lista, è sufficiente un contatore del *numero* di riferimenti. Un nuovo collegamento o un nuovo elemento della directory incrementa il numero dei riferimenti; la

cancellazione di un collegamento o un elemento decrementa questo numero. Quando il contatore è uguale a 0 si può cancellare il file, poiché non ci sono più riferimenti a tale file. Il sistema operativo UNIX usa questo metodo per i collegamenti non simbolici, o **collegamenti effettivi (hard link)**; il contatore dei riferimenti è tenuto nel blocco di controllo del file o *inode*. Impedendo che si facciano più riferimenti a una directory, si può mantenere una struttura a grafo aciclico.

Per evitare questi problemi alcuni sistemi non consentono la condivisione delle directory né i collegamenti. Nell'MS-DOS, ad esempio, la struttura della directory è una struttura ad albero anziché a grafo aciclico.

### 10.3.7 Directory con struttura a grafo generale

Un serio problema connesso all'uso di una struttura a grafo aciclico consiste nell'assicurare che non vi siano cicli. Iniziando con una directory a due livelli e permettendo agli utenti di creare sottodirectory si crea una directory con struttura ad albero. Aggiungendo nuovi file e nuove sottodirectory alla directory con struttura ad albero, la natura di quest'ultima persiste. Tuttavia, quando si aggiungono dei collegamenti a una directory con struttura ad albero, tale struttura si trasforma in una semplice struttura a grafo, come quella illustrata nella Figura 10.12.

Il vantaggio principale di un grafo aciclico è dato dalla semplicità degli algoritmi necessari per attraversarlo e per determinare quando non ci siano più riferimenti a un file. È preferibile evitare un duplice attraversamento di sezioni condivise di un grafo aciclico, soprattutto per motivi di prestazioni. Se un file particolare è stato appena cercato in una sottodirectory condivisa, ma non è stato trovato, è preferibile evitare una seconda ricerca nella stessa sottodirectory, che costituirebbe solo una perdita di tempo.

Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un ciclo infinito di ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca.

Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con le strutture delle directory a grafo aciclico, la presenza di uno 0 nel

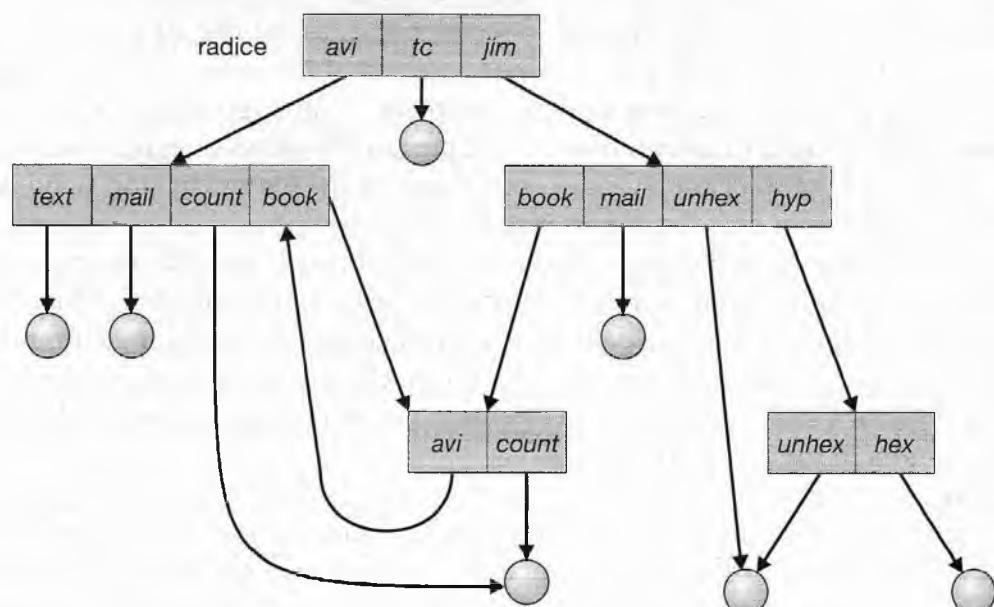


Figura 10.12 Directory a grafo generale.

contatore dei riferimenti significa che non esistono più riferimenti al file o alla directory, e quindi il file può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore dei riferimenti possa essere non nullo, anche se non è più possibile far riferimento a una directory o a un file. Questa anomalia è dovuta alla possibilità di autoriferimento (ciclo) nella struttura delle directory. In questo caso è generalmente necessario usare un metodo di “ripulitura” (*garbage collection*) per stabilire quando sia stato cancellato l’ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. Tale metodo implica l’attraversamento del file system, durante il quale si contrassegna tutto ciò che è accessibile; in un secondo passaggio si raccoglie in un elenco di blocchi liberi tutto ciò che non è contrassegnato. Una procedura di marcatura analoga è utilizzabile per assicurare che un’attraversamento o una ricerca coprano tutto quel che si trova nel file system una sola volta. L’applicazione di questo metodo a un file system basato su dischi richiede però molto tempo, perciò viene tentata solo di rado.

Inoltre, poiché è necessaria solo a causa della presenza dei cicli, è molto più conveniente lavorare con una struttura a grafo aciclico. La difficoltà consiste nell’evitare i cicli quando si aggiungono nuovi collegamenti alla struttura. Per sapere quando un nuovo collegamento ha completato un ciclo si possono impiegare gli algoritmi che permettono di individuare la presenza di cicli nei grafi. Dal punto di vista del calcolo, però, questi algoritmi sono onerosi, soprattutto quando il grafo si trova in memoria secondaria. Nel caso particolare di directory e collegamenti, un semplice algoritmo prevede di evitare i collegamenti durante l’attraversamento delle directory: si evitano così i cicli senza alcun carico ulteriore.

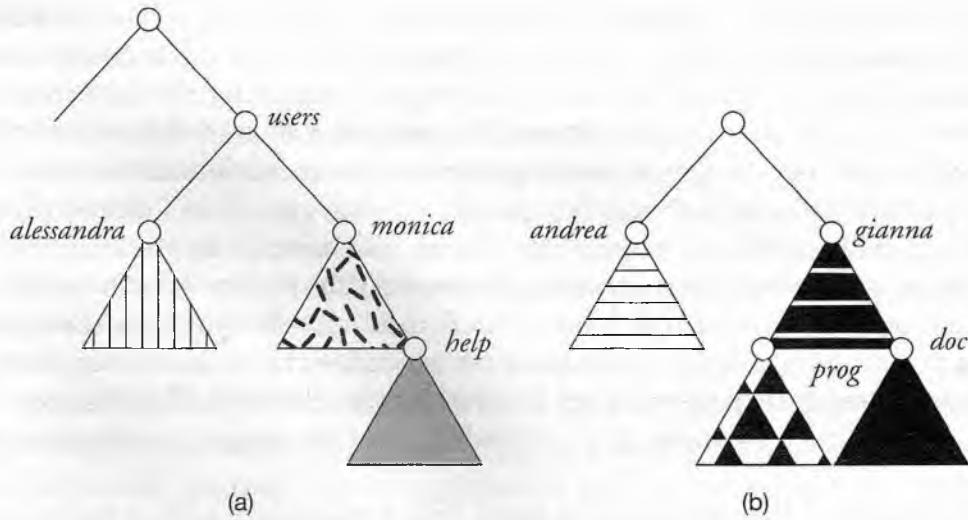
## 10.4 Montaggio di un file system

Così come si deve *aprire* un file per poterlo usare, per essere reso accessibile ai processi di un sistema, un file system deve essere *montato*. La struttura delle directory può ad esempio essere composta di volumi, che devono essere montati affinché siano disponibili nello spazio dei nomi di un file system.

La procedura di montaggio è molto semplice: si fornisce al sistema operativo il nome del dispositivo e la sua locazione (detta **punto di montaggio**) nella struttura di file e directory alla quale agganciare il file system. Alcuni sistemi operativi richiedono un file system prefissato, mentre altri ispezionano le strutture del dispositivo e determinano il tipo del file system presente. Di solito, un punto di montaggio è una directory vuota cui sarà agganciato il file system che deve essere montato. Ad esempio, in un sistema UNIX, un file system contenente le directory iniziali degli utenti si potrebbe montare come `/home`; quindi la directory iniziale dell’utente *gianna* avrebbe il percorso `/home/gianna`. Se lo stesso file system si montasse come `/users` il percorso per quella directory sarebbe `/users/gianna`.

Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system contenuto nel dispositivo. La verifica si compie chiedendo al driver del dispositivo di leggere la directory di dispositivo e controllando che tale directory abbia il formato previsto. Infine, il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato. Questo schema permette al sistema operativo di attraversare la sua struttura della directory, passando da un file system all’altro secondo le necessità.

Per illustrare l’operazione di montaggio, si consideri il file system rappresentato nella Figura 10.13, in cui i triangoli rappresentano sottoalberi di directory rilevanti. La Figura 10.13(a), mostra un file system esistente, mentre nella Figura 10.13(b) è raffigurato un volume non ancora montato che risiede in `/device/dsk`. A questo punto, si può accedere

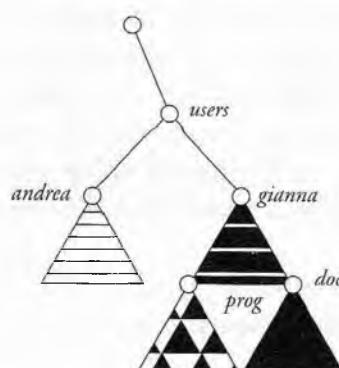


**Figura 10.13** File system; (a) esistente; (b) volume non montato.

solo ai file del file system esistente. Nella Figura 10.14 si possono vedere gli effetti dell'operazione di montaggio del volume residente in `/device/dsk` al punto di montaggio `/users`. Se si smonta il volume, il file system ritorna alla situazione rappresentata nella Figura 10.13.

Per rendere più chiare le loro funzioni, sistemi operativi impongono una semantica a queste operazioni. Ad esempio, un sistema potrebbe vietare il montaggio in una directory contenente file, o rendere disponibile il file system montato in tale directory e nascondere i file preesistenti nella directory finché non si *smonti* il file system, concludendone l'uso e permettendo l'accesso ai file originariamente presenti in tale directory. Come ulteriore esempio, un sistema potrebbe permettere il montaggio ripetuto dello stesso file system in diversi punti di montaggio, o potrebbe imporre una sola possibilità di montaggio per ciascun file system.

Si considerino le azioni del sistema operativo Macintosh. Ogni volta che il sistema rileva per la prima volta un disco (i dischi sono rilevati nella fase d'avviamento, mentre i dischetti quando sono inseriti nella relativa unità), il sistema operativo Macintosh cerca un file system nel dispositivo. Se ne trova uno, esegue automaticamente il montaggio del file system a livello della radice, aggiungendo un'icona di cartella sulla scrivania virtuale, etichettata con il nome del file system (secondo quel che è memorizzato nella directory di dispositivo). A questo punto l'utente può selezionare l'icona con il mouse e quindi vedere il contenuto del nuovo file system appena montato. L'OS X di Mac si comporta in modo si-



**Figura 10.14** Punto di montaggio.

mile a UNIX BSD, sul quale si basa. Tutti i file system vengono montati sotto la directory/Volumes. L'interfaccia grafica di Mac OS X nasconde questa caratteristica e mostra i file system come se fossero tutti montati al livello della radice.

La famiglia di sistemi operativi Microsoft Windows (95, 98, NT, 2000, 2003, XP, Vista) mantiene una struttura della directory a due livelli estesa, con una lettera di unità associata a dispositivi e volumi, che hanno una struttura della directory a grafo generale associata a una lettera di unità. Il percorso completo per uno specifico file è quindi della forma `lettera_di_unità:\percorso\file`. Le versioni più recenti di Windows permettono di montare un file system su qualunque punto dell'albero delle directory, esattamente come UNIX. I sistemi Windows rilevano automaticamente tutti i dispositivi e montano tutti i file system rilevati all'avvio della macchina. In altri sistemi, per esempio UNIX, occorre montare esplicitamente i file system. Un file di configurazione del sistema contiene una lista di dispositivi (e relativi punti di montaggio) da montare automaticamente all'avvio, ma altri montaggi possono essere eseguiti manualmente.

Le questioni riguardanti il montaggio di file system sono approfondite nel Paragrafo 11.2.2.

## 10.5 Condivisione di file

Nei paragrafi precedenti sono state presentate le motivazioni alla base della necessità di condivisione dei file, oltre ad alcune difficoltà che si incontrano nel permettere che diversi utenti possano condividere file. Tale possibilità è particolarmente utile per utenti che vogliono collaborare e per ridurre le risorse richieste per raggiungere un certo obiettivo di calcolo. Quindi, nonostante le difficoltà inerenti alla condivisione, i sistemi operativi orientati agli utenti devono soddisfare questa esigenza.

In questo paragrafo si considerano diversi aspetti della condivisione dei file, innanzitutto l'aspetto relativo alla molteplicità degli utenti e ai diversi metodi di condivisione possibili. Una volta che più utenti possono condividere file, l'obiettivo diventa estendere la condivisione a più file system, compresi i file system remoti. Infine, ci possono essere diverse interpretazioni delle azioni conflittuali intraprese su file condivisi. Ad esempio, se più utenti stanno scrivendo nello stesso file, ci si può chiedere se il sistema dovrebbe permettere tutte le operazioni di scrittura, oppure dovrebbe proteggere le azioni di ciascun utente da quelle degli altri.

### 10.5.1 Utenti multipli

Se un sistema operativo permette l'uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e alla loro protezione. Data una struttura della directory che permette la condivisione di file da parte degli utenti, il sistema deve poter mediare questa condivisione. Il sistema può permettere a ogni utente, in modo predefinito, di accedere ai file degli altri utenti, oppure può richiedere che un utente debba esplicitamente concedere i permessi di accesso ai file. Questi aspetti sono alla base dei temi del controllo degli accessi e della protezione, trattati nel Paragrafo 10.6.

Per realizzare i meccanismi di condivisione e protezione, il sistema deve memorizzare e gestire più attributi di directory e file rispetto a un sistema che consente un singolo utente. Sebbene storicamente siano stati proposti molti metodi per la realizzazione di questi meccanismi, la maggior parte dei sistemi ha adottato – relativamente a ciascun file o directory –

i concetti di *proprietario* (o utente) e *gruppo*. Il proprietario è l'utente che può cambiare gli attributi di un file o directory, concedere l'accesso e che, in generale, ha il maggior controllo sul file o directory. L'attributo di gruppo di un file si usa per definire il sottoinsieme di utenti autorizzati a condividere l'accesso al file. Ad esempio, il proprietario di un file in un sistema UNIX può fare qualsiasi operazione sul file, mentre i membri del gruppo possono compiere un sottoinsieme di queste operazioni e il resto degli utenti un altro sottoinsieme. Il proprietario del file può definire l'esatto insieme di operazioni che i membri del gruppo e gli altri utenti possono eseguire. Maggiori dettagli sugli attributi che regolano i permessi sono trattati nel paragrafo successivo.

Gli identificatori del gruppo e del proprietario di un certo file o directory (ID) sono memorizzati insieme con gli altri attributi del file. Quando un utente richiede di compiere un'operazione su un file, per verificare se l'utente richiedente è il proprietario del file si può confrontare l'identificatore utente con l'attributo che identifica il proprietario. Analogamente si confrontano gli identificatori di gruppo. Ne risultano i permessi che l'utente ha sul file, e che il sistema considera per consentire o impedire l'operazione richiesta.

Molti sistemi operativi hanno più file system locali, inclusi volumi di un unico disco, o più volumi in diversi dischi connessi al sistema. In questi casi, la verifica degli identificatori e il confronto dei permessi si possono fare facilmente dopo l'operazione di montaggio dei file system.

## 10.5.2 File system remoti

L'avvento delle reti (Capitolo 16) ha permesso la comunicazione tra calcolatori separati da grandi distanze. Le reti permettono la condivisione di risorse sparse nell'area di un *campus* universitario o addirittura in diversi luoghi del mondo. Un'ovvia risorsa da condividere sono i dati, nella forma di file.

I metodi con i quali i file si condividono in una rete sono cambiati molto, seguendo l'evoluzione della tecnologia delle reti e dei file. Un primo metodo consiste nel trasferimento dei file richiesto in modo esplicito dagli utenti, attraverso programmi come l'`ftp`. Un secondo metodo, molto diffuso, è quello del **file system distribuito** (*distributed file system*, DFS), che permette la visibilità nel calcolatore locale delle directory remote. Il terzo metodo, il **World Wide Web**, è, da un certo punto di vista, il contrario del primo. Per accedere ai file remoti si usa un programma di consultazione (*browser*), e operazioni distinte – essenzialmente un involucro (*wrapper*) per l'`ftp` – per trasferirli.

L'`ftp` si usa sia per l'accesso anonimo sia per quello autenticato. L'accesso anonimo permette di trasferire file senza essere utenti accreditati nel sistema remoto. Il World Wide Web usa quasi esclusivamente lo scambio di file anonimo. Un DFS comporta un'integrazione molto più stretta tra il calcolatore che accede ai file remoti e il calcolatore che fornisce i file. Tale integrazione incrementa la complessità, illustrata in questo paragrafo.

### 10.5.2.1 Modello client-server

I file system remoti permettono il montaggio di uno o più file system di uno o più calcolatori remoti in un calcolatore locale. In questo caso, il calcolatore contenente i file si chiama *server*, mentre il calcolatore che richiede l'accesso ai file si chiama *client*. La relazione tra client e server è piuttosto comune tra i calcolatori di una rete. In generale, il server dichiara che determinate risorse sono disponibili ai client, specificando esattamente quali (in questo caso, quali file) ed esattamente a quali client. Un server può gestire richieste provenienti da più client, e un client può accedere a più server, secondo i dettagli del particolare sistema client-server.

Il server in genere specifica i file disponibili su di un volume o livello di directory. L'identificazione certa dei client è più difficile; proprio perché può avvenire facilmente tramite i relativi nomi simbolici di rete, o tramite altri identificatori, si può altrettanto facilmente **ingannare** un server imitando l'identificatore di un client accreditato (si tratta del cosiddetto *spoofing*). Un client non accreditato o privo di determinate autorizzazioni può ingannare un server presentandosi con l'identità di un altro client che possiede le autorizzazioni che il client impostore vuole ottenere. Tra le soluzioni più sicure ci sono quelle che prevedono l'autenticazione reciproca dei client e dei server tramite chiavi di cifratura. Sfortunatamente, l'introduzione di tecniche per la sicurezza introduce nuovi problemi, ad esempio la necessità della compatibilità tra client e server (si devono impiegare gli stessi algoritmi di cifratura) e dello scambio sicuro delle chiavi di cifratura (l'intercettazione delle chiavi può permettere accessi non autorizzati). Questi problemi sono sufficientemente difficili da far sì che nella maggioranza dei casi si usino metodi di autenticazione insicuri.

Nel caso di UNIX e del suo file system di rete (*network file system*, NFS), l'autenticazione avviene, ordinariamente, tramite le informazioni di connessione relative al client. In questo schema, gli identificatori (ID) che identificano l'utente devono coincidere nel client e nel server; diversamente, il server non può determinare i diritti d'accesso ai file. Si consideri ad esempio un utente con un identificatore uguale a 1000 nel client e a 2000 nel server. Una richiesta per uno specifico file dal client al server non potrà essere gestita correttamente, perché il server cercherà di determinare se l'utente 1000 ha i permessi d'accesso al file, invece di usare il *reale* identificatore dell'utente che è 2000. L'accesso sarà concesso o negato secondo un'informazione di autenticazione sbagliata. Il server deve fidarsi del client e assumere che quest'ultimo gli presenta l'identificatore corretto. I protocolli NFS permettono relazioni da molti a molti; cioè più server possono fornire file a più client. Infatti, un calcolatore può comportarsi sia da server per altri client NFS, sia da client di altri server NFS.

Una volta montato il file system remoto, le richieste delle operazioni su file sono inviate al server, attraverso la rete, per conto dell'utente, usando il protocollo DFS. Normalmente, una richiesta di apertura di file si invia insieme con l'identificatore dell'utente richiedente. Il server quindi applica i normali controlli d'accesso per determinare se l'utente ha le credenziali per accedere al file nel modo richiesto; se tali controlli hanno esito positivo, riporta un handle ("maniglia") d'accesso al file (*file handle*) all'applicazione client, che la usa per eseguire sul file operazioni di lettura, scrittura e altro. Il client chiude il file quando non deve più accedervi. Il sistema operativo può applicare una semantica simile a quella adottata per il montaggio di un file system locale, oppure una semantica diversa.

### 10.5.2.2 Sistemi informativi distribuiti

Per semplificare la gestione dei servizi client-server, i sistemi informativi distribuiti, noti anche come servizi di **nominazione distribuiti**, sono stati concepiti per fornire un accesso unificato alle informazioni necessarie per il calcolo remoto. Il **sistema dei nomi di dominio** (*domain name system*, DNS) fornisce le traduzioni dai nomi dei calcolatori agli indirizzi di rete per l'intera Internet (compreso il World Wide Web). Prima che s'inventasse il DNS e che si diffondesse capillarmente nella rete, si scambiavano tra i calcolatori, per posta elettronica o *ftp*, file contenenti le stesse informazioni. Questo metodo non poteva adattarsi dinamicamente all'aumento delle dimensioni della rete Internet. Il DNS è trattato ulteriormente nel Paragrafo 16.5.1.

Altri sistemi informativi distribuiti forniscono uno spazio identificato da *nome utente/parola d'ordine/identificatore utente/identificatore di gruppo* per un servizio distribuito. I sistemi UNIX hanno adottato un'ampia varietà di metodi per l'informazione distribuita. Sun Microsystems ha introdotto il sistema *yellow pages* (poi ribattezzato *network information*

*service*, NIS), adottato da gran parte dell'industria. Questo servizio centralizza la memorizzazione dei nomi degli utenti e dei calcolatori, delle informazioni sulle stampanti e altro. Sfortunatamente, usa metodi di autenticazione insicuri, ad esempio l'invio di parole d'ordine dell'utente non cifrate (*in chiaro*) e l'identificazione dei calcolatori attraverso gli indirizzi IP. Il sistema NIS+ di Sun è una versione del NIS più sicura, ma è anche molto più complessa e non ha avuto una gran diffusione.

Nel caso delle reti di Microsoft (*common internet file system*, CIFS), le informazioni di rete si usano insieme con gli elementi di autenticazione dell'utente (nome dell'utente e parola d'ordine) per creare un **nome utente di rete** (*network login*) che il server usa per decidere se permettere o negare l'accesso a un file system richiesto. Affinché questa autenticazione sia valida, i nomi utente devono essere uguali nel calcolatore client e nel server (come per l'NFS). Per fornire un unico spazio di nomi per gli utenti, Microsoft usa due strutture di nominazione distribuita: i **domini** costituiscono la vecchia tecnologia di nominazione; la nuova tecnologia, disponibile nei sistemi operativi Windows 2000 e Windows XP, è l'**active directory**. Una volta impostata, la funzione di nominazione è usata per autenticare gli utenti da tutti i client e da tutti i server.

L'industria si sta orientando verso il protocollo LDAP (*lightweight directory-access protocol*) come meccanismo sicuro per la nominazione distribuita. Lo stesso *active directory* è basato sull'LDAP. Sun Microsystems permette l'uso del protocollo LDAP per l'autenticazione degli utenti e per altri servizi di ricerca di informazioni a livello dell'intero sistema, ad esempio tutte le stampanti disponibili. È pensabile che, se la convergenza sull'uso dell'LDAP avrà successo, un'organizzazione potrà usare una singola directory LDAP distribuita per memorizzare le informazioni su tutti gli utenti e le risorse di tutti i calcolatori dell'organizzazione stessa. Si avrebbe un **unico punto d'accesso sicuro** per gli utenti, che inserirebbero una sola volta le proprie informazioni di autenticazione per avere accesso a tutti i calcolatori dell'organizzazione. Questa soluzione semplificherebbe anche i compiti degli amministratori di sistema, concentrando in un unico punto informazioni ora sparse in vari file in ciascun sistema o in diversi servizi di informazione distribuiti.

### 10.5.2.3 Malfunzionamenti

I file system locali possono presentare malfunzionamenti per varie cause: problemi dei dischi che li contengono, alterazione dei dati relativi alle strutture delle directory o a informazioni necessarie alla gestione dei dischi (chiamate collettivamente **metadati**), malfunzionamenti dei controllori dei dischi, problemi ai cavi di connessione o agli adattatori. Anche il comportamento involontario degli utenti o dell'amministratore di sistema può causare la perdita di file, d'intero directory o addirittura la cancellazione di volumi. Molte di queste cause di malfunzionamento portano al crollo del sistema (*crash*), all'emissione di una condizione d'errore e alla necessità di un intervento umano per risolvere il problema.

L'uso di file system remoti implica maggiori possibilità di malfunzionamenti; a causa della complessità dei sistemi di rete e della necessità di interazioni tra calcolatori remoti, i problemi che possono interferire con il corretto funzionamento dei file system remoti sono infatti molto più numerosi. Nel caso delle reti, si possano verificare interruzioni del collegamento tra due calcolatori, dovute a malfunzionamenti o a improprie configurazioni dell'architettura, oppure a questioni relative alla realizzazione degli aspetti di rete in uno dei calcolatori coinvolti. Sebbene alcune reti includano meccanismi di tolleranza ai guasti, compresi cammini multipli tra ogni coppia di calcolatori, altre non li prevedono. Qualsiasi malfunzionamento potrebbe interrompere il flusso dei comandi del DFS.

Si consideri un client mentre usa un file system remoto. Il client ha il file system remoto montato nel proprio file system e potrebbe avere qualche file remoto aperto; tra le va-

rie attività potrebbe richiedere elenchi dei file nelle directory remote per aprire quelli necessari, svolgere operazioni di lettura e scrittura e chiudere i file. Si consideri ora un malfunzionamento della rete, un crollo del server remoto, oppure anche uno spegnimento programmato di quel server, tali da determinare l'inaccessibilità del file system remoto. Questo scenario è piuttosto comune, quindi il client non dovrebbe comportarsi come nel caso di una perdita del file system locale. Piuttosto, il sistema dovrebbe terminare tutte le operazioni sul server non più raggiungibile, oppure posticiparle finché il server sarà nuovamente disponibile. Questa semantica di trattamento dei malfunzionamenti si definisce e si realizza come parte del protocollo di un file system remoto. La terminazione di tutte le operazioni può portare alla perdita di dati (e della pazienza) da parte degli utenti. La maggior parte dei protocolli DFS impone o permette la posticipazione delle operazioni sul file system remoto, con la speranza che il calcolatore remoto diventi nuovamente disponibile in breve tempo.

Per realizzare questo tipo di recupero dai malfunzionamenti è necessario mantenere alcune **informazioni di stato** sia sui client sia sui server. Se sia i client sia i server tengono traccia delle loro attività correnti e dei loro file aperti, entrambi possono riprendersi senza trarre da un malfunzionamento. Nel caso in cui il server "crolli" ma debba rilevare la presenza di file system remoti e file aperti, l'NFS segue un criterio semplice realizzando un **DFS senza stato**. Sostanzialmente, assume che una richiesta di un client per la lettura o scrittura di un file non sia avvenuta, sempre che il file system non sia stato montato in modo remoto e il file in questione aperto prima della richiesta. Il protocollo NFS trasferisce tutte le informazioni necessarie per localizzare il file appropriato e per svolgere l'operazione richiesta sul file. Allo stesso modo, non tiene traccia di quali client abbiano montato i propri volumi esportati, assumendo anche in questo caso che, se perviene una richiesta, debba essere legittima. Sebbene questo metodo senza stato renda l'NFS tollerante ai guasti e sia piuttosto facile da realizzare, lo rende insicuro. Un server NFS potrebbe ad esempio permettere richieste contrattate di lettura o scrittura da un client non autorizzato anche se la necessaria operazione di montaggio e il controllo dei permessi richiesti non sono avvenuti. Questioni del genere sono regolamentate dallo standard industriale NFS versione 4, in cui NFS è dotato di stato per migliorare le proprie funzionalità, prestazioni e sicurezza.

### 10.5.3 Semantica della coerenza

La **semantica della coerenza** è un importante criterio per la valutazione di qualsiasi file system che consenta la condivisione dei file. Si tratta di una caratterizzazione del sistema che specifica la semantica delle operazioni in cui più utenti accedono contemporaneamente a un file condiviso. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere osservate da altri utenti. La semantica è tipicamente realizzata come codice facente parte del codice del file system.

La semantica della coerenza è direttamente correlata agli algoritmi di sincronizzazione dei processi del Capitolo 6. Tuttavia, a causa delle lunghe latenze e delle basse velocità di trasferimento dei dischi e delle reti, i complessi algoritmi descritti in tale capitolo di solito non s'impiegano per l'I/O su file. Ad esempio, l'esecuzione di una transazione atomica su dischi remoti può coinvolgere molte comunicazioni di rete e molte letture e scritture nei dischi. I sistemi che tentano una così completa serie di funzioni tendono ad avere scarse prestazioni. Una realizzazione riuscita di semantica della condivisione si trova nel file system del sistema Andrew.

Nella trattazione seguente si suppone che una serie d'accessi, cioè letture e scritture, tentati da un utente allo stesso file sia sempre compresa tra una coppia di operazioni `open()` e `close()`. Tale serie d'accessi si chiama **sessione di file**. Per illustrare questo concetto si descrivono sommariamente qui di seguito alcuni esempi di semantica della coerenza.

### 10.5.3.1 Semantica UNIX

Il file system di UNIX, descritto nel Capitolo 17, usa la seguente semantica della coerenza.

- ◆ Le scritture in un file aperto da parte di un utente sono immediatamente visibili ad altri utenti che hanno aperto contemporaneamente lo stesso file.
- ◆ Esiste un metodo di condivisione in cui gli utenti condividono il puntatore alla localizzazione corrente nel file, quindi l'avanzamento del puntatore da parte di un utente influenza su tutti gli utenti che condividono il file. In questo caso il file ha una singola immagine e tutti gli accessi si alternano (intercalandosi) a prescindere dalla loro origine.

Nella semantica UNIX un file è associato a una singola immagine fisica, accessibile come una risorsa esclusiva. La contesa di quest'immagine singola determina il differimento dei processi utenti.

### 10.5.3.2 Semantica delle sessioni

Il file system Andrew (*Andrew file system*, AFS), descritto nel Capitolo 17, usa la seguente semantica della coerenza:

- ◆ le scritture in un file aperto da un utente non sono visibili immediatamente ad altri utenti che hanno aperto contemporaneamente lo stesso file;
- ◆ una volta chiuso il file, le modifiche apportate sono visibili solo nelle sessioni che iniziano successivamente. Le istanze del file già aperte non riportano queste modifiche.

Secondo questa semantica, un file può essere temporaneamente associato a parecchie immagini, probabilmente diverse. Di conseguenza, più utenti possono eseguire accessi concorrenti di lettura o scrittura sulla rispettiva immagine del file senza subire ritardi. Non si impone quasi alcun vincolo nella gestione degli accessi.

### 10.5.3.3 Semantica dei file condivisi immutabili

Un altro metodo è quello dei file condivisi immutabili; una volta che un file è stato dichiarato *condiviso* dal suo creatore, non può essere modificato. Un file immutabile presenta due caratteristiche chiave: il suo nome non si può riutilizzare e il suo contenuto non può essere modificato. Quindi il nome di un file immutabile indica che i contenuti di quel file sono fissi. Come si descrive nel Capitolo 17, la realizzazione di questa semantica in un sistema distribuito è semplice; infatti la condivisione è molto disciplinata poiché consente la sola lettura.

## 10.6 Protezione

La salvaguardia delle informazioni contenute in un sistema di calcolo dai danni fisici (la questione della *affidabilità*) e da accessi impropri (la questione della *protezione*) è fondamentale.

Generalmente l'affidabilità è assicurata da più copie dei file. Molti calcolatori hanno programmi di sistema che copiano i file dai dischi ai nastri a intervalli regolari, ad esempio una volta al giorno, alla settimana o al mese; quest'operazione di copiatura può essere automatica, o controllata dall'intervento di un operatore. Lo scopo è quello di conservare copie di riserva utili nei casi in cui il file system andasse distrutto a causa di problemi dei dispositivi: errori di lettura o scrittura, cali o cadute della tensione elettrica, rotture delle testine, sporcizia, temperature estreme, atti vandalici, ecc. I file possono inoltre essere cancellati ac-

cidentalmente, e anche errori di programmazione possono causare la perdita del contenuto dei file. L'affidabilità è trattata con maggiori dettagli nel Capitolo 12.

La protezione si può ottenere in molti modi. Per un piccolo sistema monoutente, la protezione si ottiene rimovendo fisicamente i dischetti e chiudendoli in un cassetto della scrivania oppure in un armadietto. In un sistema multiutente sono necessari altri metodi.

## 10.6.1 Tipi d'accesso

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. I sistemi che non permettono l'accesso ai file di altri utenti non richiedono protezione; quindi si può ottenere una completa protezione proibendo l'accesso. In alternativa si può permettere un accesso totalmente libero senza alcuna protezione. Questi orientamenti sono entrambi eccessivi, quindi non si possono applicare in generale; ciò che serve in realtà è un **accesso controllato**.

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi d'accesso. Gli accessi si permettono o si negano secondo diversi fattori, innanzitutto i tipi d'accesso richiesti. Si possono controllare distinte operazioni.

- ◆ **Lettura.** Lettura da file.
- ◆ **Scrittura.** Scrittura o riscrittura di file.
- ◆ **Esecuzione.** Caricamento di file in memoria ed esecuzione.
- ◆ **Aggiunta.** Scrittura di nuove informazioni in coda ai file.
- ◆ **Cancellazione.** Cancellazione di file e liberazione del relativo spazio per un possibile riutilizzo.
- ◆ **Elencazione.** Elencazione del nome e degli attributi dei file.

Si possono controllare anche altre operazioni, come ridenominazione, copiatura o modifica dei file. Tuttavia, in molti sistemi queste funzioni di livello superiore si possono realizzare tramite un programma di sistema che compie alcune chiamate di sistema di livello inferiore, quindi è sufficiente garantire la protezione a livello inferiore. Ad esempio, la copiatura di un file si può realizzare semplicemente con una sequenza di richieste di lettura; in questo caso un utente con accesso per la lettura di un file può richiederne la copiatura, la stampa o altro.

Sono stati proposti molti meccanismi di protezione. Come sempre, ogni meccanismo presenta vantaggi e svantaggi, e deve essere appropriato alla particolare applicazione che richiede la protezione. Un piccolo calcolatore usato soltanto da pochi membri di un gruppo di ricerca non richiede la stessa protezione del sistema di calcolo di una grande società, usato per operazioni di ricerca, finanza e per il lavoro del personale. Il problema della protezione è trattato nel Capitolo 14.

## 10.6.2 Controllo degli accessi

Il problema della protezione comunemente si affronta rendendo l'accesso dipendente dall'identità dell'utente. Più utenti possono richiedere diversi tipi d'accesso a un file o a una directory. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una **lista di controllo degli accessi** (*access-control list*, ACL) a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti. Quando un utente richiede un accesso a un file specifico il sistema operativo esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista si autorizza l'accesso, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

Questo sistema ha il vantaggio di permettere complessi metodi d'accesso. Il problema maggiore delle liste di controllo degli accessi è la loro lunghezza: per permettere a tutti di leggere un file, la lista deve contenere tutti gli utenti con accesso per la lettura. Questa tecnica comporta due inconvenienti:

- ◆ la costruzione di una lista di questo tipo può essere un compito noioso e non gratificante, soprattutto se la lista degli utenti del sistema non è già nota;
- ◆ l'elemento della directory, precedentemente di dimensione fissa, richiede di essere di dimensione variabile, quindi anche la gestione dello spazio è più complicata.

Questi problemi si possono risolvere introducendo una versione condensata della lista di controllo degli accessi.

Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte.

- ◆ **Proprietario.** È l'utente che ha creato il file.
- ◆ **Gruppo.** Si tratta di un insieme di utenti che condividono il file e richiedono tipi di accesso simili.
- ◆ **Universo.** Tutti gli altri utenti del sistema.

Il più comune orientamento recente prevede la combinazione delle liste di controllo degli accessi con lo schema di controllo degli accessi per proprietario, gruppo e universo (più facile da realizzare). Il sistema operativo Solaris 2.6 (e le sue versioni successive) impiega le tre categorie d'accesso in modo predefinito ma, se si vuole una maggiore selettività del controllo degli accessi, permette l'attribuzione di liste di controllo degli accessi a specifici file e directory.

Si consideri, ad esempio, una persona, Donatella, che sta scrivendo un nuovo libro. Donatella ha assunto tre studenti, Giulia, Paolo e Carlo, per aiutarla a lavorare al progetto. Il testo del libro è tenuto in un file chiamato `libro`. La protezione associata a tale file prevede quel che segue:

- ◆ Donatella può compiere tutte le operazioni sul file;
- ◆ Giulia, Paolo e Carlo possono solo leggere e scrivere il file ma non possono cancellarlo;
- ◆ tutti gli altri utenti possono leggere, ma non scrivere, il file (Donatella ha interesse che il libro sia letto dal maggior numero possibile di persone, in modo da ottenere pareri competenti).

Per ottenere tale protezione si deve creare un nuovo gruppo composto da Giulia, Paolo e Carlo, e si deve associare il nome del gruppo, ad esempio `testo`, al file `libro` con i diritti d'accesso conformi al criterio descritto.

Si consideri un ospite cui Donatella vorrebbe concedere un accesso temporaneo al Capitolo 1. L'ospite non si può aggregare al gruppo `testo` poiché ciò gli darebbe accesso a tutti i capitoli, e considerato che i file possono essere in un solo gruppo, non si può associare un altro gruppo al Capitolo 1. Laggiunta della funzione delle liste di controllo degli accessi permette di inserire l'ospite nella lista di controllo degli accessi del Capitolo 1.

Affinché questo schema funzioni correttamente, è necessario uno stretto controllo dei permessi e delle liste di controllo degli accessi, fattibile in diversi modi. Nel sistema UNIX ad esempio solo un utente con compiti di gestione (o un *superuser*) può creare e modificare i gruppi, quindi questo controllo si ottiene con la partecipazione umana. Nel sistema VMS, il proprietario del file può creare e modificare tale lista. Le liste di controllo degli accessi sono trattate anche nel Paragrafo 14.5.2.

Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi. Ogni campo è formato di un insieme di bit, ciascuno dei quali permette o impedisce l'accesso che gli è associato. Nel sistema UNIX, ad esempio, sono definiti tre campi di tre bit ciascuno: **rwx**, dove **r** controlla l'accesso per la lettura, **w** quello per la scrittura e **x** per l'esecuzione. Un campo distinto è riservato al proprietario del file, al gruppo proprietario e a tutti gli altri utenti. In questo schema, per registrare le informazioni di protezione sono necessari nove bit per file. Così, nell'esempio, i campi di protezione per il file **libro** s'impostano come segue: per Donatella, il proprietario, tutti e tre i bit; per il gruppo **testo** i bit **r** e **w**; per l'universo il solo bit **r**.

Nel combinare i due metodi si presenta una difficoltà nell'interfaccia utente; gli utenti devono poter indicare l'impostazione opzionale dei permessi nella lista di controllo degli accessi per un file. Nell'esempio di Solaris, un segno “+” aggiunge un permesso d'accesso:

```
19 -rw--r--r--+ 1 alessandra gruppo 130 May 25 22:13 file1
```

Una specifica serie di comandi **setfacl** e **getfacl** si usa per gestire le liste di controllo degli accessi.

Gli utenti di Windows XP, in genere, gestiscono le liste di controllo degli accessi tramite un'interfaccia grafica. La Figura 10.15 mostra la finestra dei permessi di un file del file

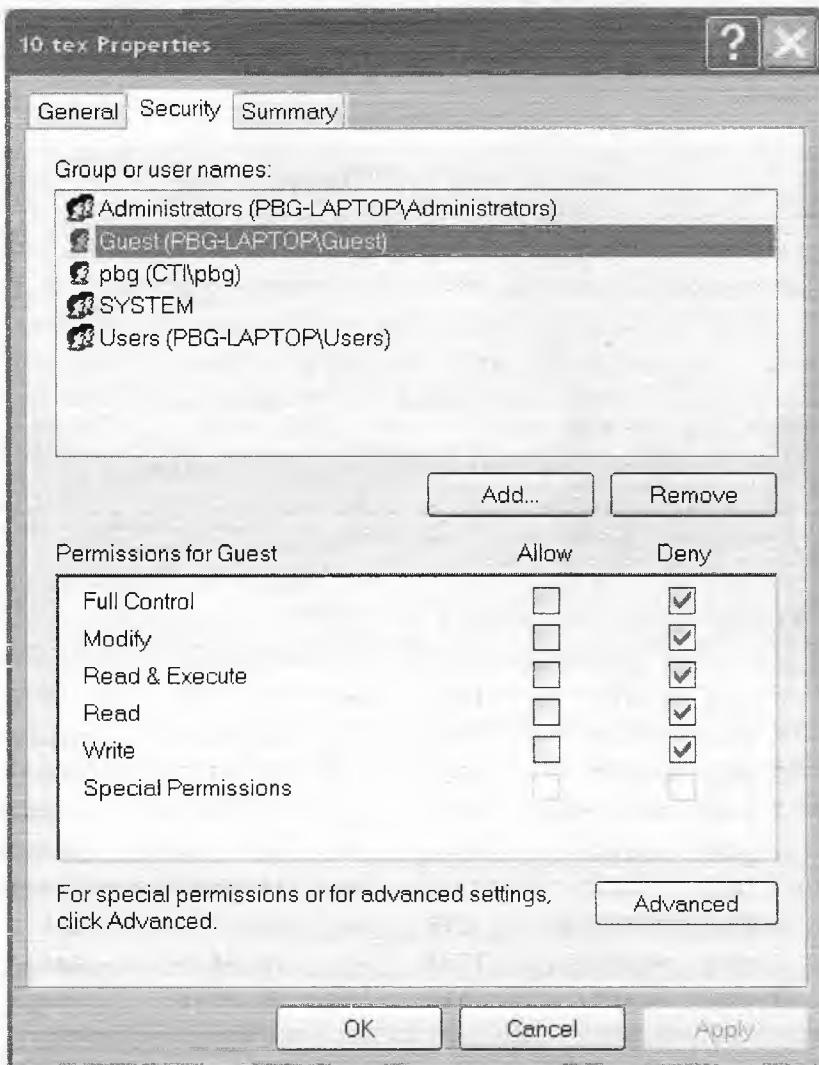


Figura 10.15 Gestione della lista di controllo degli accessi in Windows XP

system NTFS di Windows XP. In questo esempio si vieta esplicitamente all'utente Guest l'accesso al file `10.tex`.

Un'altra difficoltà s'incontra nell'assegnazione delle precedenze quando ci sono conflitti tra i permessi e liste di controllo degli accessi. Ad esempio, se Andrea è in un gruppo di file, che ha il permesso di lettura, ma il file ha un lista di controllo degli accessi contenente i permessi di lettura e scrittura per Andrea, si pone il problema della concessione del permesso di scrittura. Nel sistema operativo Solaris si attribuiscono i permessi contenuti nelle liste di controllo degli accessi; sono più selettivi e non sono predefiniti. Si segue il principio che la maggiore specificità deve essere prioritaria.

### 10.6.3 Altri metodi di protezione

Un altro metodo di protezione consiste nell'associazione di una parola d'ordine (*password*) a ciascun file. Proprio come l'accesso al sistema di calcolo è spesso controllato da una parola d'ordine, anche l'accesso a ogni file può avere lo stesso tipo di protezione. Se le parole d'ordine sono scelte a caso e si cambiano spesso, questo schema può essere efficace, poiché limita l'accesso a un file agli utenti che conoscono la parola d'ordine. Questo presenta ha tuttavia diversi svantaggi: innanzitutto, il numero di parole d'ordine da ricordare può diventare molto alto, rendendo tale metodo impraticabile; secondariamente, se si impiega la stessa parola d'ordine per tutti i file, la sua scoperta li rende tutti accessibili. La protezione è basata sul principio del "o tutto o niente". Per risolvere questo problema alcuni sistemi, ad esempio il TOPS-20, permettono a un utente di associare una parola d'ordine a una directory anziché

#### PERMESSI IN UN SISTEMA UNIX

Nel sistema UNIX la protezione delle directory è gestita in modo simile alla protezione dei file. In altre parole, a ciascuna directory sono associati tre campi (proprietario, gruppo e universo), ciascuno composto dei tre bit `rwx`. Quindi, un utente può elencare il contenuto di una directory solamente se il bit `r` è inserito nel campo appropriato. Analogamente, un utente può modificare la propria directory corrente in un'altra directory (ad esempio `foo`) solo se è associato, nel campo appropriato, il bit di `foo`.

Nella Figura 10.16 è illustrato un esempio di elenco del contenuto di una directory nell'ambiente UNIX. Il primo campo descrive le protezioni di file e directory, il carattere `d` presente all'inizio del campo contraddistingue le directory; inoltre, l'elenco contiene il numero di collegamenti relativi al file, il nome del proprietario e del gruppo, la dimensione del file in byte, la data di creazione e infine il nome del file (con l'eventuale estensione).

<code>-rw-rw-r--</code>	1	pbg	staff	31200	set	3	08:30	intro.ps
<code>drwx-----</code>	5	pbg	staff	512	lug	8	09:33	privato/
<code>drwxrwxr-x</code>	2	pbg	staff	512	lug	8	09:35	doc/
<code>drwxrwx---</code>	2	pbg	studente	512	ago	3	14:13	studente-prog/
<code>-rw-r--r--</code>	1	pbg	staff	9423	feb	24	2003	program.c
<code>-rwxr-xr-x</code>	1	pbg	staff	20471	feb	24	2003	program
<code>drwx--x--x</code>	4	pbg	facoltà	512	lug	31	10:31	lib/
<code>drwx-----</code>	3	pbg	staff	1024	ago	29	06:52	mail/
<code>drwxrwxrwx</code>	3	pbg	staff	512	lug	8	09:35	test/

Figura 10.16 Esempio di elenco del contenuto di una directory.

a un singolo file. Il sistema operativo IBM VM/CMS consente di associare tre parole d'ordine ai minidischi: una per la lettura, una per la scrittura, e una per gli accessi multipli per la scrittura.

Alcuni sistemi operativi monoutente come MS-DOS e le versioni del sistema Macintosh prima di Mac OS X, non offrono grande protezione dei file. Nel caso in cui tali sistemi più antiquati siano posti in reti dove sia necessario condividere file, occorre aggiungere loro dei meccanismi di protezione. Progettare una caratteristica di un nuovo sistema operativo è quasi sempre più facile che aggiungerla a uno esistente. Inoltre, tali aggiornamenti sono di solito meno efficaci e si integrano meno bene nel sistema.

In una struttura della directory a più livelli è necessario proteggere non solo i singoli file, ma anche gruppi di file contenuti in directory; è cioè necessario disporre di un meccanismo per la protezione delle directory. Le operazioni riguardanti le directory da proteggere sono piuttosto diverse dalle operazioni sui file. Esse sono la creazione e la cancellazione dei file in una directory; probabilmente va anche controllata la possibilità, per un utente, di determinare l'esistenza di un file in una directory. Talvolta la conoscenza dell'esistenza di un file e del suo nome può essere di per sé significativa, perciò l'elencazione del contenuto di una directory dev'essere un'operazione protetta. Quindi, affinché un nome di percorso possa far riferimento a un file in una certa directory, all'utente deve essere consentito l'accesso sia al file sia alla directory. Nei sistemi dove i file possono avere numerosi nomi di percorso (come quelli con struttura a grafo aciclico o a grafo generale) un certo utente può avere diversi diritti d'accesso secondo il nome di percorso di cui fa uso.

## 10.7 Sommario

Un file è un tipo di dati astratto definito e realizzato dal sistema operativo. È una sequenza di elementi logici (o *record*), ciascuno dei quali può essere un byte, una riga di lunghezza fissa o variabile, oppure un elemento di dati più complesso. Il sistema operativo può gestire in modo specifico diversi tipi di elementi logici o può lasciare tale gestione al programma applicativo.

Il compito più importante del sistema operativo consiste nell'associare i file ai dispositivi fisici di memorizzazione, ad esempio dischi o nastri magnetici. Poiché le dimensioni dei blocchi fisici dei dispositivi possono non coincidere con quelle dei record logici, può essere necessario riunire un certo numero di record logici in modo da ottenere una dimensione pari a quella di un blocco fisico. Anche questo compito può essere gestito dal sistema operativo, oppure lasciato al programma applicativo.

Ciascun dispositivo di un file system conserva una tabella dei contenuti del volume o una directory di dispositivo che elenca la locazione dei file presenti nel dispositivo. Inoltre, è utile creare directory per permettere di organizzare i file. Una directory a livello singolo in un sistema multiutente causa problemi di nominazione, poiché ogni file deve avere un nome unico. Una directory a due livelli limita questo problema all'ambito relativo a ciascun utente creando una directory distinta per ciascun utente, che dispone di una directory contenente i suoi file. Le directory contengono la lista dei file che le costituiscono, insieme con informazioni a essi associate: locazione nei dischi, lunghezza, tipo, ora di creazione, ora dell'ultimo uso, e così via.

La naturale generalizzazione del concetto di directory a due livelli è la directory con struttura ad albero. Tale tipo di struttura permette a un utente di creare sottodirectory in cui organizzare i file. Le strutture delle directory a grafo aciclico permettono la condivisione di sottodirectory e file, ma complicano le funzioni di ricerca e cancellazione. Una struttura a

grafo generale permette la massima flessibilità nella condivisione dei file e delle directory, ma talvolta richiede operazioni di “ripulitura” (*garbage collection*) per recuperare lo spazio inutilizzato nei dischi.

I dischi sono suddivisi in uno o più volumi, ciascuno contenente un file system o privo di struttura. Per essere resi disponibili, i file system si possono montare nelle strutture di nominazione del sistema. Lo schema di nominazione varia da sistema a sistema. Una volta montati, i file all’interno del volume sono disponibili per l’uso. I file system si possono smontare per disabilitarne l’accesso o per attività di manutenzione.

La condivisione dei file dipende dalla semantica definita dal sistema. I file possono avere più lettori, più scrittori, o limiti alla condivisione. I file system distribuiti consentono ai sistemi client di montare volumi o directory di server, man mano che vi accedono tramite una rete. I file system remoti pongono delle sfide in termini di affidabilità, prestazioni e sicurezza. I sistemi informativi distribuiti mantengono informazioni su utenti, calcolatori e accessi in modo che i client e i server condividano le informazioni di stato per la gestione dell’uso e degli accessi.

Poiché i file rappresentano il principale meccanismo di memorizzazione delle informazioni, è necessario che siano dotati di un sistema di protezione. Ogni tipo d’accesso ai file si può controllare separatamente: lettura, scrittura, esecuzione, aggiunta, elencazione del contenuto di directory, e così via. La protezione dei file si può ottenere con parole d’ordine, liste d’accesso, oppure tecniche appositamente predisposte per le situazioni specifiche.

## Esercizi pratici

- 10.1 Alcuni sistemi cancellano automaticamente tutti i file utente quando un utente si disconnette oppure quando un lavoro termina, a meno che l’utente non richieda esplicitamente che questi vengano conservati; altri sistemi conservano tutti i file a meno che l’utente non li cancelli esplicitamente. Discutete i meriti di ciascun approccio.
- 10.2 Perché alcuni sistemi tengono traccia del tipo di un file, mentre altri lasciano questo compito all’utente e altri semplicemente non implementano tipi di file multipli? Quale sistema è “migliore”?
- 10.3 In modo simile, alcuni sistemi mettono a disposizione molte tipologie diverse di strutture per i dati di un file, mentre altri trattano solo un semplice flusso di byte. Quali sono i vantaggi e gli svantaggi di tali approcci?
- 10.4 È possibile simulare una struttura di directory multilivello con una struttura di directory a livello singolo nella quale possono essere usati nomi arbitrariamente lunghi? Se la risposta è affermativa, spiegate come ciò sia fattibile e confrontate questo schema con lo schema a directory multilivello. In caso di risposta negativa, spiegate che cosa impedisce il successo della simulazione. Come cambierebbe la vostra risposta se la lunghezza del nome dei file fosse limitata a sette caratteri?
- 10.5 Spiegate lo scopo delle operazioni `open()` e `close()`.
- 10.6 Fornite un esempio di applicazione nella quale si debba accedere ai dati in un file nell’ordine seguente:
  - a. sequenziale;
  - b. casuale.

- 10.7 In alcuni sistemi una sottodirectory può essere letta e scritta da un utente autorizzato, proprio come i file ordinari.
- Descrivete i problemi di protezione che ne potrebbero derivare.
  - Suggerite uno schema per trattare ognuno di questi problemi di protezione.
- 10.8 Considerate un sistema che supporti 5.000 utenti. Supponete di voler permettere a 4.990 utenti di accedere a un dato file.
- Come specifichereste questo schema di protezione in UNIX?
  - Potete suggerire un altro schema di protezione utilizzabile a questo scopo più efficacemente dello schema fornito da UNIX?
- 10.9 Alcuni ricercatori hanno suggerito che, invece di associare una lista di accesso a ogni file (dove la lista specifica quali utenti possono accedere al file e come), dovremmo avere *una lista di controllo degli utenti* associata a ogni utente (dove la lista specifica a quali file un utente può accedere e come). Discutete i meriti relativi a questi due schemi.

## Esercizi

- 10.10 Considerate un file system in cui si può cancellare un file e reclamare il suo spazio di memoria secondaria mentre esistono ancora collegamenti (*link*) a esso. Dite quale problema si può presentare se si crea un nuovo file nella stessa area di memoria o con lo stesso nome di percorso assoluto. Spiegate come tali problemi siano evitabili.
- 10.11 La tabella dei file aperti registra le informazioni riguardanti i file aperti in quel momento. Il sistema operativo dovrebbe mantenere una tabella separata per ciascun utente oppure mantenere una tabella unica per tutti gli utenti con le indicazioni sui file a cui accedono in un certo momento? Se due diversi programmi o utenti eseguono accessi al medesimo file, questi dovrebbero comparire come accessi separati nella tabella dei file aperti?
- 10.12 Quali sono vantaggi e svantaggi di un sistema che fornisce lock obbligatori anziché lock consigliati, il cui utilizzo è rimesso al giudizio degli utenti?
- 10.13 Spiegate quali sono vantaggi e svantaggi della registrazione, tra gli attributi di un file, del nome del programma che crea il file stesso, come avviene nel sistema operativo Macintosh.
- 10.14 Alcuni sistemi aprono un file automaticamente quando ci si riferisce a esso per la prima volta, e lo chiudono al termine del lavoro. Illustrate vantaggi e svantaggi di questo schema, confrontandolo con quello tradizionale, in cui l'utente deve aprire e chiudere il file esplicitamente.
- 10.15 Qualora il sistema operativo dovesse apprendere che una certa applicazione accederà ai dati di un file in modo sequenziale, come potrebbe sfruttare questa informazione per migliorare le prestazioni?
- 10.16 Illustrate un'applicazione che potrebbe trarre vantaggio da un sistema operativo che offre l'accesso casuale ai file indicizzati.
- 10.17 Analizzate vantaggi e svantaggi di permettere collegamenti a file che oltrepassano i punti di montaggio (vale a dire, collegamenti che rimandano a file memorizzati in un volume differente).

- 10.18 Alcuni sistemi consentono la condivisione dei file usando una singola copia di ogni file; altri sistemi impiegano più copie, una per ciascun utente che condivide il file. Discutete i vantaggi di ciascun metodo.
- 10.19 Considerate vantaggi e svantaggi insiti nell'associare a file system remoti (memorizzati su file server) una semantica del fallimento diversa da quella prevista nei file system locali.
- 10.20 Quali sono le implicazioni derivanti dall'abbinare a file residenti su file system remoti la semantica della coerenza di UNIX per l'accesso condiviso?

## 10.8 Note bibliografiche

---

Analisi generali sui file system sono reperibili in [Grosshans 1986]; [Golden e Pechura 1986] descrive la struttura dei file system dei microcalcolatori. I sistemi di gestione delle basi di dati e le loro strutture di file sono ampiamente descritti in [Korth e Silberschatz 2001].

La struttura delle directory a più livelli è stata realizzata per la prima volta nel sistema MULTICS [Organick 1972]. Attualmente la maggior parte dei sistemi dispone di una struttura delle directory a più livelli, ad esempio Linux, [Bovet e Cesati 2002], Mac OS X (<http://www.apple.com/macosx/>), Solaris [McDougal e Mauro 2007] e tutte le versioni di Windows, incluso Windows 2000 [Russinovich e Solomon 2005].

Il Network File System (NFS), progettato dalla Sun Microsystems, permette di distribuire le strutture delle directory sui calcolatori di una rete. NFS è approfondito nei dettagli nel Capitolo 17. La versione 4 di NFS è descritta in RFC3505 (<http://www.ietf.org/rfc/rfc3505.txt>). Una trattazione generale sui file system in Solaris si può trovare nella guida di Sun *System Administration Guide: Devices and File Systems* (<http://docs.sun.com/app/docs/doc/817-5093>).

Inizialmente proposto da [Su 1982], DNS è passato attraverso diverse revisioni; [Mockapetris 1987] lo ha esteso con importanti funzionalità. Più recentemente, [Eastlake 1999] ha proposto estensioni relative alla sicurezza che permettano a DNS di detenere delle chiavi di sicurezza.

Il protocollo LDAP, noto anche come X.509, è un sottoinsieme del protocollo di directory distribuita X.500. È stato definito da [Yeong et al. 1995] e incorporato in molti sistemi operativi.

Sono in corso interessanti ricerche nell'area delle interfacce dei file system, in particolare per i problemi relativi alla nominazione dei file e ai loro attributi. Ad esempio, il sistema operativo Plan 9 dei Bell Laboratories (Lucent Technology) rende tutti gli oggetti simili a file system. In tal modo, per ottenere l'elenco dei processi nel sistema, un utente deve semplicemente leggere il contenuto della directory `/proc`; per veder che ore sono si deve leggere il file `/dev/time`.

## Capitolo 11

# Realizzazione del file system



### OBIETTIVI

- Descrizione dei dettagli realizzativi di file system locali e strutture di directory.
- Analisi della realizzazione di file system remoti.
- Esame dell'allocazione dei blocchi e dei pro e contro degli algoritmi per la gestione dello spazio libero.

Come illustrato nel Capitolo 10, il file system fornisce il meccanismo per la memorizzazione e l'accesso al contenuto dei file, compresi dati e programmi. Il file system risiede permanentemente nella *memoria secondaria*, progettata per contenere in modo permanente grandi quantità di dati. Questo capitolo riguarda principalmente i problemi connessi alla memorizzazione e all'accesso ai file nel più comune mezzo di memoria secondaria, il disco. Si esaminano varie modalità d'uso dei file, l'allocazione dello spazio dei dischi, il recupero dello spazio liberato, la registrazione delle locazioni dei dati, e l'interfaccia di altri componenti del sistema operativo alla memoria secondaria. Nel corso della trattazione si considerano anche i problemi riguardanti le prestazioni.

## 11.1 Struttura del file system

I dischi costituiscono la maggior parte della memoria secondaria in cui si conserva il file system. Hanno due caratteristiche importanti che ne fanno un mezzo conveniente per la memorizzazione dei file:

1. si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione;
2. è possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale sia in modo diretto, e passare da un file all'altro spostando le testine di lettura e scrittura e attendendo la rotazione del disco.

La struttura dei dischi è analizzata in modo particolareggiato nel Capitolo 12.

Anziché trasferire un byte alla volta, per migliorare l'efficienza dell'I/O, i trasferimenti tra memoria centrale e dischi si eseguono per *blocchi*. Ciascun blocco è composto da uno o più settori. Secondo l'unità a disco, la dimensione dei settori è compresa tra 32 byte e 4096 byte; di solito è pari a 512 byte.

Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più **file system** che consentono di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione piuttosto diversi. Il primo riguarda la definizione dell'aspetto del file system agli occhi dell'utente. Questo compito implica la definizione di un file e dei suoi attributi, delle operazioni permesse su un file e della struttura delle directory per l'organizzazione dei file. Il secondo riguarda la creazione di algoritmi e strutture dati che permettano di far corrispondere il file system logico ai dispositivi fisici di memoria secondaria.

Lo stesso file system è generalmente composto da molti livelli distinti. La struttura illustrata nella Figura 11.1 è un esempio di struttura stratificata. Ogni livello si serve delle funzioni dei livelli inferiori per crearne di nuove impiegate dai livelli superiori.

Il livello più basso, il *controllo dell'I/O*, costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria. Un driver di dispositivo si può concepire come un traduttore che riceva comandi ad alto livello, come "recupera il blocco 123", e che emette specifiche istruzioni di basso livello per i dispositivi, usate dal controllore che fa da interfaccia tra i dispositivi di I/O e il resto del sistema. Un driver di dispositivo di solito scrive specifiche configurazioni di bit in specifiche locazioni della memoria del controllore di I/O per indicare quali azioni il dispositivo di I/O debba compiere, e in quali locazioni. I dettagli dei driver dei dispositivi e le strutture per l'I/O sono trattati nel Capitolo 13.

Il **file system di base** deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco, ad esempio unità 1, cilindro 73, superficie 2, settore 10. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi dei file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un I/O richiesto. Le cache servono a conservare metadati di file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

Il **modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici, che il file system di base deve trasferire, negli indirizzi dei blocchi fisici. I blocchi logici di ciascun file sono numerati da 0 (o 1) a  $n$ ; i



**Figura 11.1** File system stratificato.

blocchi fisici contenenti tali dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione. Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Infine, il **file system logico** gestisce i metadati; si tratta di tutte le strutture del file system, eccetto gli effettivi dati (il contenuto dei file). Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i **blocchi di controllo dei file** (*file control block*, FCB), contenenti informazioni sui file, come la proprietà, i permessi, e la posizione del contenuto. Come si discute nel Capitolo 10 e Capitolo 14, il file system logico è responsabile anche della protezione e della sicurezza.

Nei file system stratificati la duplicazione di codice è ridotta al minimo. Il controllo dell'I/O e, talvolta, il codice di base del file system, possono essere comuni a numerosi file system, che poi gestiscono il file system logico e i moduli per l'organizzazione dei file secondo le proprie esigenze. Sfortunatamente, la stratificazione può comportare un maggior overhead del sistema operativo, che può generare un conseguente decadimento delle prestazioni. L'utilizzo della stratificazione e le scelte sul numero di strati da impiegare e sulle loro funzionalità rappresentano una grande sfida per la progettazione di nuovi sistemi.

Esistono svariati tipi di file system al giorno d'oggi, e non è raro che i sistemi operativi ne prevedano più d'uno. Molti CD-ROM, a esempio, sono scritti nel formato ISO 9660, uno standard concordato dai produttori di CD-ROM. Oltre ai file system dei supporti rimovibili, ciascun sistema operativo possiede un file system, o più di uno, basato sui dischi. UNIX adotta il **File System UNIX** (UFS), che si fonda a sua volta sul File System Berkeley Fast (FFS). Windows NT, 2000 e XP adottano i formati FAT, FAT32 e NTFS (o File System di Windows NT), così come i formati per CD-ROM, DVD e dischetti. Sebbene Linux possa funzionare con più di quaranta file system diversi, quello standard è noto come **file system esteso**, le cui versioni maggiormente diffuse sono ext2 ed ext3. Esistono anche file system distribuiti, ossia tali che un file system del server è montato da uno o più client in una rete.

La ricerca relativa ai file system continua a essere un'area attiva della progettazione e dell'implementazione dei sistemi operativi. Google ha progettato un proprio file system per soddisfare esigenze di memorizzazione e recupero dati specifiche dell'azienda. Un altro progetto interessante è il file system FUSE, che garantisce flessibilità nell'utilizzo del sistema grazie all'implementazione e all'esecuzione di file system a livello utente invece che a livello del codice del kernel. Gli utenti di FUSE possono aggiungere nuovi file system a numerosi sistemi operativi e utilizzarli per gestire i propri file.

## 11.2 Realizzazione del file system

Per permettere ai processi di richiedere l'accesso al contenuto dei file, i sistemi operativi offrono le chiamate di sistema `open()` e `close()`. In questo paragrafo si approfondiscono le strutture dati e le operazioni usate per realizzare le funzioni del file system.

### 11.2.1 Introduzione

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano secondo il sistema operativo e il file system, ma esistono dei principi generali. Nei dischi, il file system tiene informazioni su come eseguire l'avviamento di

un sistema operativo memorizzato nei dischi stessi, il numero totale di blocchi, il numero e la locazione dei blocchi liberi, la struttura delle directory e i singoli file. Molte di loro sono analizzate in modo particolareggiato nel seguito di questo capitolo.

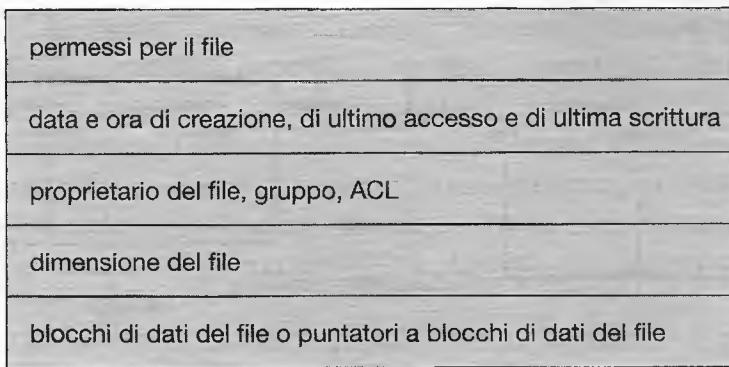
Fra le strutture presenti nei dischi ci sono le seguenti.

- ◆ Il **blocco di controllo dell'avviamento** (*boot control block*), contenente le informazioni necessarie al sistema per l'avviamento di un sistema operativo da quel volume; se il disco non contiene un sistema operativo, tale blocco è vuoto. Di solito è il primo blocco di un volume. Nell'UFS, si chiama **blocco d'avviamento** (*boot block*); nell'NTFS, **settore d'avviamento della partizione** (*partition boot sector*).
- ◆ I **blocchi di controllo dei volumi** (*volume control block*); ciascuno di loro contiene i dettagli riguardanti il relativo volume (o partizione), come il numero e la dimensione dei blocchi nel disco, il contatore dei blocchi liberi e i relativi puntatori, il contatore dei blocchi di controllo dei file liberi e i relativi puntatori. Nell'UFS si chiama **superblocco**; nell'NTFS si chiama **tabella principale dei file** (*master file table*, MFT).
- ◆ Le **strutture delle directory** (una per file system) usate per organizzare i file. Nel caso dell'UFS comprendono i nomi dei file e i numeri di **inode** associati. Nel caso dell'NTFS sono memorizzate nella **tabella principale dei file** (*master file table*).
- ◆ I **blocchi di controllo di file** (FCB), contenenti molti dettagli dei file, compresi i permessi d'accesso ai relativi file, i proprietari, le dimensioni e le locazioni dei blocchi di dati. Nell'UFS si chiamano *inode*; nell'NTFS, queste informazioni sono memorizzate all'interno della tabella principale dei file, che si serve di una struttura di base di dati relazionale, con una riga per ciascun file.

Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l'uso di cache. I dati si carico al momento del montaggio e si eliminano allo smontaggio. Le strutture che vi possono essere incluse sono di diverso tipo:

- ◆ la tabella di montaggio interna alla memoria che contiene informazioni relative a ciascun volume montato;
- ◆ la struttura della directory, tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (per le directory che costituiscono dei punti di montaggio, può essere presente un puntatore alla tabella dei volumi);
- ◆ la tabella generale dei file aperti, contenente una copia del blocco di controllo del file per ciascun file aperto, insieme con altre informazioni;
- ◆ la tabella dei file aperti per ciascun processo, contenente un puntatore all'appropriato elemento della tabella generale dei file aperti, insieme con altre informazioni;
- ◆ i buffer conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Esso crea e alloca un nuovo FCB. (In alternativa, nel caso dei file system che creano tutti i blocchi di controllo al momento della loro installazione, esso alloca semplicemente un blocco di controllo libero.) Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con il blocco di controllo associato, e la scrive nuovamente sul disco. Una tipica struttura di blocco di controllo dei file (FCB) è illustrata nella Figura 11.2.



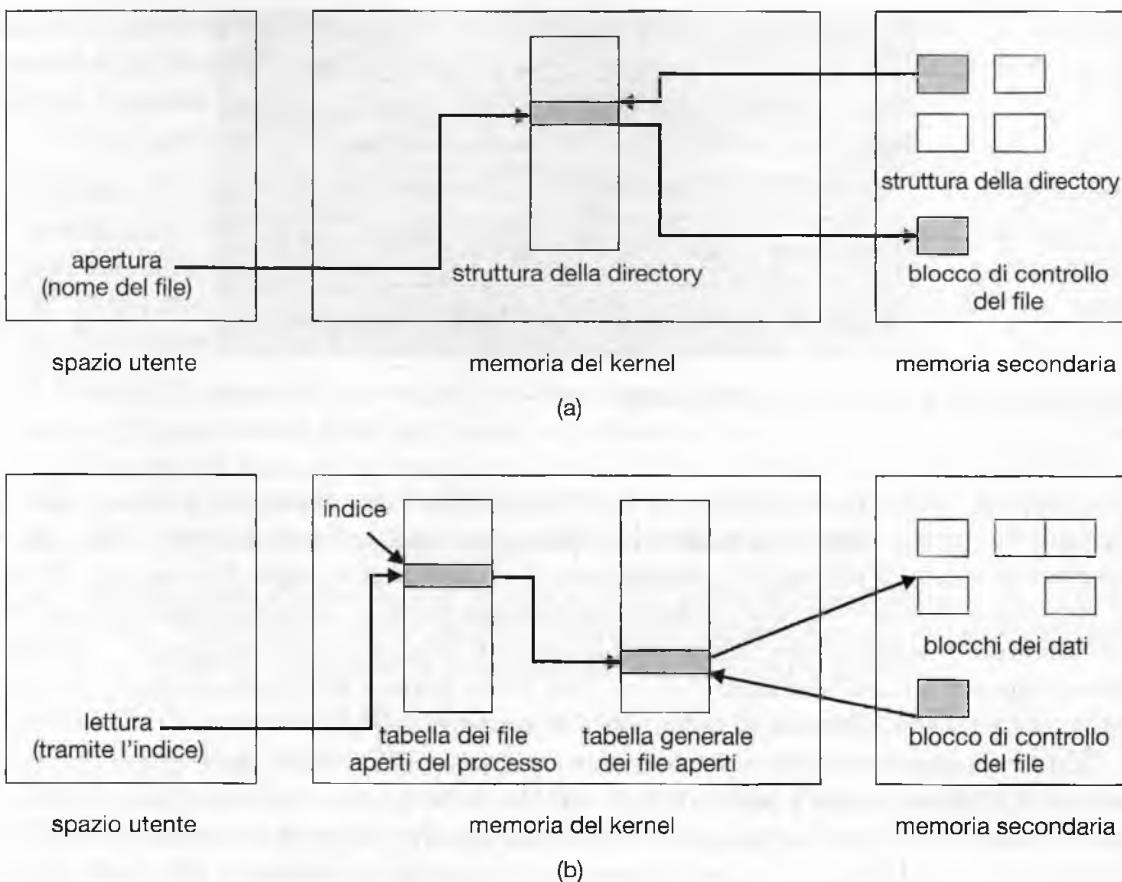
**Figura 11.2** Tipico blocco di controllo dei file.

Alcuni sistemi operativi, compreso UNIX, trattano le directory esattamente come i file, distinguendole con un campo per il tipo che indica che si tratta di una directory. Altri, tra cui il sistema operativo Windows NT, dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file. Indipendentemente da queste questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'I/O su directory a numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'I/O.

Una volta creato un file, per essere usato per operazioni di I/O deve essere *aperto*. La chiamata di sistema `open()` passa un nome di file al file system. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata `open()` dapprima esamina la tabella dei file aperti in tutto il sistema; in caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo (per ogni processo che stia usando il file) che punta alla tabella dei file aperti in tutto il sistema. L'algoritmo può eliminare significativi rallentamenti. Una volta aperto il file, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'FCB nella tabella generale dei file aperti, tenuta in memoria. Questa tabella non solo contiene l'FCB, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.

Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella generale e con alcuni altri campi. Questi altri campi possono comprendere un puntatore alla posizione corrente nel file (per successive operazioni `read()` o `write()`) e il tipo d'accesso richiesto all'apertura del file. La `open()` riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore. Il nome del file potrebbe non essere contenuto nella tabella dei file aperti, visto che, una volta che l'FCB appropriato è stato individuato nei dischi, il sistema non ne ha bisogno. Tuttavia, potrebbe venir memorizzato in una cache per risparmiare tempo sulle aperture successive dello stesso file. Il nome dato all'elemento della tabella è **descrittore di file** (*file descriptor*) in UNIX, e **handle del file** in Windows. Finché un file non viene chiuso, tutte le operazioni si compiono sulla tabella dei file aperti usando questo elemento.

Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella generale. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrive l'informazione aggiornata sul file nella struttura delle directory nei dischi e si cancella il relativo elemento nella tabella generale dei file aperti.



**Figura 11.3** Strutture del file system che si mantengono nella memoria; (a) apertura di file; (b) lettura di file.

Alcuni sistemi complicano ulteriormente lo schema descritto, usando il file system come interfaccia per altri aspetti del sistema, come la comunicazione in rete. Ad esempio, nell'UFS, la tabella generale dei file aperti contiene gli *inode* e altre informazioni su file e directory, ma contiene anche informazioni simili per le connessioni di rete e i dispositivi. In questo modo si può usare un unico meccanismo per molteplici fini.

Le questioni concernenti l'uso delle cache per queste strutture non vanno però trascurate; usando questo schema, tutta l'informazione su un file aperto, eccetto i suoi effettivi blocchi di dati, è tenuta in memoria. Il sistema UNIX BSD è noto per il suo uso di cache ovunque sia possibile risparmiare su operazioni di I/O nei dischi. La sua frequenza media di successi nella cache, pari all'85 per cento, dimostra l'utilità di queste tecniche.

La Figura 11.3 riassume le strutture che si usano nella realizzazione di un file system.

## 11.2.2 Partizioni e montaggio

Un disco si può configurare in vari modi, secondo il sistema operativo che lo gestisce. Si può suddividere in più partizioni, oppure un volume può comprendere più partizioni su molteplici dischi. Qui trattiamo il primo caso, mentre il secondo, che si può considerare un caso particolare di organizzazione RAID, è trattato nel Paragrafo 12.7.

Ciascuna partizione è priva di struttura logica (*raw partition*) se non contiene alcun file system. Se nessun file system è appropriato, si usa un disco privo di struttura logica (*raw disk*). Il sistema operativo UNIX impiega una partizione priva di struttura per l'area d'avviamento dei processi; per questo scopo usa un formato specifico. Allo stesso modo alcu-

ni sistemi di gestione di basi di dati usano dischi privi di un'ordinaria struttura logica e formattano i dati secondo le proprie necessità. Un disco privo di struttura logica può anche contenere informazioni necessarie per sistemi RAID di gestione dei dischi, ad esempio le mappe di bit che indicano quali blocchi sono duplicati in altri dischi, e quali sono stati modificati e si devono aggiornare negli altri dischi. Analogamente, può contenere una piccola base di dati di informazioni sulla configurazione RAID, ad esempio, quali dischi appartengono a ciascun insieme RAID. Il Paragrafo 12.5.1 affronta altri aspetti concernenti l'uso dei dischi privi di struttura logica.

Le informazioni relative all'avviamento del sistema si possono registrare in un'apposita partizione, che anche in questo caso ha un proprio formato, poiché nella fase d'avviamento il sistema non ha ancora caricato i driver di dispositivo del file system e quindi non può interpretarne il formato. Questa partizione consiste piuttosto in una serie sequenziale di blocchi, che si carica in memoria come un'immagine. L'esecuzione dell'immagine comincia a una locazione prefissata, ad esempio il primo byte. L'immagine d'avviamento può contenere più informazioni di quelle che servono per un singolo sistema operativo. I PC, ad esempio, e altri sistemi si possono configurare per l'installazione di più sistemi operativi (*dual-booted*). In questo caso l'area d'avviamento può contenere un modulo, detto caricatore d'avviamento (*boot loader*), capace di interpretare diversi file system e diversi sistemi operativi. Una volta caricato, può avviare uno dei sistemi operativi disponibili nei dischi. Ciascun disco può avere più partizioni, ognuna contenente un diverso tipo di file system e un sistema operativo differente.

Nella fase di caricamento del sistema operativo, si esegue il montaggio della **partizione radice** (*root partition*), che contiene il kernel del sistema operativo e in alcuni casi altri file di sistema. Secondo il sistema operativo, il montaggio degli altri volumi avviene automaticamente in questa fase oppure si può compiere successivamente in modo esplicito. Durante l'operazione di montaggio, il sistema verifica che il dispositivo contenga un file system valido chiedendo al dispositivo di leggere la directory di dispositivo e verificando che la directory abbia il formato corretto. Se così non fosse, è necessaria una verifica della coerenza della partizione e una eventuale correzione, con o senza l'intervento dell'utente. Infine, il sistema annota nella struttura della **tabella di montaggio** in memoria che un file system è stato montato insieme al tipo di file system. I dettagli di questa funzione dipendono dal sistema operativo. I sistemi basati sui sistemi operativi Microsoft Windows eseguono il montaggio di ogni volume in uno spazio di nomi separato, identificato da una lettera seguita dai due punti (:). Ad esempio, per memorizzare che un file system è stato montato in F:, il sistema operativo introduce un puntatore al file system in un campo della struttura del dispositivo corrispondente a F:. Quando un processo specifica la lettera dell'unità, il sistema operativo trova il puntatore al file system appropriato e attraversa la struttura delle directory in quel dispositivo per trovare lo specifico file o directory. Le recenti versioni di Windows permettono il montaggio di un file system in qualsiasi punto all'interno della struttura della directory esistente.

In UNIX, l'operazione di montaggio di un file system si può compiere in qualsiasi directory. Questa funzione si realizza impostando un flag nella copia dell'*inode* tenuta in memoria di quella directory, che segnala che la directory è un punto di montaggio. Un campo dell'*inode* punta a un elemento nella tabella di montaggio, che indica quale dispositivo è montato in quella posizione. L'elemento della tabella di montaggio contiene un puntatore al superblocco del file system in quel dispositivo. Questo schema permette al sistema operativo di attraversare facilmente la propria struttura della directory, passando da un file system all'altro secondo le necessità.

### 11.2.3 File system virtuali

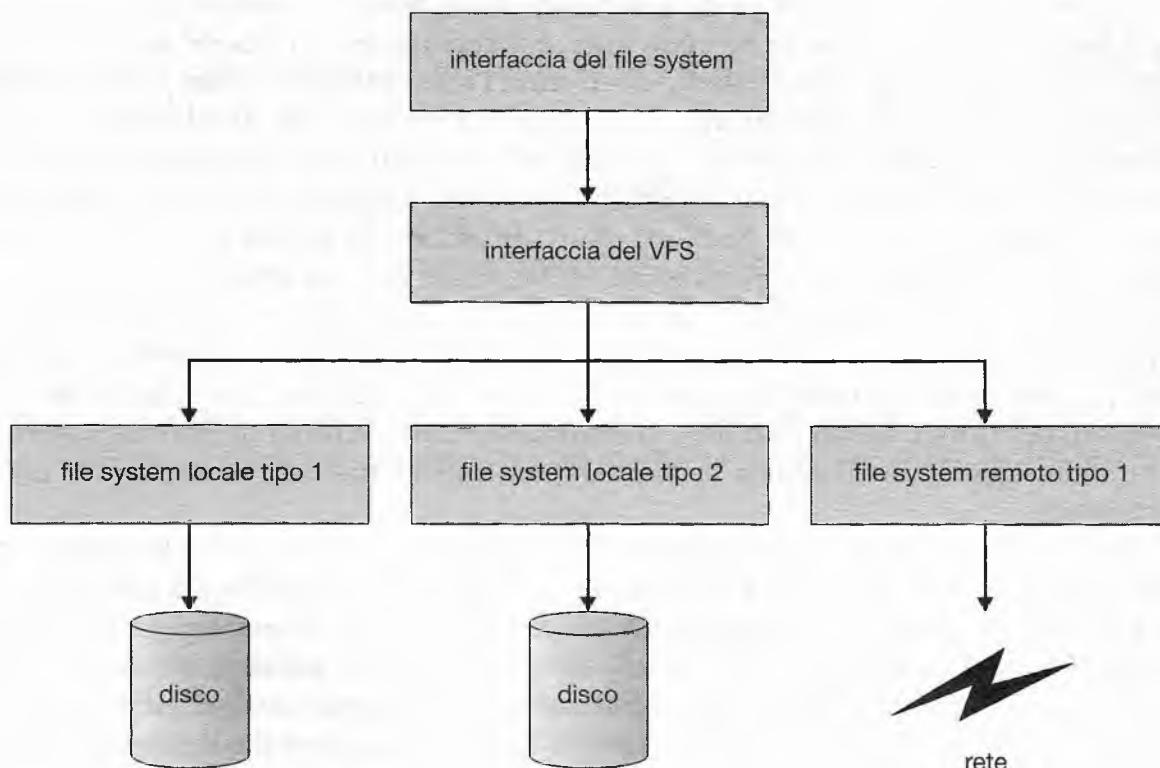
Nel paragrafo precedente si sottolinea il fatto che i sistemi operativi moderni devono gestire contemporaneamente tipi di file system diversi. Per capire come si può realizzare questa funzione occorre tuttavia considerare il modo in cui un sistema operativo può consentire l'integrazione di diversi tipi di file system in un'unica struttura della directory, così da permettere agli utenti di spostarsi senza problemi da un tipo di file system all'altro, mentre percorrono lo spazio del file system complessivo.

Un metodo ovvio ma non ottimale per realizzare più tipi di file system è scrivere procedure di gestione di file e directory separate per ciascun tipo di file system. Al contrario, la maggior parte dei sistemi operativi, compreso UNIX, impiega tecniche orientate agli oggetti per semplificare e organizzare in maniera modulare la soluzione. L'uso di queste tecniche rende possibile la realizzazione, nella stessa struttura, di tipi di file system molto diversi tra loro, compresi i file system di rete, come l'NFS. Gli utenti possono accedere a file contenuti in diversi file system nei dischi locali, o anche in file system disponibili tramite la rete.

Per isolare le funzioni di base delle chiamate di sistema dai dettagli di realizzazione si adoperano apposite strutture dati. In questo modo la realizzazione del file system si articola in tre strati principali, riportati in modo schematico nella Figura 11.4. Il primo strato è l'interfaccia del file system, basata sulle chiamate di sistema `open()`, `read()`, `write()` e `close()` e sui descrittori di file.

Il secondo strato si chiama strato del file system virtuale (*virtual file system*, VFS) e svolge due funzioni importanti.

1. Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme. Nello stesso calcolatore possono coesistere più interfacce VFS, che permettono un accesso trasparente a diversi tipi di file system montati localmente.



**Figura 11.4** Schema di un file system virtuale.

2. Permette la rappresentazione univoca di un file su tutta la rete. Il VFS è basato su una struttura di rappresentazione dei file detta *vnode* che contiene un indicatore numerico unico per tutta la rete per ciascun file. (Gli *inode* di UNIX sono unici solo all'interno di un singolo file system.) Tale unicità per tutta la rete è richiesta per la gestione del file system di rete. Il kernel contiene una struttura *vnode* per ciascun nodo attivo, sia che si tratti di un file sia che si tratti di una directory.

Quindi, il VFS distingue i file locali da quelli remoti, e distingue i file locali secondo i relativi tipi di file system.

Il VFS attiva le operazioni specifiche del file system per gestire le richieste locali secondo i tipi di file system, e invoca le procedure del protocollo NFS per le richieste remote. Gli handle del file si costruiscono secondo i *vnode* relativi e s'inviano a queste procedure come argomenti. Lo strato che realizza il protocollo NFS è il più basso dell'architettura.

Esaminiamo succintamente l'architettura VFS di Linux. I quattro tipi più importanti di oggetti definiti in questo sistema sono:

- ◆ l'oggetto *inode*, che rappresenta il singolo file;
- ◆ l'oggetto *file*, che rappresenta un file aperto;
- ◆ l'oggetto *superblock*, che rappresenta un intero file system;
- ◆ l'oggetto *dentry*, che rappresenta il singolo elemento della directory.

Per ognuno di questi tipi, VFS specifica un insieme di operazioni da implementare. Ciascun oggetto di uno di questi tipi contiene un puntatore a una tabella di funzioni; questa, alla sua volta, contiene gli indirizzi delle effettive funzioni che implementano le operazioni richieste dalla specifica dell'oggetto. Per esempio, una versione semplificata della API di alcune delle operazioni dell'oggetto file è:

- ◆ `int open(...)` – apre il file.
- ◆ `ssize_t read(...)` – legge dal file.
- ◆ `ssize_t write(...)` – scrive sul file.
- ◆ `int mmap(...)` – mappa il file in memoria.

Ogni implementazione dell'oggetto file per uno specifico tipo di file deve implementare tutte le funzioni specificate nella definizione dell'oggetto file. (La definizione completa dell'oggetto file si trova nella struttura `struct file_operations`, nel file `/usr/include/linux/fs.h`.)

Questo schema permette a VFS di eseguire operazioni su uno degli oggetti in questione invocando l'appropriata funzione della tabella delle funzioni dell'oggetto, senza dover conoscere i dettagli dello specifico oggetto. Per esempio, VFS non sa, né vuol sapere, se un certo inode rappresenti un file su disco, un file di directory o un file remoto. L'implementazione appropriata dell'operazione `read()` per l'oggetto in questione è comunque reperibile sempre nel medesimo punto all'interno della tabella delle funzioni: invocando tale implementazione, VFS può disinteressarsi dei dettagli legati all'effettiva lettura dei dati.

## 11.3 Realizzazione delle directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del file system. Per tale ragione è necessario comprendere i vari aspetti di questi algoritmi, trattati di seguito.

### 11.3.1 Lista lineare

Il più semplice metodo di realizzazione di una directory è basato sull'uso di una lista lineare contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo. Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era assegnato. Esistono vari metodi per riutilizzare un elemento della directory: si può contrassegnare l'elemento come non usato (attribuendogli un nome speciale, come un nome vuoto, oppure un bit d'uso in ogni elemento), oppure può essere aggiunto a una lista di elementi di directory liberi; una terza possibilità prevede la copiatura dell'ultimo elemento della directory in una locazione liberata e la diminuzione della lunghezza della directory. Per ridurre il tempo di cancellazione di un file si può usare anche una lista concatenata.

Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file. Le informazioni sulla directory vengono usate frequentemente, e gli utenti avvertirebbero una gestione lenta dell'accesso a tali informazioni. In effetti, molti sistemi operativi impiegano una cache per memorizzare le informazioni sulla directory usata più recentemente. La presenza nella cache delle informazioni richieste ne evita la continua rilettura dai dischi. Una lista ordinata permette una ricerca binaria e riduce il tempo medio di ricerca, tuttavia il requisito dell'ordinamento può complicare la creazione e la cancellazione di file, poiché, per tenere ordinata la lista, può essere necessario spostare quantità notevoli di informazioni sulla directory. In questo caso, può essere d'aiuto una struttura dati più raffinata, come un B-albero. Un vantaggio della lista ordinata è che consente di produrre l'elenco ordinato del contenuto della directory senza una fase d'ordinamento separata.

### 11.3.2 Tabella hash

Un'altra struttura dati che si usa per realizzare le directory è la **tabella hash**. In questo metodo una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato usando come operando il nome del file e riporta un puntatore al nome del file nella lista lineare. Questa struttura dati può diminuire notevolmente il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per evitare **collisioni**, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

Le maggiori difficoltà legate a una tabella hash sono la sua dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione. Si supponga, ad esempio, di realizzare una tabella hash di 64 elementi; la funzione hash converte i nomi di file in interi da 0 a 63, ad esempio, usando il resto di una divisione per 64. Per creare in un secondo tempo un sessantacinquesimo file occorre allungare la tabella hash della directory, ad esempio fino a 128 elementi. Occorre quindi una nuova funzione hash per associare i nomi di file all'intervallo compreso tra 0 e 127, e gli elementi esistenti nella directory si devono riorganizzare in modo da riflettere i loro nuovi valori della funzione hash.

Alternativamente, ciascun elemento della tabella hash, anziché un singolo valore, può essere una lista concatenata; ciò consente di risolvere le collisioni aggiungendovi il nuovo elemento. Le ricerche vengono alquanto rallentate, poiché la ricerca per nome può richiedere l'attraversamento della lista concatenata degli elementi in collisione della tabella hash; probabilmente tale metodo è comunque più veloce di una ricerca lineare nell'intera directory.

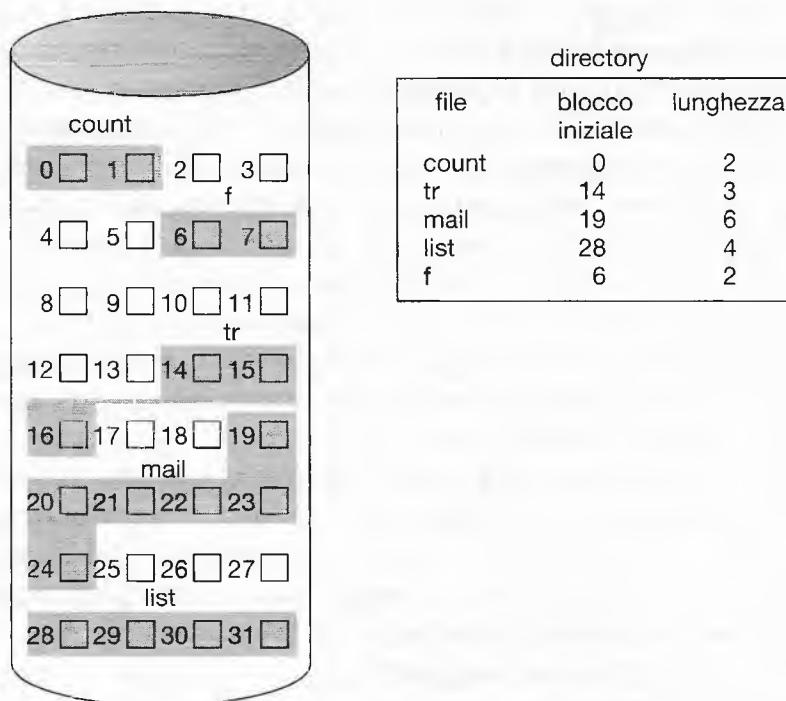
## 11.4 Metodi di allocazione

La natura ad accesso diretto dei dischi permette una certa flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere infatti contigua, concatenata o indicizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi. Alcuni sistemi, come l'R DOS di Data General per la linea di calcolatori Nova, dispongono di tutti e tre i metodi. Generalmente, però, un sistema usa un unico metodo per tutti i file all'interno di un tipo di file system.

### 11.4.1 Allocazione contigua

Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui del disco. Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso. Con questo ordinamento l'accesso al blocco  $b + 1$  dopo il blocco  $b$  non richiede normalmente alcuno spostamento della testina. Se la testina deve essere spostata dall'ultimo settore di un cilindro al primo settore del cilindro successivo lo spostamento avviene su una sola traccia. Quindi, il numero dei posizionamenti (*seek*) richiesti per accedere a file il cui spazio è allocato in modo contiguo è trascurabile, così com'è trascurabile il tempo di ricerca (*seek time*), quando quest'ultimo è necessario. Il sistema operativo IBM VM/CMS usa l'allocazione contigua poiché consente tali buone prestazioni.

L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco (inteso come numero di blocco) e dalla lunghezza (espressa in blocchi). Se il file è lungo  $n$  blocchi e comincia dalla locazione  $b$ , allora occupa i blocchi  $b, b + 1, b + 2, \dots, b + n - 1$ . L'elemento di directory per ciascun file indica l'indirizzo del blocco d'inizio e la lunghezza dell'area assegnata per questo file (Figura 11.5).



**Figura 11.5** Allocazione contigua dello spazio dei dischi.

Accedere a un file il cui spazio è assegnato in modo contiguo è facile. Quando si usa un accesso sequenziale, il file system memorizza l'indirizzo dell'ultimo blocco cui è stato fatto riferimento e, se è necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco  $i$  di un file che comincia al blocco  $b$  si può accedere immediatamente al blocco  $b + i$ . Quindi, sia l'accesso sequenziale sia quello diretto si possono gestire con l'allocazione contigua.

L'allocazione contigua presenta però alcuni problemi; una difficoltà riguarda l'individuazione dello spazio per un nuovo file. La realizzazione del sistema di gestione dello spazio libero, illustrata nel Paragrafo 11.5, determina il modo in cui tale compito viene eseguito. Si può usare ogni sistema di gestione, anche se alcuni sono più lenti di altri.

Il problema dell'allocazione contigua dello spazio dei dischi si può considerare un'applicazione particolare del problema generale dell'**allocazione dinamica della memoria**, trattato nel Paragrafo 8.3; il problema generale è, infatti, quello di soddisfare una richiesta di dimensione  $n$  data una lista di buchi liberi. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono quelli del primo buco abbastanza grande (*first-fit*) e del più piccolo tra i buchi abbastanza grandi (*best-fit*). Simulazioni hanno dimostrato che questi due criteri sono più efficienti di quello di scelta del buco più grande (*worst-fit*) sia in termini di tempo sia d'uso della memoria. Nessuno dei due è palesemente migliore rispetto all'uso della memoria, ma la scelta del primo buco abbastanza grande è generalmente più rapida.

Questi algoritmi soffrono della **frammentazione esterna**: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. La frammentazione esterna si ha ognqualvolta lo spazio libero è suddiviso in pezzi, e diviene un problema quando il più grande di tali pezzi contigui non è sufficiente a soddisfare una richiesta; la memoria viene frammentata in tanti buchi, nessuno dei quali è abbastanza grande da contenere i dati. Secondo la capacità dei dischi e la dimensione media dei file, la frammentazione esterna può essere un problema più o meno grave.

Una strategia per prevenire la perdita di una quantità significativa di spazio sul disco a causa della frammentazione esterna consiste nel copiare un intero file system su un altro disco o nastro. Quindi si liberava completamente il primo disco creando un ampio spazio libero contiguo. La procedura provvedeva poi a copiare nuovamente i file nel disco, assegnando tale spazio contiguo. Questo schema **compatta** efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Il costo di questa compattazione è rappresentato dal tempo necessario, ed è particolarmente pesante per i dischi di grande capacità che impiegano l'allocazione contigua; in essi, compattare lo spazio può richiedere ore e può essere necessario eseguire tale operazione settimanalmente. Alcuni sistemi richiedono l'esecuzione **non in linea** (*off-line*) di questa funzionalità, ossia con il file system non montato. Durante questo "periodo morto" (*down time*) il normale funzionamento del sistema non è possibile, quindi tale compattazione va evitata a tutti i costi per i calcolatori in attività. Molti sistemi moderni sono invece in grado di eseguire la deframmentazione **in linea** (*on-line*), ossia durante il loro normale funzionamento, a prezzo, però, di una sostanziale diminuzione delle prestazioni.

Un altro problema che riguarda l'allocazione contigua è la determinazione della quantità di spazio necessaria per un file. Quando si crea un file, occorre trovare e allocare lo spazio di cui necessita. Esiste il problema di conoscere la dimensione del file da creare; in alcuni casi questa dimensione si può stabilire in modo abbastanza semplice, ad esempio quando si copia un file esistente; in generale, tuttavia, non è facile stimare la dimensione di un file che deve contenere dati emessi da un programma.

Se un file riceve poco spazio, può essere impossibile estenderlo: soprattutto nel caso in cui si adoperi il criterio di allocazione del più piccolo tra i buchi abbastanza grandi, lo spazio oltre le due estremità del file può essere già in uso, quindi non è possibile ampliare il fi-

le in modo contiguo. Esistono allora due possibilità. La prima è che il programma utente si possa terminare con un idoneo messaggio d'errore. L'utente deve allora allocare più spazio ed eseguire di nuovo il programma. Queste esecuzioni ripetute possono essere onerose; per prevenire tale circostanza, normalmente l'utente sovrastima la quantità di spazio necessaria, sprecandone parecchio. L'altra possibilità consiste nel trovare un buco più grande, copiare il contenuto del file nel nuovo spazio e rilasciare lo spazio precedente. Queste operazioni si possono ripetere finché esiste spazio, anche se ciò può far perdere tempo. In questo caso tuttavia non è necessario informare esplicitamente l'utente su che cosa stia succedendo; anche se più lentamente, il sistema prosegue le attività nonostante il problema.

Anche se si conosce in anticipo la quantità di spazio necessaria per un file, l'allocazione preventiva può in ogni modo essere inefficiente. A un file che cresce lentamente in un periodo di tempo lungo (mesi o anni) si deve allocare spazio sufficiente per la sua dimensione finale, anche se molto di quello spazio può rimanere inutilizzato per parecchio tempo. Il file ha perciò un'estesa frammentazione interna.

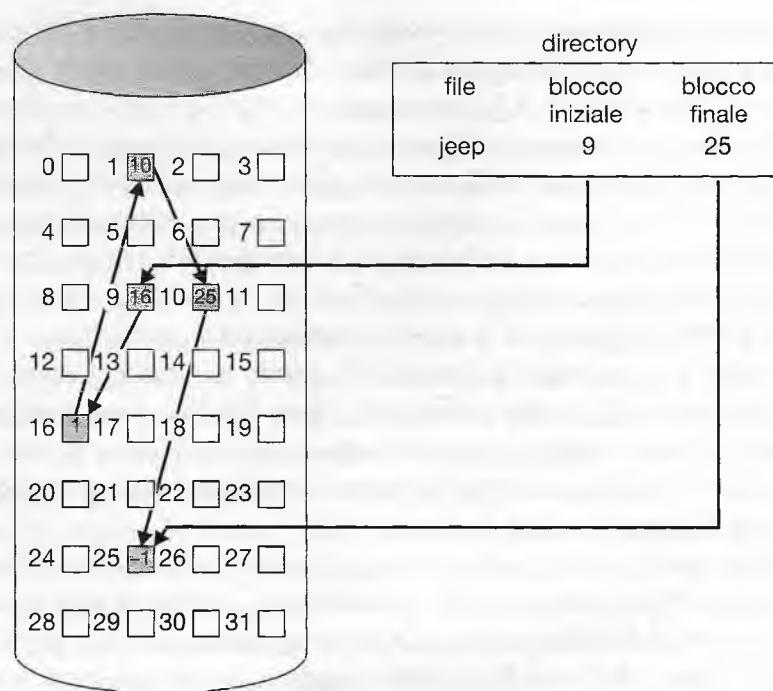
Per ridurre al minimo questi inconvenienti, alcuni sistemi operativi fanno uso di uno schema di allocazione contigua modificato: inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio, un'estensione. La locazione dei blocchi del file si registra come una locazione e un numero dei blocchi, insieme con l'indirizzo del primo blocco dell'estensione seguente. In alcuni sistemi il proprietario del file può impostare la dimensione dell'estensione, e tale possibilità, se il proprietario è impreciso, può causare inefficienze. La frammentazione interna può ancora essere un problema se le estensioni sono troppo grandi; si possono presentare problemi dovuti alla frammentazione esterna quando si assegnano e si rilasciano estensioni di dimensione variabile. Il file system commerciale Veritas impiega le estensioni per ottimizzare le prestazioni; è un sostituto ad alte prestazioni dell'ordinario UFS di UNIX.

## 11.4.2 Allocazione concatenata

L'**allocazione concatenata** risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Ad esempio, un file di cinque blocchi può cominciare dal blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25 (Figura 11.6). Ogni blocco contiene un puntatore al blocco successivo. Questi puntatori non sono disponibili all'utente, quindi se ogni blocco è formato di 512 byte e un indirizzo del disco (il puntatore) richiede 4 byte, l'utente vede blocchi di 508 byte.

Per creare un nuovo file si crea semplicemente un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore s'inizializza a n11 (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo della dimensione s'imposta a 0. Un'operazione di scrittura nel file determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura in tale blocco, e la concatenazione di tale blocco alla fine del file. Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori da un blocco all'altro. Con l'allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione. Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco.

L'allocazione concatenata presenta comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenzia-



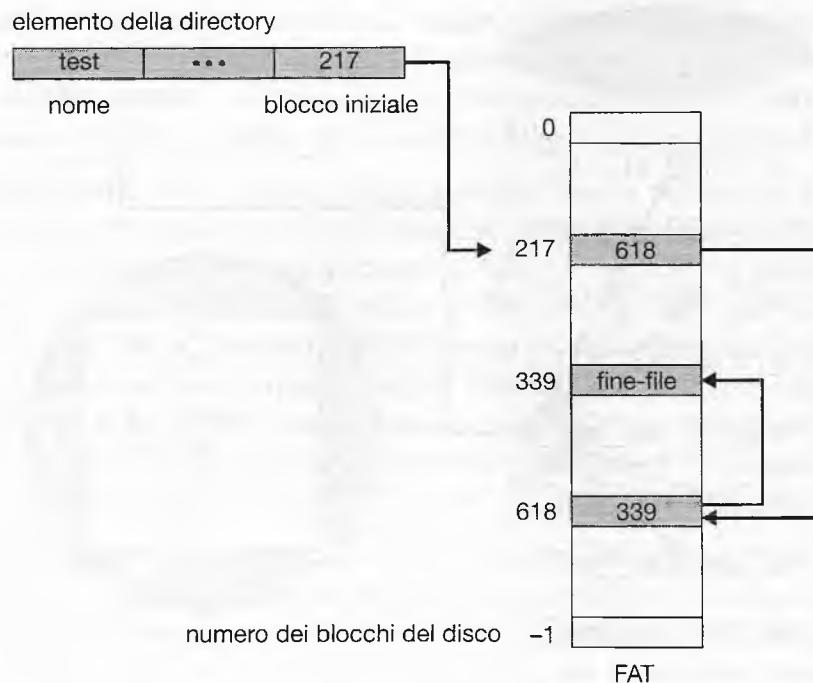
**Figura 11.6** Allocazione concatenata dello spazio dei dischi.

le. Per trovare l' $i$ -esimo blocco di un file occorre partire dall'inizio del file e seguire i puntatori finché non si raggiunge l' $i$ -esimo blocco. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina. Di conseguenza, per file il cui spazio è assegnato in modo concatenato, la funzione d'accesso diretto è inefficiente.

Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Se un puntatore richiede 4 byte di un blocco di 512 byte, allora lo 0,78 per cento del disco è usato per i puntatori anziché per le informazioni: ogni file richiede un po' più spazio di quanto ne richiederebbe altrimenti.

La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in **cluster** (gruppi di blocchi), e nell'allocare i cluster anziché i blocchi. Ad esempio, il file system può definire cluster di 4 blocchi e operare nel disco soltanto per unità di cluster. Così i puntatori usano una quantità di spazio di disco che si riduce in modo proporzionale al numero di cluster. Questo metodo permette che la corrispondenza tra blocchi logici e blocchi fisici rimanga semplice, ma migliora la produttività del disco: si hanno meno posizionamenti della testina del disco e diminuisce lo spazio necessario per l'allocazione dei blocchi e la gestione della lista dei blocchi liberi. Il costo di questo metodo è dato da un incremento della frammentazione interna, poiché se un cluster è parzialmente pieno si spreca più spazio di quanto se ne sprecherebbe con un solo blocco parzialmente pieno. I cluster si possono usare per ottimizzare l'accesso ai dischi in molti altri algoritmi, quindi s'impiegano nella maggior parte dei sistemi operativi.

Un altro problema riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, s'immagini che cosa accadrebbe se un puntatore andasse perduto o danneggiato. Un errore di programmazione del sistema operativo oppure un errore di un'unità a disco potrebbero causare il prelevamento del puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. Una soluzione parziale a tale problema consiste nell'usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco; questi schemi però sono ancora più onerosi per ogni file.



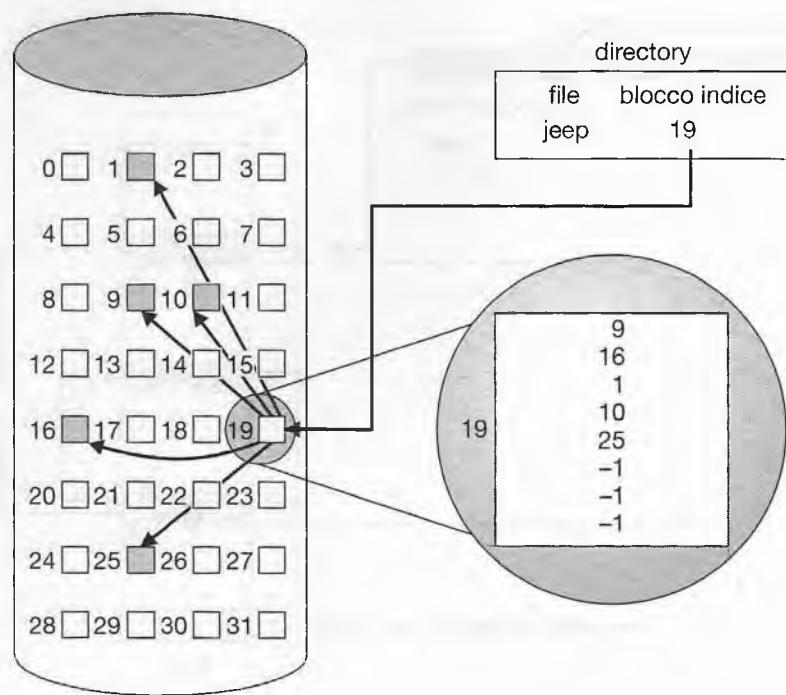
**Figura 11.7** Tabella di allocazione dei file.

Una variante importante del metodo di allocazione concatenata consiste nell'uso della **tavella di allocazione dei file** (*file allocation table*, FAT). Tale metodo di allocazione, semplice ma efficiente, dello spazio dei dischi è usato nei sistemi operativi MS-DOS e OS/2. Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file. Un esempio esplicativo di tale metodo è dato dalla struttura della FAT della Figura 11.7, dove il file in questione è formato dai blocchi 217, 618 e 339.

Lo schema di allocazione basato sulla FAT, se non si usa una cache, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio del volume per leggere la FAT e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco. Un vantaggio è dato dall'ottimizzazione del tempo d'accesso diretto, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella FAT.

### 11.4.3 Allocazione indicizzata

L'allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni dei file, entrambi presenti nell'allocazione contigua. Tuttavia, in mancanza di una FAT, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il di-



**Figura 11.8** Allocazione indicizzata dello spazio dei dischi.

sco e si devono recuperare in ordine. L'allocazione indicizzata risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il **blocco indice**.

Ogni file ha il proprio blocco indice: si tratta di un array d'indirizzi di blocchi del disco. L' $i$ -esimo elemento del blocco indice punta all' $i$ -esimo blocco del file. La directory contiene l'indirizzo del blocco indice, com'è illustrato nella Figura 11.8. Per individuare e leggere l' $i$ -esimo blocco occorre usare il puntatore che si trova nell' $i$ -esimo elemento del blocco indice, per poi localizzare e leggere il blocco desiderato. Questo schema è simile a quello della paginazione descritto nel Paragrafo 8.4.

Una volta creato il file, tutti i puntatori del blocco indice sono impostati a `nil`. Quando si scrive l' $i$ -esimo blocco per la prima volta, il gestore dei blocchi liberi fornisce un blocco; l'indirizzo di questo blocco viene inserito nell' $i$ -esimo elemento del blocco indice. Poiché ogni blocco libero del disco può soddisfare una richiesta di maggiore spazio, l'allocazione indicizzata consente l'accesso diretto senza soffrire di frammentazione esterna.

Lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spazio aggiuntivo necessario per l'allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l'allocazione concatenata si perde il solo spazio di un puntatore per blocco, complessivamente uno o due puntatori; con l'allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori sono diversi da `nil`.

Questo punto solleva la questione della dimensione del blocco indice. Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo è possibile; ma se il blocco indice è troppo piccolo non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione.

- ◆ **Schema concatenato.** Un blocco indice è formato normalmente di un solo blocco di disco; perciò, ciascun blocco indice può essere letto e scritto esattamente con un'operazione. Per permettere la presenza di lunghi file è possibile collegare tra loro parecchi

blocchi indice. Ad esempio, un blocco indice può contenere una piccola intestazione in cui sono riportati il nome del file e l'insieme dei primi 100 indirizzi del blocco di disco. L'indirizzo successivo, vale a dire l'ultima parola del blocco indice, è `nil` (per un file piccolo) oppure è un puntatore a un altro blocco indice (per un file lungo).

- ◆ **Indice a più livelli.** Una variante della rappresentazione concatenata consiste nell'impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file. Per accedere a un blocco, il sistema operativo usa l'indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco di dati richiesto. Questo metodo può continuare fino a un terzo o quarto livello, secondo la massima dimensione desiderata del file. Con blocchi di 4096 byte si possono memorizzare 1024 puntatori di 4 byte in un blocco indice. Due livelli di indici consentono 1.048.576 blocchi di dati, che permettono di avere file sino a 4 GB.
- ◆ **Schema combinato.** Un'altra possibilità, è la soluzione adottata nell'UFS, consistente nel tenere i primi 15 puntatori del blocco indice nell'*inode* del file. I primi 12 di questi 15 puntatori puntano a **blocchi diretti**, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Quindi, i dati per piccoli file (non più di 12 blocchi) non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4 KB, è possibile accedere direttamente fino a 48 KB di dati. Gli altri tre puntatori puntano a **blocchi indiretti**. Il primo puntatore di blocco indiretto è l'indirizzo di un **blocco indiretto singolo**; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Quindi c'è un puntatore di **blocco indiretto doppio** contenente l'indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. L'ultimo puntatore contiene l'indirizzo di un **blocco indiretto triplo**. Con questo metodo, il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti sistemi operativi. Un puntatore a file di 32 bit consente di arrivare a soli  $2^{32}$  byte, 4 GB. Molte versioni del sistema operativo UNIX, tra le quali Solaris e l'IBM AIX ora gestiscono puntatori a file sino a 64 bit. Puntatori di questa dimensione permettono di avere file e file system di dimensioni dell'ordine dei terabyte. Un *inode* UNIX è mostrato nella Figura 11.9.

Gli schemi d'allocazione indicizzata soffrono di alcuni dei problemi di prestazioni dell'allocazione concatenata. In particolare, i blocchi indice si possono caricare in memoria, ma i blocchi dei dati possono essere sparsi per un'intero volume.

#### 11.4.4 Prestazioni

I metodi d'allocazione presentati hanno diversi livelli di efficienza di memorizzazione e differenti tempi d'accesso ai blocchi di dati; entrambi i fattori sono importanti nella scelta del metodo o dei metodi d'allocazione più adatti da impiegare in un sistema operativo.

Prima di scegliere un metodo di allocazione, è necessario determinare il modo in cui si usano i sistemi: un sistema con una prevalenza di accessi sequenziali farà uso di un metodo differente da quello di un sistema con una prevalenza di accessi diretti.

Per qualsiasi tipo d'accesso, l'allocazione contigua richiede un solo accesso per ottenerne un blocco. Poiché è facile tenere l'indirizzo iniziale del file in memoria, si può calcolare immediatamente l'indirizzo del disco dell'*i*-esimo blocco, oppure del blocco successivo, e leggerlo direttamente.

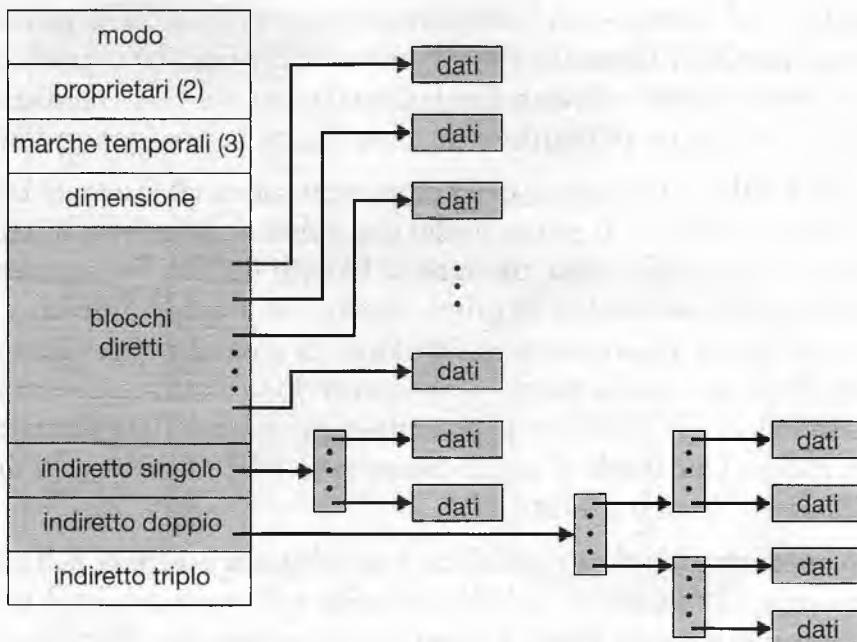


Figura 11.9 Inode di UNIX.

Con l'allocazione concatenata si può tenere in memoria anche l'indirizzo del blocco successivo e leggerlo direttamente. Questo metodo è valido per l'accesso sequenziale mentre, per quel che riguarda l'accesso diretto, un accesso all' $i$ -esimo blocco può richiedere  $i$  letture del disco. Questo spiega perché l'allocazione concatenata non si dovrebbe usare per un'applicazione che richiede accessi diretti.

Da tutto ciò segue che alcuni sistemi gestiscono i file ad accesso diretto usando l'allocazione contigua, e i file ad accesso sequenziale tramite l'allocazione concatenata. Per questi sistemi, il tipo d'accesso si deve dichiarare al momento della creazione del file. Un file creato per l'accesso sequenziale è un file concatenato e non si può usare per l'accesso diretto. Un file creato per l'accesso diretto è contiguo e consente entrambi i tipi d'accesso, purché se ne dichiari la lunghezza massima al momento della sua creazione. In questo caso, il sistema operativo deve avere strutture dati idonee e algoritmi capaci di gestire *entrambi* i metodi di allocazione. I file si possono convertire da un tipo all'altro creando un nuovo file del tipo desiderato, nel quale si copia il contenuto del vecchio file; quest'ultimo si può quindi cancellare e il nuovo file rinominare.

L'allocazione indicizzata è più complessa. Se il blocco indice è già in memoria, l'accesso può essere diretto. Tuttavia, per tenere il blocco indice in memoria occorre una quantità di spazio considerevole. Se questo spazio di memoria non è disponibile, occorre leggere prima il blocco indice e quindi il blocco di dati desiderato. Per un indice a due livelli possono essere necessarie due letture del blocco indice. Se un file è estremamente grande, per compiere l'accesso a un blocco che si trovi vicino alla fine del file, prima di leggere il blocco dei dati occorre leggere tutti i blocchi indice per seguire la catena dei puntatori. Quindi le prestazioni dell'allocazione indicizzata dipendono dalla struttura dell'indice, dalla dimensione del file e dalla posizione del blocco desiderato.

Alcuni sistemi combinano l'allocazione contigua con l'allocazione indicizzata, usando quella contigua per i file piccoli (fino a tre o quattro blocchi) e passando automaticamente a quella indicizzata per i file grandi. Poiché generalmente i file sono piccoli, e in questo caso l'allocazione contigua è efficiente, le prestazioni medie possono risultare abbastanza buone.

Nel 1991, ad esempio, la versione di UNIX di Sun Microsystems è stata modificata per migliorare le prestazioni dell'algoritmo di allocazione del file system. Alcune misure delle prestazioni hanno mostrato che la massima produttività del disco in una tipica stazione di lavoro (SPARCstation1, da 12 MIPS) richiedeva il 50 per cento d'impegno della CPU e produceva un'ampiezza di banda di soli 1,5 MB al secondo (56 KB era la massima dimensione del trasferimento per DMA di allora). Per migliorare le prestazioni, la Sun ha apportato alcune modifiche per allocare lo spazio in cluster di 56 KB ogniqualvolta fosse possibile. Questo metodo di allocazione riduceva la frammentazione esterna e i tempi di ricerca e di latenza. Inoltre le procedure di lettura dei dischi sono state ottimizzate per leggere in questi grandi cluster. La struttura dell'*inode* non è stata modificata. Queste modifiche insieme con l'uso della lettura anticipata e del rilascio indietro (discussi nel Paragrafo 11.6.2) hanno prodotto una diminuzione del 25 per cento dell'impegno della CPU e una produttività notevolmente migliorata.

Sono possibili e si usano effettivamente anche molte altre ottimizzazioni. Data la disparità tra velocità della CPU e velocità dei dischi, non è irragionevole aggiungere al sistema operativo migliaia di istruzioni solo per risparmiare alcuni movimenti della testina. Con il passare del tempo tale disparità aumenta a tal punto che, per ottimizzare i movimenti della testina, si possono ragionevolmente usare centinaia di migliaia di istruzioni.

## 11.5 Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere, se è possibile, nuovi file (i dischi ottici a una sola scrittura permettono una sola scrittura in qualsiasi settore e quindi il riutilizzo è fisicamente impossibile). Per tener traccia dello spazio libero in un disco, il sistema conserva una **lista dello spazio libero**; vi sono registrati tutti gli spazi *liberi*, cioè non allocati ad alcun file o directory. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista. Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati. A dispetto del suo nome, la lista dello spazio libero potrebbe non essere realizzata come una lista, come vedremo più avanti.

### 11.5.1 Vettore di bit

Spesso la lista dello spazio libero si realizza come una **mappa di bit**, o **vettore di bit**. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0.

Si consideri, ad esempio, un disco dove i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 sono liberi e gli altri sono allocati. La mappa di bit dello spazio libero è la seguente:

00111100111110001100000011100000...

I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o  $n$  blocchi liberi consecutivi nel disco; in effetti, molti calcolatori forniscono istruzioni di manipolazione dei bit utilizzabili con efficacia a tale scopo. Ad esempio, la famiglia di CPU Intel a partire dall'80386 e la famiglia di CPU Motorola a partire dal 68020 (rispettivamente, nei PC e nei vecchi Macintosh) hanno istruzioni che riportano lo scostamento del primo bit con valore 1 contenuto in una parola. Infatti il sistema operativo Apple Macintosh usava il metodo del vettore di bit per allocare lo spa-

zio dei dischi. Per trovare il primo blocco libero, il sistema operativo Macintosh controllava in modo sequenziale ogni parola nella mappa di bit per verificare che il valore non fosse 0, poiché una parola con valore 0 ha tutti i bit a 0 e rappresenta un insieme di blocchi assegnati. La prima parola non 0 viene scenduta alla ricerca del primo bit 1, che indica la locazione del primo blocco libero. Il numero del blocco è dato dalla seguente espressione:

$$(numero\ di\ bit\ per\ parola) \times (numero\ di\ parole\ di\ valore\ 0) + scostamento\ del\ primo\ bit\ 1.$$

Anche in questo caso le caratteristiche dell'architettura guidano le funzioni del sistema operativo. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, e viene di tanto in tanto scritto in memoria secondaria allo scopo di consentire eventuali operazioni di ripristino; è possibile tenere il vettore in memoria centrale se i dischi sono piccoli, come quelli usati nei microcalcolatori; tale soluzione non è applicabile ai dischi più grandi. Un disco di 1,3 GB con blocchi di 512 byte richiederebbe una mappa di bit di oltre 332 KB per tenere traccia dei suoi blocchi liberi. Il raggruppamento di quattro blocchi riduce questo numero a 83 KB per disco. Un disco di 1 TB con blocchi di 4 KB richiede 32 MB per memorizzare la propria mappa di bit. Dato che la dimensione del disco è in costante crescita, i problemi legati ai vettori di bit continueranno ad aggravarsi. Un file system di 1 PB richiederà una mappa di bit di 32 GB solo per gestire il suo spazio libero.

### 11.5.2 Lista concatenata

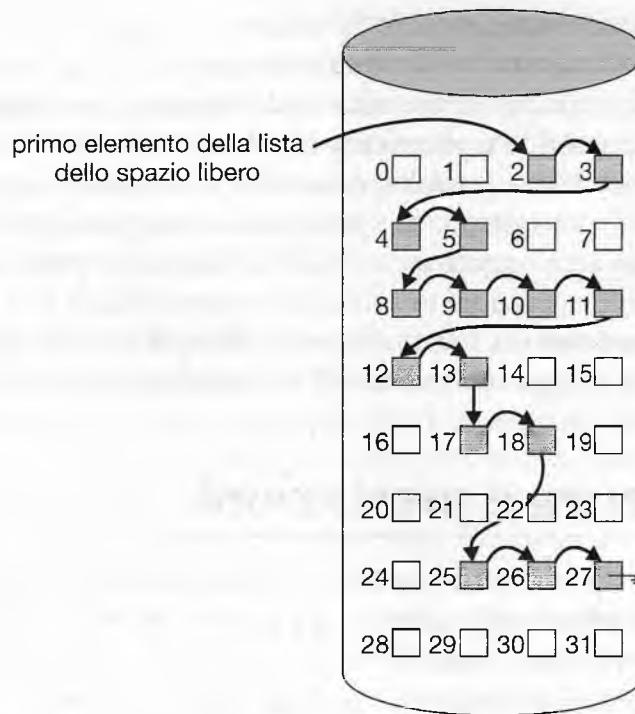
Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Questo primo blocco contiene un puntatore al successivo blocco libero, e così via. Facciamo riferimento al nostro esempio precedente (Paragrafo 11.5.1) nel quale i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 erano liberi, mentre i restanti blocchi erano allocati. In questa situazione dovremmo mantenere un puntatore al blocco 2, trattandosi del primo blocco libero. Il blocco 2 conterebbe un puntatore al blocco 3, che punterebbe al blocco 4, che punterebbe al blocco 5, il quale punterebbe a sua volta al blocco 8, e così via (Figura 11.10). Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di I/O. Fortunatamente la necessità di attraversare la lista dello spazio libero non è frequente. Di solito il sistema operativo ha semplicemente bisogno di un blocco libero perché possa assegnarlo a un file, quindi si usa il primo blocco della lista. Il metodo che fa uso della FAT include il conteggio dei blocchi liberi nella struttura dati per l'allocazione; non è necessario un metodo separato.

### 11.5.3 Raggruppamento

Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di  $n$  blocchi liberi nel primo di questi. I primi  $n - 1$  di questi blocchi sono effettivamente liberi; l'ultimo blocco contiene gli indirizzi di altri  $n$  blocchi liberi, e così via. L'importanza di questo metodo, diversamente dall'ordinaria lista concatenata, è data dalla possibilità di trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

### 11.5.4 Conteggio

Un altro orientamento sfrutta il fatto che, generalmente, più blocchi contigui si possono allocare o liberare contemporaneamente, soprattutto quando lo spazio viene allocato usando l'algoritmo di allocazione contigua o attraverso l'uso di cluster. Quindi, anziché tenere una lista di  $n$  indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero



**Figura 11.10** Lista concatenata degli spazi liberi su disco.

$n$  di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dello spazio libero è formato da un indirizzo del disco e un contatore. Anche se ogni elemento richiede più spazio di quanto ne richieda un semplice indirizzo del disco, se il contatore è generalmente maggiore di 1 la lista complessiva è più corta. Tenete presente che questo metodo, il tracciamento dello spazio libero, è simile al metodo generale per allocare i blocchi. Queste voci possono essere salvate in un B-albero o in una lista concatenata, in modo da permettere controlli, inserzioni e cancellazioni efficienti.

### 11.5.5 Mappe di spazio

Il file system ZFS di Sun è stato progettato per contenere un gran numero di file, directory e persino di file system (in ZFS è possibile creare gerarchie di file system). Se non progettate e implementate adeguatamente, le strutture di dati risultanti possono essere grandi e inefficienti. Su queste scale, i metadati I/O possono avere un impatto notevole sulle prestazioni. Notate ad esempio che, se la lista dello spazio libero è implementata come una mappa di bit, le mappe di bit devono essere modificate sia quando i blocchi vengono allocati sia quando vengono liberati. Liberare 1 GB di dati su un disco di 1 TB potrebbe comportare l'aggiornamento di migliaia di blocchi di mappe di bit, perché quei blocchi di dati potrebbero essere sparpagliati sull'intero disco.

Nel suo algoritmo di gestione dello spazio libero ZFS utilizza una combinazione di tecniche per controllare la dimensione delle strutture di dati e minimizzare l'I/O necessario a gestire quelle strutture. Per prima cosa, ZFS crea **metalastre** (*metaslab*) per dividere lo spazio sul dispositivo in parti che abbiano una dimensione gestibile. Un dato volume potrebbe contenere centinaia di metalastre. Ogni metalastra è associata a una mappa di spazio. ZFS utilizza l'algoritmo di conteggio per memorizzare informazioni riguardanti i blocchi liberi. Piuttosto che scrivere strutture di calcolo su disco, impiega le tecniche dei file system con registrazione delle modifiche per registrarle. La mappa di spazio è un registro di tutte le attivi-

tà del blocco (allocazione e liberazione), in ordine cronologico, in formato di conteggio. Quando ZFS decide di allocare o liberare spazio su una metalastra, carica la mappa di spazio associata nella memoria in una struttura ad albero bilanciato (per operazioni molto efficienti), indicizzata da scostamenti, e replica il log in tale struttura.

La mappa di spazio all'interno della memoria è un'accurata rappresentazione dello spazio allocato e libero nella metalastra. ZFS condensa la mappa il più possibile combinando blocchi liberi contigui in una singola voce. Infine la lista dello spazio libero viene aggiornata sul disco come parte delle operazioni orientate alle transazioni di ZFS. Durante la fase di raccolta e classificazione possono verificarsi ancora richieste di blocchi, che vengono soddisfatte dal registro. In sostanza, il registro, insieme all'albero bilanciato, costituiscono la lista libera.

## 11.6 Efficienza e prestazioni

Dopo avere descritto le opzioni di allocazione dei blocchi e di gestione delle directory, è possibile considerare i loro effetti sulle prestazioni e l'efficienza d'uso dei dischi. I dischi tendono di solito a essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti più rilevanti di un calcolatore. In questo paragrafo si considerano diverse tecniche utili per migliorare l'efficienza e le prestazioni della memoria secondaria.

### 11.6.1 Efficienza

L'uso efficiente di un disco dipende fortemente dagli algoritmi usati per l'allocazione del disco e la gestione delle directory. Ad esempio, gli *inode* di UNIX sono assegnati preventivamente in un volume. Anche un disco "vuoto" impiega una certa percentuale del suo spazio per gli *inode*. D'altra parte, l'allocazione preventiva degli *inode* e la loro distribuzione nel volume migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocazione e di gestione dei blocchi liberi adottati da UNIX, i quali cercano di mantenere i blocchi di dati di un file vicini al blocco che ne contiene l'*inode* allo scopo di ridurre il tempo di ricerca.

Come ulteriore esempio, si consideri lo schema che fa uso dei cluster presentato nel Paragrafo 11.4, che migliora le prestazioni di ricerca in un file e trasferimento di un file al costo di una maggiore frammentazione interna. Per ridurre questa frammentazione, il BSD UNIX varia la dimensione del cluster al crescere della dimensione del file. Cluster più grandi si usano dove possono essere riempiti, mentre cluster più piccoli si usano per file di piccole dimensioni e per l'ultimo cluster di un file.

Si devono tenere in considerazione anche il tipo di dati normalmente contenuti in un elemento di una directory (o di un *inode*). Di solito si memorizza la *data dell'ultima scrittura* per fornire informazioni all'utente e per determinare se per il file occorre la creazione o l'aggiornamento di una copia di riserva. Alcuni sistemi mantengono anche la *data dell'ultimo accesso* per consentire all'utente di risalire all'ultima volta che un file è stato letto. Per mantenere queste informazioni, ogniqualvolta si legge un file, si deve aggiornare un campo della directory. Questa modifica richiede la lettura nella memoria del blocco, la modifica della sezione e la riscrittura del blocco nel disco, poiché sui dischi si può operare solamente per blocchi (o cluster). Quindi, ogni volta che si apre un file per la lettura, si deve leggere e scrivere anche l'elemento della directory a esso associato. Ciò può essere inefficiente per file cui si accede frequentemente, quindi nella fase della progettazione del file system è necessario confrontare i benefici con i costi rispetto alle prestazioni. In generale, è necessario considerare l'influenza sull'efficienza e sulle prestazioni di *ogni* informazione che si vuole associare a un file.

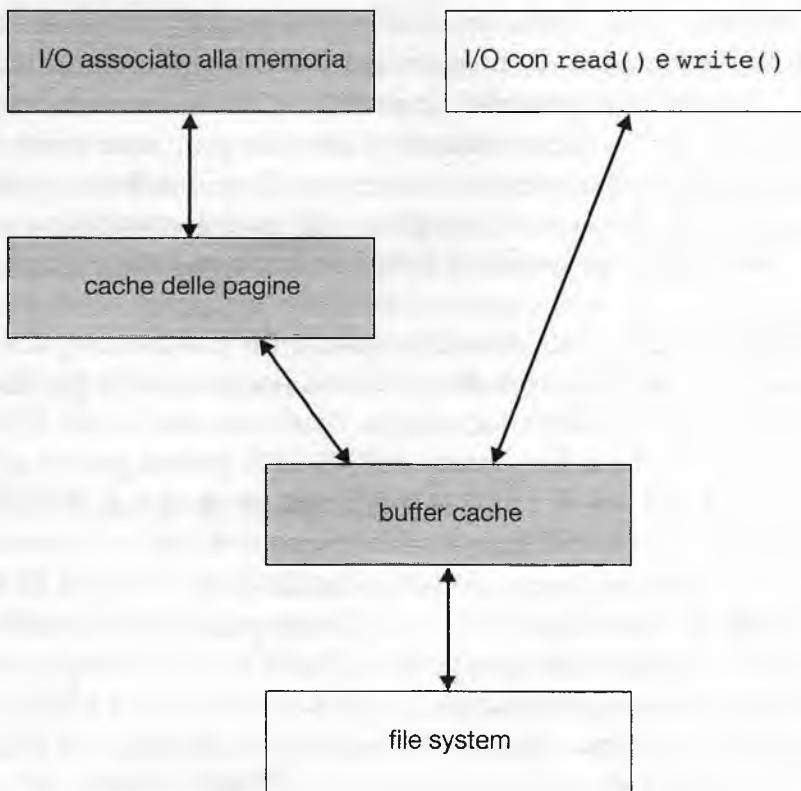
A titolo d'esempio, si consideri come l'efficienza sia influenzata dalle dimensioni dei puntatori usati per l'accesso ai dati. La maggior parte dei sistemi usa puntatori di 16 o 32 bit ovunque all'interno del sistema operativo. Questa dimensione limita la lunghezza di un file a  $2^{16}$  (64 KB) o  $2^{32}$  byte (4 GB). Alcuni sistemi impiegano puntatori di 64 bit per portare il limite a  $2^{64}$  byte, un numero effettivamente molto grande. Comunque, i puntatori di 64 bit richiedono più spazio per la loro memorizzazione e di conseguenza fanno sì che i metodi di allocazione e di gestione dello spazio libero (liste concatenate, indici, e così via) impieghino più spazio.

Una delle difficoltà nella scelta della dimensione dei puntatori, o di qualsiasi altra dimensione di allocazione fissa all'interno di un sistema operativo, è la pianificazione degli effetti provocati dal cambiamento della tecnologia. Basti considerare che il primo IBM PC XT aveva un disco di 10 MB e che il file system dell'MS-DOS poteva gestire solamente 32 MB. (Ciascun elemento della FAT era di 12 bit e puntava a un cluster di 8 KB.) Con la crescita della capacità dei dischi, i dischi più grandi si dovevano suddividere in partizioni di 32 MB, poiché il file system non poteva tener traccia di blocchi disposti oltre i 32 MB. Quando divennero comuni dischi di capacità superiore ai 100 MB, si dovettero modificare le strutture dati e gli algoritmi usati dall'MS-DOS per gestire i dischi in modo da consentire file system più grandi. (La dimensione di ciascun elemento della FAT fu portata a 16 bit e più tardi a 32 bit.) La decisione iniziale fu presa per motivi di efficienza; tuttavia, con l'avvento della Versione 4 dell'MS-DOS milioni di utenti si trovarono a disagio quando dovettero passare ai nuovi, più grandi file system. Il file system ZFS di Sun adotta puntatori di 128 bit, che in teoria non dovrebbero mai necessitare di un'estensione. (La minima massa di un dispositivo in grado di archiviare  $2^{128}$  byte tramite memorizzazione al livello atomico è di circa 272 trilioni di chilogrammi.)

Come altro esempio, si consideri l'evoluzione del sistema operativo Solaris di Sun Microsystems. Originariamente molte strutture dati avevano lunghezza fissa ed erano assegnate all'avviamento del sistema. Queste strutture comprendevano la tabella dei processi e quella dei file aperti. Una volta riempite queste tabelle, non si potevano più creare nuovi processi o aprire nuovi file, sicché il sistema non riusciva nel proprio compito di fornire un servizio agli utenti. L'unico modo di aumentare le dimensioni di queste tabelle era la ricompilazione del kernel e il riavvio del sistema. Dall'uscita di Solaris 2 quasi tutte le strutture dati del kernel si assegnano in modo dinamico, eliminando questi limiti artificiali alle prestazioni del sistema. Naturalmente, gli algoritmi che manipolano queste tabelle sono ora più complessi e il sistema operativo è un po' più lento dovendo allocare e rilasciare dinamicamente gli elementi delle tabelle, ma questo è il prezzo da pagare per la generalizzazione delle funzioni.

## 11.6.2 Prestazioni

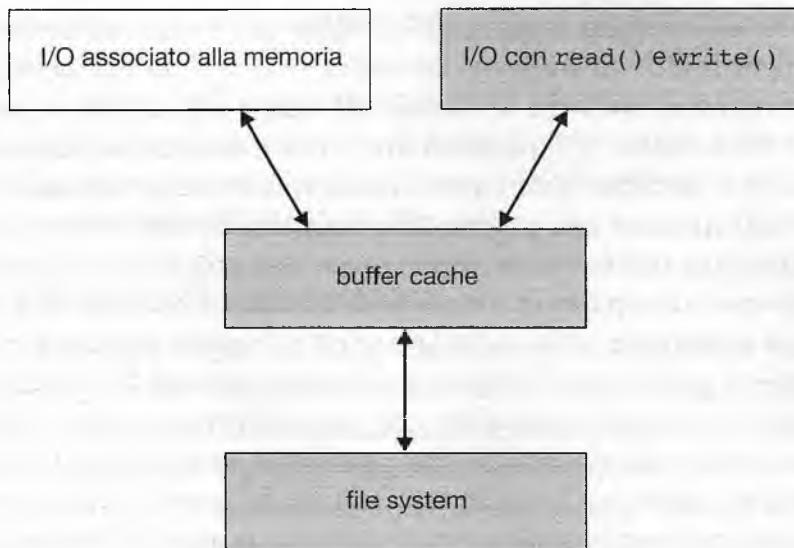
Dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi. Come si osserva nel Capitolo 13, alcuni controllori di unità a disco contengono una quantità di memoria locale sufficiente per la creazione di una **cache** interna al controllore sufficientemente grande da memorizzare un'intera traccia del disco alla volta. Eseguito il posizionamento della testina, si legge la traccia nella cache del controllore del disco a partire dal settore sotto cui si viene a trovare la testina (riducendo il tempo di latenza). Il controllore trasferisce quindi al sistema operativo tutte le richieste di settori. Quando i blocchi sono trasferiti dal controllore del disco alla memoria centrale, il sistema operativo ha la possibilità di inserirli in una propria cache nella memoria centrale.



**Figura 11.11** I/O senza una buffer cache unificata.

Alcuni sistemi riservano una sezione separata della memoria centrale come **cache del disco**, dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una **cache delle pagine** per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system; l'uso degli indirizzi virtuali è molto più efficiente dell'uso dei blocchi fisici di disco. Diversi sistemi, compreso Solaris, Linux, Windows NT, 2000 e XP, usano le cache delle pagine, sia per le pagine relative ai processi sia per i dati dei file. Questo metodo è noto come **memoria virtuale unificata**.

Alcune versioni di UNIX e Linux prevedono la cosiddetta **buffer cache unificata**. Per illustrarne i vantaggi si considerino le due possibilità di aprire un file e accedervi: l'uso della mappatura dei file in memoria (Paragrafo 9.7) e l'uso delle ordinarie chiamate di sistema `read()` e `write()`. Senza una buffer cache unificata, si verifica una situazione simile a quella illustrata nella Figura 11.11. In questo caso, le chiamate di sistema `read()` e `write()` passano attraverso la buffer cache. La chiamata con associazione alla memoria richiede l'uso di due cache, la cache delle pagine e la buffer cache. L'associazione alla memoria prevede la lettura dei blocchi di disco dal file system e la loro memorizzazione nella buffer cache. Poiché il sistema di memoria virtuale non può interfacciarsi con la buffer cache, si deve copiare nella cache delle pagine il contenuto del file presente nella buffer cache. Questa situazione è nota come **double caching** proprio perché i dati del file system richiedono un doppio passaggio di cache. Non solo ciò comporta uno spreco di memoria, ma anche uno spreco di cicli della CPU e di I/O dovuti a un ulteriore trasferimento di dati nella memoria del sistema. Inoltre, eventuali incoerenze tra le due cache possono generare errori nella memorizzazione dei dati nei file. Con una buffer cache unificata, sia l'associazione alla memoria sia le chiamate di sistema `read()` e `write()` usano la stessa cache delle pagine,



**Figura 11.12** I/O con una buffer cache unificata.

con il vantaggio di evitare il double caching e di permettere al sistema di memoria virtuale di gestire dati del file system. La Figura 11.12 illustra l’uso della buffer cache unificata.

Indipendentemente dalla gestione delle cache per blocchi di disco oppure per pagine (o per entrambi), l’algoritmo LRU è in generale ragionevole per la sostituzione dei blocchi o delle pagine. Tuttavia, l’evoluzione degli algoritmi di gestione delle cache delle pagine usati dal sistema operativo Solaris rivela le difficoltà nella scelta di un algoritmo ottimale. Tale sistema operativo permette ai processi e alla cache delle pagine di condividere la memoria inutilizzata; prima della versione 2.5.1, non si facevano distinzioni tra l’allocazione delle pagine a un processo o alla cache delle pagine, con la conseguenza che un sistema che eseguiva molte operazioni di I/O usava la maggior parte della memoria disponibile per la cache delle pagine. A causa dell’alta frequenza delle operazioni di I/O, quando la memoria libera diventa troppo esigua, il modulo di scansione delle pagine (Paragrafo 9.10.2) sottrae pagine ai processi anziché alla cache delle pagine. In Solaris 2.6 e in Solaris 7 è stata realizzata in forma opzionale la tecnica di *paginazione con priorità*, secondo la quale il modulo di scansione delle pagine dà la priorità alle pagine dei processi rispetto a quelle della cache delle pagine. Il sistema operativo Solaris 8 ha aggiunto un limite prefissato tra pagine dei processi e cache delle pagine per il file system, impedendo a ciascun meccanismo di sottrarre totalmente la memoria all’altro. Con Solaris 9 e 10 sono stati nuovamente modificati gli algoritmi per migliorare l’uso della memoria e contenere il fenomeno della paginazione degenere.

Ci sono altri aspetti che possono influenzare le prestazioni di I/O, come quelli che riguardano la necessità di scritture sincrone o asincrone. Le **scritture sincrone** avvengono nell’ordine in cui le riceve il sottosistema per la gestione del disco e non subiscono la memorizzazione transitoria. Quindi la procedura chiamante prima di proseguire deve attendere che i dati raggiungano l’unità a disco. Nella maggior parte dei casi si usano **scritture asincrone**. Nelle scritture asincrone si memorizzano i dati nella cache e si restituisce immediatamente il controllo alla procedura chiamante. Le scritture dei metadati, tra le altre, possono essere sincrone. I sistemi operativi spesso includono un flag nella chiamata di sistema `open()` per permettere a un processo di richiedere che le operazioni di scrittura si eseguano in modo sincrono. I sistemi di gestione delle basi di dati ad esempio usano questa funzione per realizzare le transazioni atomiche, in modo da assicurare che i dati raggiungano la memoria stabile nell’ordine richiesto.

Alcuni sistemi ottimizzano la cache delle pagine adottando, secondo il tipo d'accesso ai file, differenti algoritmi di sostituzione. Le pagine relative a un file da leggere o scrivere in modo sequenziale non si dovrebbero sostituire nell'ordine LRU, infatti la pagina usata più di recente sarà usata nuovamente per ultima, o forse mai. Gli accessi sequenziali si potrebbero invece ottimizzare con tecniche note come rilascio indietro e lettura anticipata. Il **rilascio indietro** (*free-behind*) rimuove una pagina dalla memoria di transito non appena si verifica una richiesta della pagina successiva; le pagine precedenti con tutta probabilità non saranno più usate e quindi sprecano spazio in memoria di transito. Con la **lettura anticipata** (*read-ahead*) si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive: è probabile che queste pagine siano richieste una volta terminata l'elaborazione della pagina corrente. Il recupero di questi dati dal disco con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole. La presenza nel controllore di una cache per le tracce non elimina la necessità di adottare la tecnica di lettura anticipata in un sistema multiprogrammato, ciò a causa dell'elevata latenza e del sovraccarico determinato dai tanti piccoli trasferimenti dalla cache per le tracce alla memoria centrale; il ricorso alla *lettura anticipata* è vantaggioso.

La cache delle pagine, il file system e i driver del disco interagiscono in modi interessanti. Quando i dati vengono scritti su un file del disco, le pagine sono memorizzate nella cache, che qui funge da buffer, mentre il driver del disco ordina la propria coda di dati in uscita in base all'indirizzo sul disco. Queste due azioni consentono al driver del disco di minimizzare gli spostamenti della testina del disco e fanno sì che la scrittura dei dati rispetti i tempi ottimali per la rotazione del disco. A meno che le scritture richieste siano sincrone, un processo, per scrivere sul disco, scrive direttamente nella cache: il sistema trasferisce infatti i dati su disco, in maniera asincrona, quando lo ritiene opportuno. Dal punto di vista del processo utente, le scritture sembreranno estremamente rapide. Durante la lettura, l'I/O a blocchi procede a qualche lettura anticipata; ma, in realtà, le scritture si giovano della modalità asincrona molto più delle letture. Per grandi quantità di dati, quindi, la scrittura su disco tramite il file system è spesso più veloce della lettura, contrariamente a ciò che suggerirebbe l'intuizione.

## 11.7 Ripristino

Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati o la loro incoerenza. In questo paragrafo vedremo anche come un sistema può essere ripristinato in seguito a un malfunzionamento.

Un crollo del sistema può causare incoerenze tra le strutture dati del file system su disco, come le strutture delle directory, i puntatori ai blocchi liberi e i puntatori agli FCB liberi. Molti file system applicano delle modifiche direttamente a queste strutture. Operazioni comuni come la creazione di un file possono comportare molti cambiamenti strutturali all'interno del file system di un disco. Le strutture delle directory vengono modificate, gli FCB e i blocchi di dati allocati e i contatori liberi per tutti questi blocchi diminuiti. Quando queste modifiche sono interrotte da un crollo del sistema, ne possono derivare incoerenze tra le strutture. Ad esempio, il contatore degli FCB liberi potrebbe indicare che un FCB è stato allocato, ma la struttura della directory potrebbe non avere un puntatore a quel FCB. L'utilizzo della cache che i sistemi operativi adottano per ottimizzare le prestazioni di I/O aggrava questo problema. Alcuni cambiamenti potrebbero andare direttamente sul disco, mentre gli

altri possono finire nella cache. Se i cambiamenti nella cache non raggiungono il disco prima che si verifichi un crollo, è possibile che la situazione peggiori ulteriormente.

Inoltre, anche i bachi nell'implementazione del file system, i controllori del disco, e persino le applicazioni utente possono indurre errori nel file system. I file system hanno variati metodi per affrontare queste circostanze, a seconda delle strutture dati e degli algoritmi del file system. Ci occuperemo adesso di questi temi.

### 11.7.1 Verifica della coerenza

Quale che sia la causa degli errori, un file system deve prima scoprire i problemi e poi correggerli. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Sfortunatamente, questo procedimento richiederà diversi minuti, o addirittura delle ore, e avverrà tutte le volte che il sistema si avvia. In alternativa, un file system può registrare il suo stato all'interno dei metadati del file system. All'inizio di ogni serie di modifiche dei metadati è impostato un bit di stato per indicare che i metadati sono in stato di modifica. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tuttavia il bit dello stato rimane impostato, entra in funzione un verificatore della coerenza.

Il verificatore della coerenza – un programma di sistema come `fsck` in UNIX o `chkdsk` in Windows – confronta i dati delle directory con quelli contenuti nei blocchi dei dischi, tentando di correggere ogni incoerenza. Gli algoritmi di allocazione e di gestione dello spazio libero determinano il genere di problemi che questo programma può riconoscere e con quanto successo riuscirà a risolverli. Ad esempio, se si adotta uno schema di allocazione concatenata con un puntatore da ciascun blocco al successivo, si può ricostruire l'intero file e ricreare il corrispondente elemento nella directory analizzando i blocchi di dati. Diversamente, la perdita di un elemento di una directory in un sistema ad allocazione indirizzata potrebbe essere disastrosa, poiché ogni blocco di dati non contiene alcuna informazione sugli altri blocchi di dati. Per questo motivo, UNIX gestisce tramite cache gli elementi delle directory per le letture, mentre qualsiasi operazione di scrittura di dati che produca l'allocazione di spazio, o altre modifiche dei metadati, è svolta in modo sincrono, prima della scrittura dei corrispondenti blocchi di dati. Naturalmente nuovi problemi possono ancora insorgere se una scrittura sincrona viene interrotta da un crollo di sistema.

### 11.7.2 File system con registrazione delle modifiche

Spesso nell'informatica si adottano algoritmi e tecnologie anche in aree diverse da quelle per le quali sono stati progettati. È il caso degli algoritmi per il ripristino, sviluppati nell'area dei sistemi di gestione delle basi di dati, basati sulla registrazione delle modifiche, descritti nel Paragrafo 6.9.2. Questi algoritmi sono stati applicati con successo al problema della verifica della coerenza, realizzando i file system orientati alle transazioni e basati sulla registrazione delle modifiche (*log-based transaction-oriented file system*), noti anche come file system annotati (*journaling file system*).

Si osservi che l'approccio della verifica della coerenza esaminato precedentemente permette in sostanza alle strutture di esibire incoerenze successivamente corrette grazie al ripristino. Questa strategia comporta tuttavia alcuni problemi. Per esempio, l'incoerenza potrebbe rivelarsi irreparabile. Inoltre, il verificatore della coerenza potrebbe non essere in grado di ripristinare le strutture, con una conseguente perdita di file o addirittura di intere directory. Ancora, il verificatore della coerenza potrebbe richiedere l'intervento umano per risolvere i conflitti, il che causa inconvenienti: in mancanza di assistenza da parte di qualcuno, il sistema potrebbe essere inutilizzabile finché una persona non gli indichi come procedere. Infine,

il verificatore della coerenza sottrae risorse al sistema: per controllare un terabyte di dati possono essere necessarie molte ore.

La soluzione a questo problema consiste nell'applicare agli aggiornamenti dei metadati relativi al file system metodi di ripristino basati sulla registrazione delle modifiche. Sia il file system NTFS sia il Veritas usano questo metodo, che è anche opzionale rispetto al file system UFS nel Solaris 7 e nelle versioni successive. In realtà, sta diventando un metodo comune in molti sistemi operativi.

Fondamentalmente, tutte le modifiche dei metadati si annotano in modo sequenziale in un file di registrazione, detto *log*. Ogni insieme di operazioni che esegue uno specifico compito si chiama **transazione**. Una volta che le modifiche sono riportate nel file di registrazione, le operazioni si considerano portate a termine con successo (*committed*) e la chiamata di sistema può restituire il controllo al processo utente, permettendogli di proseguire la sua esecuzione. Nel frattempo, si applicano alle effettive strutture del file system le operazioni scritte nel log, e man mano che si eseguono si aggiorna un puntatore che indica quali azioni sono state completate e quali sono ancora incomplete. Quando un'intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare. I **buffer circolari** scrivono fino alla fine dello spazio disponibile, e poi ricominciano dall'inizio, sovrascrivendo i vecchi contenuti. Naturalmente, si devono prendere delle misure per evitare che dati non ancora salvati siano sovrascritti. Il log si potrebbe mantenere in una sezione separata del file system, o anche in un disco separato. È più efficiente, anche se è più complesso, averlo sotto testine di lettura e scrittura separate, poiché si riducono le situazioni di contesa della testina e i tempi di ricerca (*seek time*).

Se si verifica un crollo del sistema, nel log ci potranno essere zero o più transazioni. Le transazioni presenti non sono mai state ultimate nel file system, anche se il sistema operativo le definisce portate a termine con successo, e quindi si devono completare. Le transazioni si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento, e le strutture del file system rimangono coerenti. L'unico problema che si può presentare è il caso in cui una transazione sia fallita (*aborted*), cioè non sia stata dichiarata terminata con successo prima del crollo del sistema. In questo caso, si devono annullare tutti i cambiamenti che erano stati applicati al file system dalla transazione, di nuovo mantenendo la coerenza del file system. Questo ripristino è tutto ciò che è necessario fare dopo un crollo del sistema, eliminando tutti i problemi concernenti la verifica della coerenza.

Un vantaggio indiretto dell'uso dell'annotazione in un disco degli aggiornamenti dei metadati è che gli aggiornamenti sono molto più rapidi di quelli che si applicano direttamente alle strutture dati nei dischi. La ragione di questo miglioramento sta nel vantaggio, dal punto di vista delle prestazioni, dell'I/O ad accesso sequenziale rispetto a quello ad accesso diretto. Le onerose operazioni di scrittura dei metadati ad accesso diretto e sincrono si sostituiscono con molto meno gravose operazioni di scrittura sincrone ma ad accesso sequenziale nell'area di registrazione delle modifiche di un file system con annotazione delle modifiche. I cambiamenti determinati da quelle operazioni si riportano successivamente in modo asincrono nelle strutture appropriate nei dischi attraverso operazioni di scrittura ad accesso diretto. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file.

### 11.7.3 Altre soluzioni

Un'altra alternativa alla verifica della coerenza è impiegata dal file system WAFL di Network Appliance e da ZFS di Sun. Entrambi i sistemi non sovrascrivono mai i blocchi con nuovi dati; al contrario, una transazione scrive tutti i cambiamenti di dati e metadati su nuovi

blocchi. Quando la transazione viene completata, le strutture di metadati che puntavano alla vecchia versione di questi blocchi sono aggiornate in modo da puntare ai nuovi. Il file system può quindi rimuovere i vecchi puntatori e i vecchi blocchi per renderli nuovamente disponibili. Se i vecchi puntatori e i vecchi blocchi vengono mantenuti, viene creata una **snapshot (istantanea)**, cioè un'immagine del file system prima dell'ultimo aggiornamento. Questa soluzione non dovrebbe richiedere una verifica della coerenza se l'aggiornamento del puntatore è eseguito automaticamente. Ciononostante, il WAFL possiede comunque un controllore della coerenza, perché alcuni tipi di malfunzionamento possono ancora causare un errore nei metadati (per dettagli sul sistema WAFL si veda il Paragrafo 11.9).

In merito alla coerenza del disco lo ZFS di Sun presenta un approccio ancora più innovativo. ZFS non sovrascrive mai i blocchi, come WAFL, ma è in grado di andare oltre fornendo un riassunto delle verifiche su tutti i metadati e i blocchi di dati. Questa soluzione (se combinata con RAID) assicura che i dati siano sempre corretti. ZFS non possiede quindi un controllore della coerenza. (Maggiori dettagli su ZFS sono presenti nel Paragrafo 12.7.6.)

### 11.7.4 Copie di riserva e recupero dei dati

Poiché si possono verificare malfunzionamenti e perdite di dati anche nei dischi magnetici, è necessario preoccuparsene e provvedere affinché i dati non vadano persi definitivamente. A questo scopo si possono usare programmi di sistema che consentano di fare delle **copie di riserva (backup)** dei dati residenti nei dischi in altri dispositivi di registrazione di dati, come dischetti, nastri magnetici, dischi ottici o hard disk supplementari. Il ripristino della situazione antecedente la perdita di un singolo file, o del contenuto di un intero disco, richiederà il **recupero (restore)** dei dati dalle copie di riserva.

Al fine di ridurre al minimo la quantità di dati da copiare, è possibile sfruttare le informazioni contenute nell'elemento della directory associato a ogni file. Ad esempio, se il programma di creazione delle copie di riserva sa quando è stata eseguita l'ultima copia di riserva di un file, e se la data di ultima scrittura di quel file, registrata nella directory, indica che il file da quel momento non ha subito variazioni, non sarà necessario copiare nuovamente il file. Quella che segue è una tipica sequenza di gestione delle copie di riserva.

- ◆ **Giorno 1.** Copiatura nel supporto di backup delle copie di riserva di tutti i file contenuti nel disco; detta **copiatura completa**.
- ◆ **Giorno 2.** Copiatura su un altro supporto di tutti i file modificati dal Giorno 1; si tratta di una **copiatura incrementale**.
- ◆ **Giorno 3.** Copiatura su un altro supporto di tutti i file modificati dal Giorno 2.
- 
- 
- 
- ◆ **Giorno  $n$ .** Copiatura su un altro supporto di tutti i file modificati dal Giorno  $n - 1$ . Ritorno al Giorno 1.

Il nuovo ciclo può comportare la scrittura delle nuove copie di riserva nel primo insieme di supporti di backup, oppure in un nuovo insieme; in questo modo si ha la possibilità di recuperare il contenuto dell'intero disco iniziando le operazioni di recupero dalla copia di riserva completa e proseguendo con le copie di riserva incremental. Naturalmente, più grande è  $n$ , maggiore sarà il numero di nastri o dischi da leggere per un completo recupero. Un ulteriore vantaggio di questo ciclo di creazione di copie di riserva è la possibilità di recupe-

rare qualsiasi file accidentalmente cancellato durante il ciclo, recuperandolo dalle copie del giorno precedente. La lunghezza del ciclo è un compromesso tra la quantità di supporti per le copie di riserva necessari e il numero di giorni addietro da cui si può compiere un'operazione di recupero. La lunghezza del ciclo è un compromesso fra la quantità di spazio richiesta dalle copie di riserva e il numero di giorni addietro da cui si può compiere un'operazione di recupero. Una possibilità per diminuire la quantità di nastri che è necessario leggere per portare a termine il ripristino è di eseguire inizialmente una copia di riserva completa, per poi eseguire copie dei soli file modificati nei giorni successivi. In questo modo, il ripristino può fondarsi sull'ultima copia incrementale, insieme all'ultima copia completa, senza necessità di altre copie incrementalì. Qui il compromesso da tenere a mente è che il numero di file modificati aumenta giornalmente: ogni nuova copia incrementale, quindi, richiede più spazio.

Un utente potrebbe accorgersi dopo molto tempo che si è perso o si è danneggiato un particolare file. Per questa ragione si è soliti pianificare di tanto in tanto una copiatura completa che sarà conservata in modo permanente; quindi il supporto che la contiene non sarà riutilizzato. È inoltre opportuno conservare tali copie di riserva permanenti lontano dalle copie ordinarie per proteggerle dai vari pericoli, ad esempio un incendio che può distruggere il calcolatore e tutte le copie di riserva. Se il ciclo di creazione delle copie di riserva prevede il reimpiego dei mezzi che le contengono, è anche necessario aver cura di non usarli troppe volte: se dovessero logorarsi, potrebbe essere impossibile ripristinare i dati in essi contenuti.

## 11.8 NFS

---

I file system di rete sono di particolare interesse; in genere integrano l'intera struttura della directory e l'interfaccia del sistema client. L'NFS è un buon esempio di file system di rete client-server ampiamente usato e ben realizzato, che in questo paragrafo s'impiega per esplorare i particolari che caratterizzano la realizzazione dei file system di rete.

L'NFS è sia una realizzazione sia una definizione di un sistema per l'accesso a file remoti attraverso LAN, o anche WAN. Fa parte dell'ONC+, adottato dalla maggior parte dei distributori del sistema operativo UNIX e in alcuni sistemi operativi per PC. La versione qui descritta fa parte del sistema operativo Solaris, che è a sua volta una versione modificata dello UNIX SVR4, delle stazioni di lavoro Sun e altre architetture. Esso usa i protocolli TCP/IP o i protocolli UDP/IP (secondo la rete di comunicazione). La definizione e realizzazione dell'NFS, nella sua descrizione fornita in questa sede, si accavallano: per ogni descrizione dettagliata si fa riferimento alla versione realizzata dalla Sun; mentre ogni descrizione sufficientemente generale vale anche per la sua definizione.

Ci sono numerose versioni di NFS, l'ultima delle quali è la Versione 4. Esamineremo qui la Versione 3, attualmente la più utilizzata.

### 11.8.1 Generalità

Nel contesto dell'NFS si considera un insieme di stazioni di lavoro interconnesse come un insieme di calcolatori indipendenti con file system indipendenti. Lo scopo è quello di permettere un certo grado di condivisione tra i file system, su richiesta esplicita, in modo trasparente. La condivisione è basata su una relazione client-server. Un calcolatore può essere, come spesso accade, sia un client sia un server. La condivisione è ammessa tra ogni coppia di calcolatori, anziché essere limitata ai calcolatori aventi la specifica funzione di server. Per as-

sicurare l'indipendenza dei calcolatori, la condivisione di un file system remoto ha effetto esclusivamente sul calcolatore client.

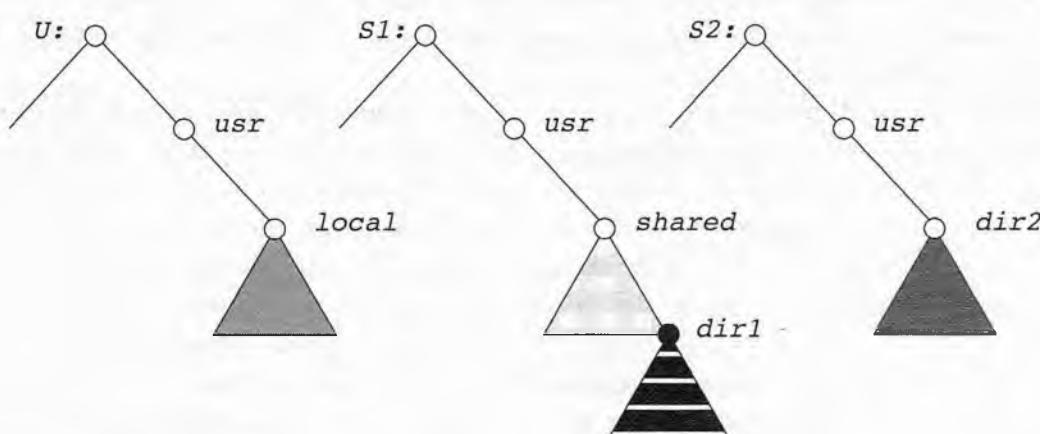
Affinché una directory remota sia accessibile in modo trasparente a un calcolatore particolare, ad esempio  $C_1$ , un client di quel calcolatore deve prima eseguire un'operazione di montaggio. La semantica dell'operazione consiste nel fatto che una directory remota si monta in corrispondenza di una directory di un file system locale. Una volta completata l'operazione di montaggio, la directory montata assume l'aspetto di un sottoalbero integrante del file system locale, e sostituisce il sottoalbero che discende dalla directory locale; questa, a sua volta, rappresenta la radice della directory appena montata. La directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito; si deve fornire la locazione (o il nome del calcolatore) della directory remota. Tuttavia, da questo momento in poi gli utenti del calcolatore  $C_1$  possono accedere ai file della directory remota in modo del tutto trasparente.

Per illustrare il montaggio dei file system, si consideri il file system riportato nella Figura 11.13, dove i triangoli rappresentano i sopraccitati sottoalberi di directory. La figura illustra tre file system di calcolatori indipendenti chiamati  $U$ ,  $S_1$  e  $S_2$ . A questo punto, in ogni calcolatore si può accedere solo a file locali. Nella Figura 11.14(a) sono riportati gli effetti del montaggio di  $S_1:/usr/shared$  in  $U:/usr/local$ . In questa figura è illustrata la visione che gli utenti di  $U$  hanno del loro file system. Occorre osservare che, una volta completato il montaggio, essi possono accedere a qualsiasi file che si trovi nella directory  $dir1$ , ad esempio usando il prefisso  $/usr/local/dir1$  in  $U$ . La directory originale  $/usr/local$  di quel calcolatore non è più visibile.

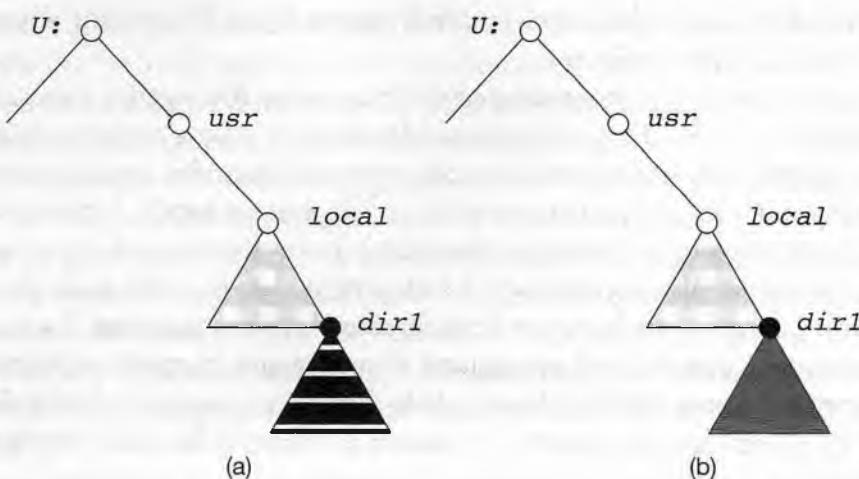
Potenzialmente ogni file system o ogni directory in un file system, nel rispetto dei diritti d'accesso, si possono montare in modo remoto in cima a qualsiasi directory locale. Le stazioni di lavoro prive di dischi possono montare i propri file system prelevandoli dai server.

In alcune versioni dell'NFS sono permessi anche i montaggi a cascata; ciò significa che un file system si può montare in corrispondenza di un altro file system montato in modo remoto. Un calcolatore è tuttavia sottoposto ai soli montaggi da esso richiesti.

Montando un file system remoto, il client non acquisisce l'accesso ai file system che erano montati sopra il primo; così, il meccanismo di montaggio non ha la proprietà transitiva. Nella Figura 11.14(b) sono riportati i montaggi a cascata relativi all'esempio precedente. Nella figura è riportato il risultato del montaggio di  $S_2:/usr/dir2$  in  $U:/usr/local/dir1$ , che è già stato montato in modo remoto da  $S_1$ . Gli utenti di  $U$  possono ac-



**Figura 11.13** Tre file system indipendenti.



**Figura 11.14** Montaggio nell’NFS; (a) montaggio; (b) montaggi a cascata.

cedere ai file di `dir2` usando il prefisso `/usr/local/dir1`. Se si monta un file system condiviso in corrispondenza delle directory iniziali degli utenti di tutti i calcolatori di una rete, un utente può aprire una sessione in qualsiasi stazione di lavoro e prelevare il proprio ambiente di lavoro iniziale. Questa proprietà permette la **mobilità dell’utente**.

Uno degli scopi nella progettazione dell’NFS era quello di operare in un ambiente eterogeneo di calcolatori, sistemi operativi e architetture di rete. La definizione dell’NFS è indipendente da questi mezzi e quindi incoraggia altre realizzazioni. Questa indipendenza si ottiene usando primitive RPC costruite su un protocollo di rappresentazione esterna dei dati (*external data representation*, XDR) usato tra due interfacce indipendenti dalla realizzazione. Quindi, se il sistema è formato da calcolatori e file system eterogenei adeguatamente interfacciati all’NFS, si possono montare file system di diversi tipi, sia localmente sia in modo remoto.

La definizione dell’NFS distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi d’accesso ai file remoti. Di conseguenza, per questi servizi si definiscono due protocolli distinti: un protocollo di montaggio e un protocollo per gli accessi ai file remoti, il **protocollo NFS**. I protocolli sono definiti come insiemi di RPC, gli elementi di base usati per realizzare, in modo trasparente, l’accesso remoto ai file.

### 11.8.2 Protocollo di montaggio

Il **protocollo di montaggio** stabilisce la connessione logica iniziale tra un server e un client. Nella versione di Sun Microsystems ogni calcolatore ha un processo server, esterno al kernel, che esegue le funzioni del protocollo.

Un’operazione di montaggio comprende il nome della directory remota da montare e il nome del calcolatore server in cui tale directory è memorizzata. La richiesta di montaggio si associa alla RPC corrispondente e s’invia al server di montaggio in esecuzione nello specifico calcolatore server. Il server conserva una **lista di esportazione** (la `/etc/dfs/dfstab` nel sistema Solaris, modificabile soltanto da un *superuser*) che specifica i file system locali esportati per il montaggio e i nomi dei calcolatori a cui tale operazione è permessa. La lista può comprendere anche i diritti d’accesso, come la sola scrittura. Per semplificare la manutenzione delle liste di esportazione e delle tabelle di montaggio, si può usare uno schema distribuito di nominazione per contenere queste informazioni e renderle disponibili agli appropriati client.

Occorre ricordare che qualsiasi directory all’interno di un file system esportato si può montare in modo remoto da un calcolatore accreditato. Di conseguenza, un’unità compo-

nente è rappresentata da una directory di questo tipo. Quando il server riceve una richiesta di montaggio conforme alla propria lista di esportazione, riporta al client un **handle del file** da usare come chiave per ulteriori accessi ai file che si trovano all'interno del file system montato. L'handle del file contiene tutte le informazioni di cui ha bisogno il server per gestire un proprio file. Nei termini dell'ambiente UNIX, l'handle del file è composto da un identificatore di file system e da un numero di *inode* per identificare la directory montata all'interno del file system esportato.

Il server contiene anche una lista dei calcolatori client e delle corrispondenti directory correntemente montate. Questa lista si usa soprattutto per scopi amministrativi, ad esempio per informare i client che un server sta andando fuori servizio. L'aggiunta o la cancellazione di elementi da questa lista sono gli unici modi in cui il protocollo dell'operazione di montaggio può modificare lo stato del server.

Generalmente un sistema ha una configurazione di montaggio predefinita che si stabilisce nella fase d'avviamento (`/etc/vfstab` in Solaris); tale configurazione si può comunque modificare. Oltre alla procedura di montaggio effettiva, il protocollo di montaggio comprende numerose altre procedure, come lo smontaggio e la restituzione della lista d'esportazione.

### 11.8.3 Protocollo NFS

Il protocollo NFS offre un insieme di RPC per operazioni su file remoti che svolgono le seguenti operazioni:

- ◆ ricerca di un file in una directory;
- ◆ lettura di un insieme di elementi di una directory;
- ◆ manipolazione di collegamenti e di directory;
- ◆ accesso ad attributi di file;
- ◆ lettura e scrittura di file.

Queste procedure si possono invocare soltanto dopo aver stabilito un handle del file per la directory montata in modo remoto.

L'omissione delle operazioni `open()` e `close()` è intenzionale. Una caratteristica importante dei server NFS è l'*assenza dell'informazione di stato*. I server non conservano informazioni sui loro client da un accesso all'altro. Non esistono parallelismi con la tabella dei file aperti o le strutture di file di UNIX da parte del server, quindi ogni richiesta deve fornire un insieme completo di argomenti, tra cui un identificatore unico di file e uno scostamento assoluto all'interno del file per svolgere le operazioni appropriate. La struttura che ne deriva è robusta; non si devono prendere misure speciali per ripristinare un server dopo un guasto. Per tale ragione, le operazioni sui file devono essere idempotenti. Ciascuna richiesta dell'NFS ha un numero di sequenza, che permette al server di determinare la duplicazione o la mancanza di richieste.

La presenza della suddetta lista di client sembra violare la proprietà dell'assenza di informazione di stato del server. Tuttavia, essa non è essenziale ai fini del corretto funzionamento del client o del server, quindi non è necessario recuperare tale lista dopo il crollo di un server; tale lista può contenere anche dati incoerenti e si considera come un semplice suggerimento.

Un'ulteriore implicazione della filosofia dei server senza informazione di stato e un risultato della sincronia di una RPC consiste nel fatto che i dati modificati, tra cui blocchi indiretti e di stato, si devono riscrivere nei dischi del server prima che i risultati siano riportati.

ti al client. Un client può cioè ricorrere a cache per i blocchi di scrittura, ma quando li invia al server, assume che abbiano raggiunto i dischi del server, che deve scrivere tutti i dati dell’NFS in modo sincrono. In questo modo il crollo di un server e il successivo ripristino saranno invisibili al client; tutti i blocchi che il server gestisce per il client resteranno intatti. La conseguente perdita di prestazioni può essere rilevante poiché si perdonano i vantaggi derivanti dall’impiego di una cache. Le prestazioni si possono incrementare impiegando dispositivi di memoria secondaria con una propria cache non volatile (di solito si tratta di memorie alimentate da una batteria). Il controllore del disco riporta che la scrittura nel disco è avvenuta quando la scrittura è avvenuta nella cache non volatile. Essenzialmente, il calcolatore “vede” una scrittura sincrona molto rapida. Questi blocchi restano intatti anche dopo un crollo del sistema, e periodicamente vengono trasferiti da tale memoria stabile al disco.

Di una singola chiamata di procedura di scrittura dell’NFS sono garantite l’atomicità e la non interferenza con altre chiamate di scrittura nello stesso file. Tuttavia, il protocollo NFS non fornisce meccanismi di controllo della concorrenza, e poiché ogni chiamata di scrittura o lettura dell’NFS può contenere non più di 8 KB di dati e i pacchetti UDP sono limitati a 1500 byte, può essere necessario dividere una chiamata di sistema `write()` in diverse RPC di scrittura; quindi, due utenti che scrivono nello stesso file remoto possono riscontrare interferenze nei loro dati. Poiché la gestione di meccanismi di bloccaggio richiede informazioni di stato, si richiede che un servizio esterno all’NFS debba fornire tali meccanismi, come nel caso di Solaris. Gli utenti sanno che per coordinare l’accesso ai file condivisi devono usare meccanismi che sono oltre la portata dell’NFS.

L’NFS è integrato nel sistema operativo tramite un VFS. Per illustrarne l’architettura si può accennare al modo in cui si gestisce un’operazione su un file remoto già aperto (Figura 11.15). Il client inizia l’operazione con un’ordinaria chiamata di sistema. Lo strato del sistema operativo fa corrispondere tale chiamata di sistema a un’operazione del VFS sull’opportuno *vnode*. Lo strato del VFS identifica il file come remoto e invoca l’opportuna procedura dell’NFS. Avviene quindi una chiamata RPC allo strato di servizio dell’NFS nel server remoto. Tale chiamata si reintroduce nello strato del VFS del sistema remoto, che riconosce essere locale e invoca l’appropriata operazione del file system. Questo cammino si ripercorre al contrario per restituire il risultato. Un vantaggio di tale architettura è che il client e il server sono identici; così un calcolatore può essere un client, un server o entrambi. L’effettivo servizio su ciascun server è eseguito da thread del kernel.

#### 11.8.4 Traduzione dei nomi di percorso

La traduzione dei nomi di percorso (*path-name translation*) del protocollo NFS prevede l’analisi sintattica di ogni percorso – per esempio `/usr/local/dir1/file.txt` – al fine di estrarre i nomi delle singole directory – nell’esempio: (1) `usr`, (2) `local` e (3) `dir1`. La traduzione dei nomi di percorso si compie suddividendo il percorso stesso in nomi componenti ed eseguendo una chiamata `lookup()` dell’NFS separata per ogni coppia formata da un nome componente e un *vnode* di directory. Quando s’incontra un punto di montaggio, la ricerca di un componente causa una RPC separata al server. Questo schema di attraversamento del nome di percorso è costoso ma necessario, poiché ogni client ha un’unica configurazione del proprio spazio di nomi logico, dettata dai montaggi che ha eseguito. Sarebbe stato molto più efficiente consegnare un nome di percorso a un server e ricevere un *vnode* d’arrivo una volta incontrato un punto di montaggio, tuttavia dovunque può essere presente un altro punto di montaggio per un particolare client sconosciuto al server senza informazione di stato.

Una cache per la ricerca dei nomi delle directory, nel sito del client, conserva i *vnode* per i nomi delle directory remote; in questo modo si accelerano i riferimenti ai file con lo

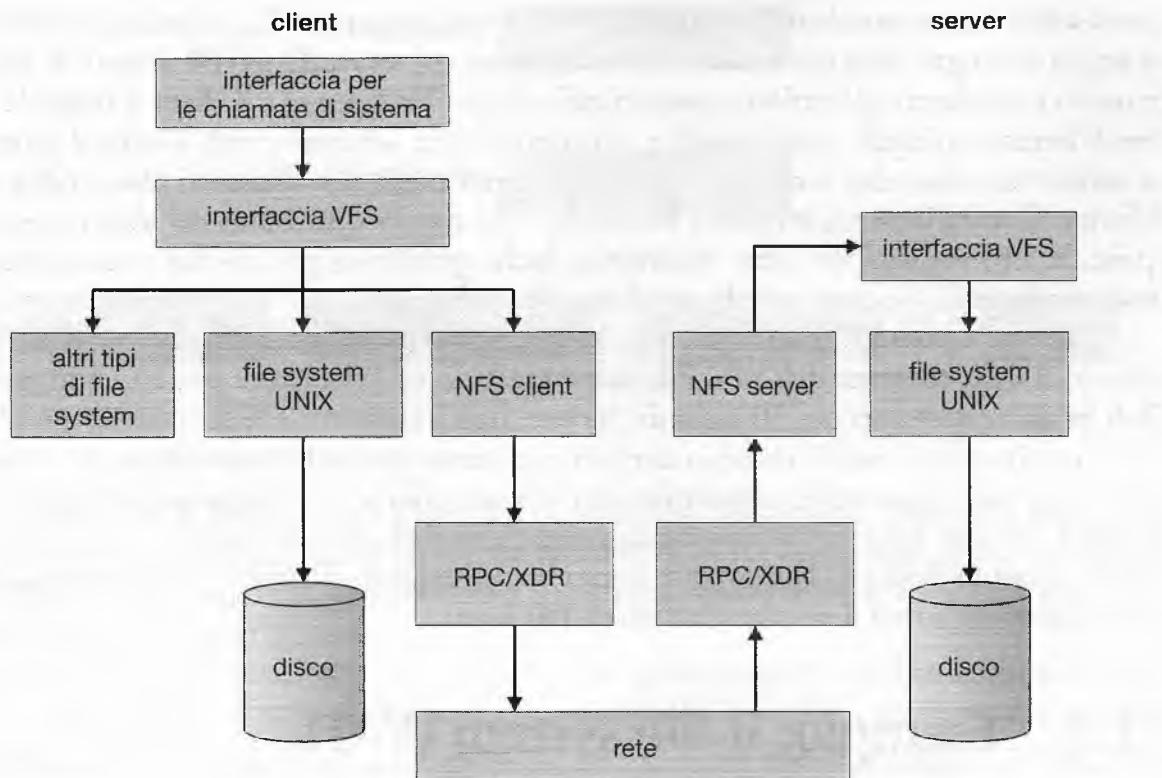


Figura 11.15 Schema dell'architettura dell'NFS.

stesso nome di percorso iniziale. Se gli attributi restituiti dal server non corrispondono agli attributi del *vnode* nella cache, si scarta il contenuto della cache della directory.

Occorre ricordare che nell'NFS è permesso il montaggio di un file system remoto in cima a un altro file system remoto già montato; si tratta del *montaggio a cascata*. Tuttavia, un server non può agire come intermediario tra un client e un altro server. Al contrario, un client deve stabilire un collegamento diretto client-server con il secondo server, montando direttamente la directory richiesta. Quando un client ha un montaggio a cascata, nel caso di un attraversamento di un nome di percorso può essere coinvolto più di un server. Tuttavia, ogni ricerca di componente si compie tra il client originale e alcuni server, perciò quando un client fa una ricerca in una directory sulla quale il server ha montato un file system, il client vede la directory sottostante e non la directory montata.

### 11.8.5 Funzionamento remoto

Eccetto che per l'apertura e la chiusura dei file, tra le normali chiamate di sistema di UNIX per operazioni su file e le RPC del protocollo NFS esiste una corrispondenza quasi da uno a uno. Quindi, un'operazione remota su un file si può tradurre direttamente nella RPC corrispondente. Dal punto di vista concettuale l'NFS aderisce al paradigma del servizio remoto, ma in pratica si usano tecniche di memorizzazione transitoria e cache per migliorare le prestazioni. Non c'è una corrispondenza diretta tra un'operazione remota e una RPC; le RPC prelevano blocchi e attributi del file che memorizzano localmente nelle cache. Le successive operazioni remote usano i dati nella cache, soggetti a vincoli di coerenza.

Esistono due cache: la cache degli attributi dei file (informazioni sugli *inode*) e la cache dei blocchi di file. Su un file aperto, il kernel fa un controllo rispetto al server remoto per stabilire se deve prelevare o riconvalidare gli attributi nella cache: i blocchi di file nella cache si

usano solo se i corrispondenti attributi nella cache sono aggiornati. La cache degli attributi viene aggiornata ogni volta che arrivano nuovi attributi dal server. Dopo 60 secondi si scartano, in modo predefinito, gli attributi presenti nella cache. Tra il server e il client si usano le tecniche di lettura anticipata (*read-ahead*) e scrittura differita (*delayed-write*). Finché il server non ha confermato che i dati sono stati scritti nei dischi, i client non liberano i blocchi di scrittura differita. Contrariamente a quanto accade nel file system distribuito del sistema operativo Sprite, la scrittura differita viene mantenuta anche quando si apre un file in concorrenza in modi conflittuali. Ne deriva che la semantica UNIX (Paragrafo 10.5.3.1) non si conserva.

Mettere a punto il sistema per migliorare le prestazioni rende difficile caratterizzare la semantica della coerenza dell’NFS. File nuovi creati in un calcolatore possono non essere visibili in altri calcolatori per 30 secondi. Inoltre, non è stabilito se le scritture eseguite in un file in un sito siano visibili anche in altri siti che hanno aperto lo stesso file per la lettura. Le nuove aperture di quel file consentono di osservare solo le modifiche già inviate al server, quindi l’NFS non fornisce né una stretta emulazione della semantica UNIX, né la semantica delle sessioni di Andrew. Nonostante questi inconvenienti, l’utilità e le alte prestazioni del meccanismo ne fanno il sistema distribuito più usato.

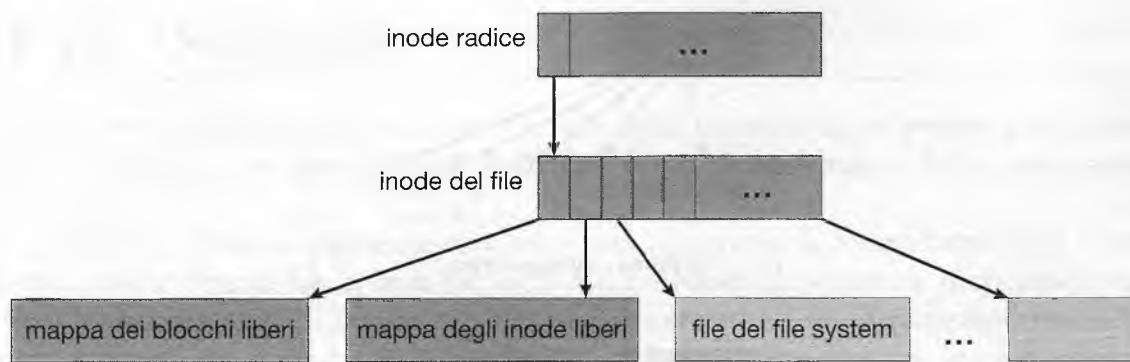
## 11.9 Esempio: il file system WAFL

L’I/O da e verso il disco si riflette significativamente sulle prestazioni del sistema. Di conseguenza, i progettisti devono esercitare grande cura sia nella progettazione sia nell’implementazione del file system. Alcuni file system sono concepiti per finalità generali, ossia sono in grado di offrire prestazioni accettabili e funzionalità adatte a file che differiscono per tipo e dimensione, e a carichi di I/O diversi. Altri sono ottimizzati per compiti specifici, nel tentativo di fornire prestazioni migliori dei sistemi a carattere generale per alcune applicazioni dedicate. Un’ottimizzazione di questo genere è rappresentata dal file system WAFL di Network Appliance. WAFL, acronimo di *write-anywhere file layout* (“modello di file per la scrittura ovunque”), è un file system potente ed elegante, ottimizzato per le scritture casuali.

È utilizzato in esclusiva dai file server di rete prodotti da Network Appliance, coerentemente con la sua vocazione di file system distribuito. Benché sia stato originariamente progettato per i soli NFS e CFS, consente l’invio di file ai client tramite i protocolli NFS, CIFS, `ftp` e `http`. Quando molti client contattano un file server attraverso questi protocolli, le richieste di letture casuali, e ancor di più quelle relative a scritture casuali, aumentano sensibilmente. I protocolli NFS e CIFS trattengono in una cache i dati provenienti dalle operazioni in lettura: per i file server, dunque, la scrittura assume massima importanza.

L’impiego del WAFL presuppone che i file server dispongano di una cache NVRAM per le scritture. I progettisti del WAFL hanno sfruttato il vantaggio di lavorare su un’architettura specifica, con una cache per la memorizzazione stabile dei dati, al fine di ottimizzare il file system per l’I/O ad accesso casuale. Dal momento che è stato concepito espressamente per un’applicazione, uno dei principi che hanno ispirato il WAFL è la facilità d’uso. I suoi autori, inoltre, hanno aggiunto una nuova funzionalità di duplicazione istantanea, per creare a più riprese, come vedremo, diverse copie a sola lettura del file system.

Il file system, pur essendo simile al Berkeley Fast, presenta varie modifiche; è basato sui blocchi e usa gli inode per la descrizione dei file. Ciascun inode contiene 16 puntatori ad altrettanti blocchi (o blocchi indiretti), che appartengono al file descritto dall’inode. Ciascun file system possiede un inode radice. Come si vede nella Figura 11.16, tutti i metadati sono custoditi all’interno di file: un file per gli inode, un altro per la mappa dei blocchi li-



**Figura 11.16** File system WAFL.

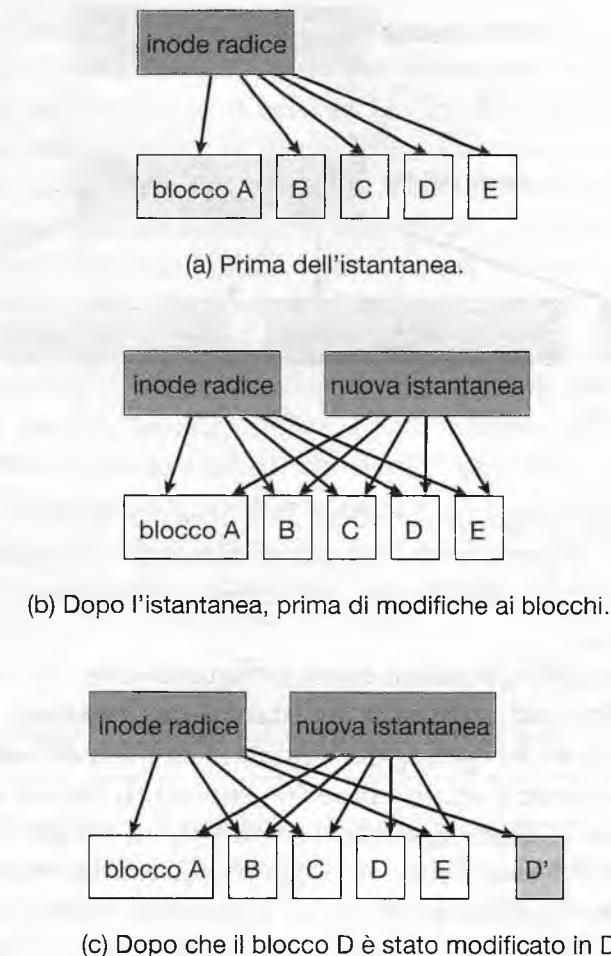
beri e un terzo per la mappa degli inode liberi. Poiché si tratta di file ordinari, i blocchi di dati non hanno un indirizzo predefinito, ma possono risiedere dovunque. Se un file system viene ampliato con l'aggiunta di dischi, esso aumenterà la lunghezza di questi file per i metadati in modo automatico.

Pertanto, i file system WAFL possono essere raffigurati come un albero di blocchi che si dipartono dall'inode radice; per catturarne un'istantanea (*snapshot*), WAFL crea una copia dell'inode radice. In seguito a ciò, ogni aggiornamento dei file o dei metadati occuperà blocchi nuovi, anziché sovrascrivere i relativi blocchi esistenti. Il nuovo inode radice punta ai metadati e ai dati nella loro versione aggiornata. Nello stesso tempo, il vecchio inode radice continua a puntare ai blocchi precedenti, non aggiornati, e in tal modo dà accesso a un'immagine del file system che ricalca esattamente il momento in cui era stato fotografato; lo spazio occupato sul disco per questa operazione è davvero esiguo. In sostanza, la copia richiede un supplemento di spazio sul disco equivalente ai soli blocchi che siano stati modificati dal momento in cui si è scattata l'istantanea.

Una differenza di rilievo in confronto a file system più tradizionali è data dalla mappa dei blocchi liberi, che possiede più di un bit per blocco. Si tratta di una maschera di bit che prevede un bit impostato a uno a ogni istantanea che stia utilizzando il blocco. Quando tutte le istantanee che hanno utilizzato un blocco sono cancellate, la maschera di bit associata è formata da zeri, e il blocco può essere riutilizzato. I blocchi usati non sono mai sovrascritti, di modo che le scritture avvengono a gran velocità, visto che le operazioni in scrittura possono sfruttare il blocco libero più vicino alla posizione corrente della testina. Vi sono, nel WAFL, molti altri modi per ottimizzare le prestazioni.

Possono coesistere allo stesso tempo numerose istantanee: se ne può scattare una per ogni ora del giorno e ogni giorno del mese. Gli utenti abilitati, consultando queste istantanee, hanno accesso a ciascuno dei file per come appariva nei vari momenti in cui è stato fotografato. Questa funzionalità è anche utile per le copie di riserva, per le fasi di test, per gestire diverse versioni di un progetto, e altro ancora. Si tratta di un'implementazione molto efficiente, che non richiede neppure di duplicare con la copiatura su scrittura ciascun blocco di dati prima che sia modificato. Altri file system offrono la medesima funzionalità, ma spesso con minor efficienza. Le istantanee del WAFL sono descritte nella Figura 11.17.

Le versioni più recenti del WAFL permettono istantanee di lettura e scrittura, note come cloni. Come le istantanee, anche i cloni sono efficienti, in quanto ne utilizzano le stesse tecniche. Una istantanea di sola lettura cattura lo stato del file system e un clone fa riferimento a quell'istantanea di sola lettura. Eventuali scritture sul clone sono memorizzate in nuovi blocchi e i puntatori del clone vengono aggiornati per fare riferimento ai nuovi blocchi. L'istantanea originale non è modificata, mantenendo così l'immagine del file system.



**Figura 11.17** Istantanee di WAFL.

prima dell'aggiornamento del clone. I cloni possono essere promossi alla sostituzione del file system originale e ciò comporta l'eliminazione dei vecchi puntatori e di ogni vecchio blocco a essi associato. I cloni sono utili per i controlli e gli aggiornamenti; la versione originale rimane infatti immutata e il clone viene cancellato quando il controllo è stato effettuato o quando l'aggiornamento fallisce.

Un'altra caratteristica che deriva naturalmente dall'implementazione del file system WAFL è la **replica**, ovvero la duplicazione e la sincronizzazione di un insieme di dati in un altro sistema attraverso una rete. Per prima cosa, l'istantanea di un file system WAFL viene duplicata in un altro sistema. Quando si scatta un'altra instantanea del sistema sorgente, per aggiornare il sistema remoto è sufficiente mandare tutti i blocchi contenuti nella nuova instantanea. Questi blocchi sono quelli che hanno subito modifiche nell'intervallo di tempo tra lo scatto delle due instantanee. Il sistema remoto aggiunge questi blocchi al file system e aggiorna i propri puntatori. Il nuovo sistema è dunque un duplicato del sistema sorgente nel momento in cui è stata scattata la seconda instantanea. La ripetizione di questo processo fa sì che il sistema remoto sia (quasi) una copia aggiornata del primo sistema. Le repliche sono utili per il ripristino da eventuali crash. Nel caso in cui il primo sistema sia vittima di una perdita totale di dati, la maggior parte dei suoi dati sarebbe comunque disponibile sulla replica del sistema remoto.

Infine è opportuno ricordare che il file system ZFS di Sun supporta le instantanee, i cloni e un sistema di repliche altrettanto efficienti.

## 11.10 Sommario

Il file system risiede permanentemente in memoria secondaria, progettata per contenere, permanentemente, una grande quantità di dati. Il più comune mezzo di memoria secondaria è il disco.

I dischi si possono segmentare in partizioni, allo scopo di controllarne l'uso e consentire più, anche diversi, file system per ogni disco. Questi file system si montano in un file system logico per renderli disponibili all'uso. I file system spesso si realizzano secondo una struttura stratificata o modulare: i livelli più bassi hanno a che fare con le caratteristiche fisiche dei dispositivi di memorizzazione; i livelli più alti hanno a che fare con i nomi simbolici e le caratteristiche logiche dei file; i livelli intermedi fanno corrispondere le caratteristiche logiche dei file alle caratteristiche fisiche dei dispositivi.

Ogni tipo di file system può avere diverse strutture e algoritmi. Uno strato VFS consente ai livelli superiori di aver a che fare con ciascun tipo di file system in modo uniforme. Nella struttura della directory del sistema si possono integrare anche i file system remoti sui quali si agisce con le ordinarie chiamate di sistema tramite l'interfaccia del VFS.

Lo spazio dei dischi può essere allocato ai file in tre modi: allocazione contigua, concatenata e indicizzata. L'allocatione contigua può risentire di frammentazione esterna. I file con accesso diretto non si possono gestire con l'allocatione concatenata. L'allocatione indicizzata, infine, può richiedere un notevole carico per il proprio blocco indice. Tali algoritmi si possono ottimizzare in molti modi. Lo spazio contiguo si può allargare attraverso delle estensioni allo scopo di aumentare la flessibilità e ridurre la frammentazione esterna. L'allocatione indicizzata si può realizzare in cluster per incrementare la produttività e ridurre il numero di elementi dell'indice necessari. L'indicizzazione in cluster di grandi dimensioni è analoga all'allocatione contigua con estensioni.

I metodi di allocatione dello spazio libero influenzano anche l'efficienza d'uso dello spazio dei dischi, le prestazioni del file system e l'affidabilità della memoria secondaria. I metodi usati comprendono i vettori di bit e le liste concatenate. Le ottimizzazioni comprendono il raggruppamento, il conteggio e la FAT, che colloca la lista concatenata in un'area contigua.

Le procedure di gestione delle directory devono tener conto dell'efficienza, delle prestazioni e dell'affidabilità. La tabella hash è il metodo usato più spesso; è veloce ed efficiente. Sfortunatamente, il danneggiamento di una tabella o il crollo del sistema possono causare discordanze tra le informazioni contenute nelle directory e il contenuto del disco. Con un verificatore di coerenza si può porre rimedio ai danni. Gli strumenti di creazione di copie di riserva (*backup*) del sistema operativo consentono la copiatura nelle unità a nastro dei dati contenuti nei dischi allo scopo di poterli ripristinare in seguito a perdite dovute a malfunzionamenti dei dispositivi fisici, errori del sistema operativo, o a errori degli utenti.

I file system di rete, come l'NFS, usano un metodo client-server per permettere agli utenti di accedere a file e directory in calcolatori remoti come se fossero in file system locali. Si traducono le chiamate di sistema del client nei protocolli di rete, per poi ritradurle in operazioni del file system nel server. L'interconnessione in reti e gli accessi di più client costituiscono una sfida nei campi della coerenza dei dati e delle prestazioni.

A causa del ruolo fondamentale che i file system hanno nel funzionamento di un sistema, le loro prestazioni e affidabilità sono fondamentali. Tecniche come quelle che impiegano le cache e i file di log migliorano le prestazioni, mentre i log e le tecniche RAID migliorano l'affidabilità. Il sistema WAFL è un esempio di ottimizzazione delle prestazioni per rispondere a uno specifico carico di I/O.

## Esercizi pratici

- 11.1 Prendete in considerazione un file costituito da 100 blocchi. Assumete che il blocco del file di controllo (e il blocco dell'indice, in caso di allocazione indicizzata) sia già in memoria. Calcolate quante operazioni di I/O del disco sono necessarie perché si realizzino strategie di allocazione contigue, concatenate e indicizzate (singolo livello), se, per un blocco, si mantengono le condizioni che seguono. Nel caso di allocazione contigua, assumete che non ci sia spazio di crescita all'inizio, ma solo alla fine. Assumete inoltre che il blocco di informazione da aggiungere sia salvato in memoria.
- Il blocco viene aggiunto all'inizio.
  - Il blocco viene aggiunto al centro.
  - Il blocco viene aggiunto alla fine.
  - Il blocco viene rimosso dall'inizio.
  - Il blocco viene rimosso dal centro.
  - Il blocco viene rimosso dalla fine.
- 11.2 Quali problemi potrebbero verificarsi se un sistema permettesse di montare il file system simultaneamente in più di una posizione?
- 11.3 Perché la mappa di bit per l'allocazione dei file deve essere conservata nella memoria di massa, e non nella memoria principale?
- 11.4 Considerate un sistema che supporta le strategie di allocazione contigua, concatenata e indicizzata. Quali criteri dovrebbero essere impiegati per decidere quale strategia è migliore per un dato file?
- 11.5 Un problema dell'allocazione contigua consiste nel fatto che l'utente deve preallocare abbastanza spazio per ogni file. Se il file diventa più grande dello spazio che gli è stato allocato, devono essere intraprese delle azioni specifiche. Questo problema può essere risolto definendo una struttura di file che consiste in un'area inizialmente contigua (di una dimensione specificata.) Se un'area viene colmata, il sistema operativo definisce automaticamente un'area di overflow collegata all'area inizialmente contigua. Se l'area di overflow viene riempita, si alloca una seconda area di overflow. Paragonate questa implementazione con le versioni standard contigue e concatenate.
- 11.6 Come possono le cache contribuire a migliorare le prestazioni? Perché i sistemi non utilizzano un maggior numero di cache, oppure cache più grandi, se esse sono così utili?
- 11.7 Perché è vantaggioso per l'utente che il sistema operativo allochi dinamicamente le proprie tabelle interne? Quali sono le conseguenze negative in cui incorrono i sistemi operativi che si comportano in questo modo?
- 11.8 Spiegate come lo strato VSF permetta al sistema operativo di supportare facilmente diversi tipi di file system.

## Esercizi

- 11.9 Considerate un file system che adopera un metodo di allocazione contigua modificato, comprensivo di estensioni. Un file contiene diverse estensioni, ognuna delle quali corrisponde a un insieme contiguo di blocchi. Un aspetto cruciale, per tali sistemi, è il grado di variabilità nella dimensione delle estensioni. Quali sono i vantaggi e gli svantaggi nel caso che:

- a. tutte le estensioni siano della stessa grandezza, che è predeterminata;
  - b. le estensioni possono essere di qualunque misura e sono allocate dinamicamente;
  - c. le estensioni variano tra poche misure fisse, che sono predeterminate.
- 11.10** Quali vantaggi offre la variante dell'allocazione concatenata che utilizza una FAT per collegare i blocchi di un file?
- 11.11** Considerate un sistema che tenga traccia dello spazio libero in una lista apposita.
- a. Supponete di perdere il puntatore alla lista. Il sistema è in grado di ricostruire la lista dello spazio libero? Argomentate la vostra risposta.
  - b. Considerate un file system simile a quello con allocazione indicizzata impiegato da UNIX. Quante operazioni di I/O del disco potrebbero essere necessarie per leggere i contenuti di un piccolo file locale posto in /a/b/c? Si assuma che nessuno dei blocchi del disco sia stato memorizzato nella cache.
  - c. Proponete una soluzione che impedisca la perdita del puntatore dovuta a un guasto della memoria.
- 11.12** In alcuni file system è possibile allocare lo spazio sul disco con diverse granularità. Un file system, ad esempio, può allocare 4 KB di spazio sul disco scegliendo un blocco unico da 4 KB oppure otto blocchi da 512 byte. In quale modo si può trarre vantaggio da questa flessibilità per migliorare le prestazioni? Quali modifiche si dovrebbero introdurre nel modello di gestione dello spazio libero per includervi questa caratteristica?
- 11.13** Discutete di come i tentativi di ottimizzazione delle prestazioni dei file system potrebbero generare difficoltà nella salvaguardia della coerenza dei sistemi, allorquando si verifichi un crash di sistema.
- 11.14** Considerate un file system in un disco con dimensioni dei blocchi logici e fisici di 512 byte. Supponete che le informazioni su ciascun file siano già in memoria. Per ciascuno dei tre metodi di allocazione (contigua, concatenata e indicizzata) fornite le risposte alle seguenti domande:
- a. dite come in questo sistema si fanno corrispondere gli indirizzi logici agli indirizzi fisici (per l'allocazione indicizzata supponete che la lunghezza di un file sia sempre inferiore a 512 blocchi);
  - b. se l'ultimo accesso è stato fatto al blocco 10 (posizione corrente), dite quanti blocchi fisici si devono leggere dal disco per accedere al blocco logico 4.
- 11.15** Considerate un file system che utilizza degli inode per rappresentare i file. I blocchi del disco hanno una dimensione di 8 KB e un puntatore a un blocco del disco richiede 4 byte. Questo file system ha 12 blocchi diretti sul disco, ma anche blocchi indiretti singoli, doppi e tripli. Qual è la dimensione massima di file che può essere memorizzata nel sistema?
- 11.16** In un dispositivo di memoria si può eliminare la frammentazione ricompattando le informazioni. I tipici dispositivi a disco non dispongono di registri di rilocazione o registri di base (come quelli usati per compattare la memoria); in questa situazione dite come si possono rilocare i file. Fornite tre motivi per i quali la ricompattazione e la rilocazione dei file vengono spesso evitate.
- 11.17** In quali situazioni l'uso della memoria come disco RAM sarebbe più utile rispetto al suo uso come cache del disco?

- 11.18 Considerate l'integrazione seguente a un protocollo per l'accesso remoto ai file. Ciascun client gestisce una cache per i nomi, in cui memorizza le traduzioni del nome di un file nel corrispondente handle del file. Quali questioni dovete tenere in considerazione nel realizzare la cache per i nomi?
- 11.19 Spiegate perché l'annotazione degli aggiornamenti ai metadati assicura il ripristino di un file system dopo un crollo del sistema.
- 11.20 Considerate il seguente schema di creazione di copie di riserva.
- ◆ **Giorno 1.** Copiatura nel supporto di backup delle copie di riserva di tutti i file contenuti nel disco.
  - ◆ **Giorno 2.** Copiatura in un altro supporto di tutti i file modificati dal giorno 1.
  - ◆ **Giorno 3.** Copiatura in un altro supporto di tutti i file modificati dal giorno 1.

Tale schema differisce dalla sequenza data nel Paragrafo 11.7.2 per il fatto che tutte le copiature riguardano tutti i file modificati dopo la prima copiatura completa. Dite quali sono i vantaggi di questo sistema rispetto a quello del Paragrafo 11.7.2 e se le operazioni di recupero sono più semplici o più difficili. Motivate le vostre risposte.

## 11.11 Note bibliografiche

Il sistema FAT dell'MS-DOS è spiegato in [Norton e Wilton 1988] e la descrizione del sistema OS/2 si trova in [Jacobucci 1988]. Questi sistemi operativi usano le famiglie di CPU Intel 8086 ([Intel 1985b], [Intel 1985a], [Intel 1986] e [Intel 1990]). I metodi di allocazione di IBM sono descritti in [Deitel 1990]. L'organizzazione interna del sistema BSD UNIX è ampiamente trattata in [McKusick et al. 1996]. [McVoy e Kleiman 1991] presentano ottimizzazioni di questi metodi realizzate per Solaris. Il file system di Google è descritto in [Ghemawat et al. 2003]. Si può trovare FUSE all'indirizzo <http://fuse.sourceforge.net/>.

L'allocazione dello spazio dei dischi per i file basata sul sistema buddy (gemellare) è trattata in [Koch 1987]. Uno schema di organizzazione dei file che garantisce il recupero dei dati con un solo accesso è trattato in [Larson e Kajla 1984]. I modelli di accesso ai file con registrazione per migliorare sia le prestazioni sia la coerenza sono stati presentati da [Rosenblum e Ousterhout 1991], [Seltzer et al. 1993], [Seltzer et al. 1995]. Gli algoritmi sugli alberi bilanciati sono discussi (insieme a molto altro materiale) in [Knuth 1998] e in [Cormen et al. 2001]. Il codice sorgente ZFS per le mappe di spazio è disponibile su [http://src.opensolaris.org/source/xref/onnv/onnvgate/usr/src/uts/common/fs/zfs/space\\_map.c](http://src.opensolaris.org/source/xref/onnv/onnvgate/usr/src/uts/common/fs/zfs/space_map.c).

L'uso delle memorie cache nella gestione dei dischi è trattato da [McKeon 1985] e [Smith 1985]; l'uso della cache nel sistema operativo sperimentale Sprite, è descritto in [Nelson et al. 1988]. Analisi generali sulle tecnologie delle memorie di massa si trovano in [Chi 1982] e [Hoagland 1985]. L'opera di [Folk e Zoellick 1987] concerne le varie strutture di file. [Silvers 2000] analizza la realizzazione della cache delle pagine nel sistema operativo FreeBSD.

[Sandberg et al. 1985], [Sandberg 1987], [Sun 1990] e [Callaghan 2000] trattano il file system di rete NFS. Le caratteristiche dei carichi di lavoro nei file system distribuiti sono studiate da [Baker et al. 1991]. Architetture basate sulla registrazione per file system di rete sono proposte in [Hartman e Ousterhout 1995], e da [Thekkath et al. 1997]. [Vahalia 1996] e [Mauro e McDougall 2001] descrivono l'NFS e il file system del sistema operativo UNIX, l'UFS. [Solomon 1998] descrive il file system NTFS del sistema operativo Windows NT. Il file system *Ext2* del sistema operativo Linux è stato trattato da [Bovet e Cesati 2002], mentre [Hitz et al. 1995] si è occupato del file system WAFL. Per approfondimenti riguardanti ZFS, si consulti <http://www.opensolaris.org/os/community/ZFS/docs>.

## Capitolo 12

# Memoria secondaria e terziaria



### OBIETTIVI

- Descrizione della struttura fisica dei dispositivi di memorizzazione secondaria e terziaria e degli effetti che derivano dal loro utilizzo.
- Spiegazione delle caratteristiche di prestazione della memoria secondaria e terziaria.
- Analisi dei servizi offerti dal sistema operativo per la memorizzazione secondaria e terziaria, compresi RAID e HSM.

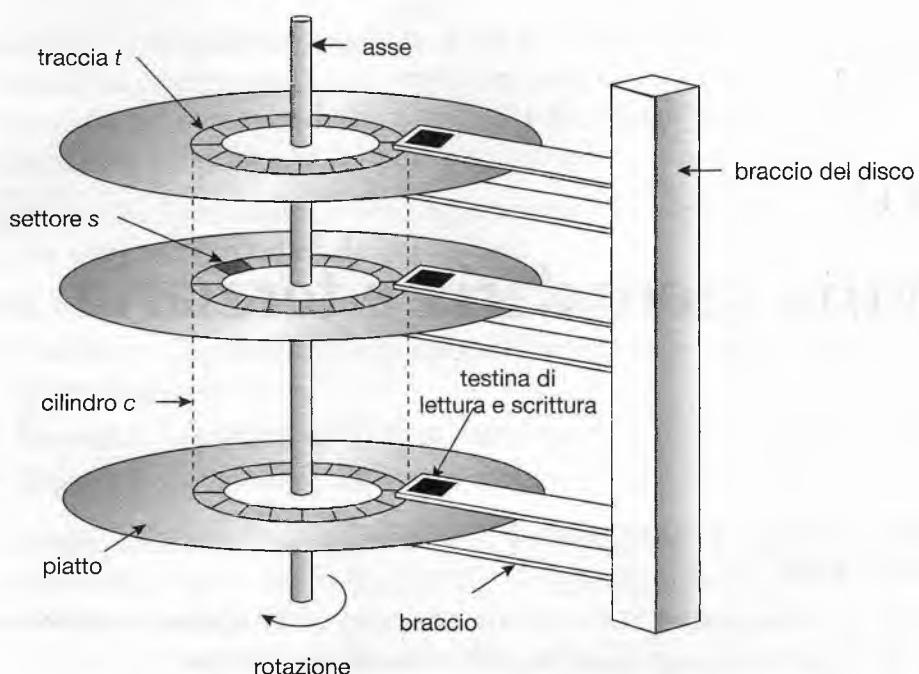
Il file system, da un punto di vista logico, si può considerare composto da tre parti: nel Capitolo 10 è stata presentata l'interfaccia per il programmatore e per l'utente del file system; nel Capitolo 11 sono state descritte le strutture dati interne e gli algoritmi usati dal sistema operativo per realizzare quest'interfaccia; nel presente capitolo si analizza il livello più basso del file system, e cioè la struttura dei supporti di memorizzazione secondaria e terziaria. Si descrivono innanzitutto gli algoritmi di scheduling delle unità a disco, che ordinano la sequenza delle operazioni di I/O al fine di migliorare le prestazioni del sistema. Quindi si illustrano la formattazione dei dischi e la gestione dei blocchi d'avviamento, dei blocchi danneggiati e dell'area d'avvicendamento dei processi. Si esamina la struttura della memoria secondaria, trattando l'affidabilità dei dischi e la realizzazione della memoria stabile. Il capitolo termina con una breve descrizione dei dispositivi di memoria terziaria e dei problemi che si presentano con il loro utilizzo da parte del sistema operativo.

## 12.1 Struttura dei dispositivi di memorizzazione

In questo paragrafo si introduce una presentazione generale della struttura fisica dei dispositivi di memorizzazione secondaria e terziaria.

### 12.1.1 Dischi magnetici

I dischi magnetici sono il mezzo fondamentale di memoria secondaria dei moderni sistemi di calcolo. Concettualmente, i dischi (Figura 12.1) sono relativamente semplici: i **piatti** dei dischi hanno una forma piana e rotonda come quella dei CD, con un diametro che comunemente varia tra 1,8 e 5,25 pollici, e le due superfici ricoperte di materiale magnetico; le informazioni si memorizzano registrandole magneticamente sui piatti.



**Figura 12.1** Schema funzionale di un disco.

Le testine di lettura e scrittura sono sospese su ciascuna superficie d'ogni piatto e sono attaccate al **braccio del disco** che le muove in blocco. La superficie di un piatto è divisa logicamente in tracce circolari a loro volta suddivise in settori; l'insieme delle tracce corrispondenti a una posizione del braccio (equidistanti dal centro dei piatti) costituisce un cilindro. In un'unità a disco possono esservi migliaia di cilindri concentrici e ogni traccia può contenere centinaia di settori. La capacità di memorizzazione di una comune unità a disco è dell'ordine delle decine di gigabyte.

Quando un disco è in funzione, un motore lo fa ruotare ad alta velocità; la maggior parte dei dischi ruota a velocità comprese tra 60 e 200 giri al secondo. L'efficienza di un disco è caratterizzata da due valori: la **velocità di trasferimento**, cioè la velocità con cui i dati fluiscono dall'unità a disco al calcolatore, e il **tempo di posizionamento**, talvolta detto tempo d'accesso diretto, che consiste nel tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato, detto **tempo di ricerca (seek time)**, e nel tempo necessario affinché il settore desiderato si porti, tramite la rotazione del disco, sotto la testina, detto **latenza di rotazione**. In genere i dischi possono trasferire parecchi megabyte di dati al secondo e hanno un tempo di ricerca e una latenza di rotazione di diversi millisecondi.

Poiché le testine di un disco sono sospese su un cuscinetto d'aria sottilissimo (dell'ordine dei micron), esiste il pericolo che la testina urti la superficie del disco; in tal caso, nonostante i piatti del disco siano ricoperti da un sottile strato protettivo, la testina può danneggiare la superficie magnetica. Tale incidente, detto **crollo della testina**, di solito non può essere riparato e comporta la sostituzione dell'intera unità a disco.

Un disco può essere **rimovibile**: ciò permette che diversi dischi siano montati secondo le necessità. I dischi magnetici rimovibili consistono generalmente in un piatto contenuto in un involucro di materiale plastico, che serve a evitare danni che possono verificarsi quando il disco non è inserito nella propria unità. I dischetti (*floppy disk*) sono dischi magnetici economici e rimovibili costituiti da un involucro di plastica che contiene un piatto flessibile. Poiché generalmente la testina delle unità a dischetti poggia direttamente sulla superficie

del disco, per ridurne l'usura, si fa ruotare il dischetto molto più lentamente dei piatti di un'unità a disco (*hard disk*). La capacità di memorizzazione di un dischetto è dell'ordine di circa 1,44 MB. Sono disponibili dischi rimovibili che funzionano in modo assai simile alle normali unità a disco e che hanno capacità dell'ordine dei gigabyte.

Un'unità a disco è connessa a un calcolatore attraverso un insieme di fili detto **bus** di I/O; esistono diversi tipi di tale bus, tra i quali i bus EIDE (*enhanced integrated drive electronics*), ATA (*advanced technology attachment*), ATA seriale (*serial ATA*), USB (*universal serial bus*), FC (*fiber channel*) e SCSI. Il trasferimento dei dati in un bus è eseguito da speciali unità di elaborazione dette **controllori**: gli **adattatori** (*adapter*) o **controllori di macchina** (*host controller*) sono i controllori posti all'estremità relativa al calcolatore del bus; i **controllori dei dischi** (*disk controller*) sono incorporati in ciascuna unità a disco. Per eseguire un'operazione di I/O il calcolatore inserisce un comando nell'adattatore, generalmente mediante porte di I/O mappate in memoria, com'è descritto nel Paragrafo 9.7.3; l'adattatore invia il comando al controllore del disco, che agisce sugli elementi elettromeccanici dell'unità a disco per portare a termine il comando. I controllori dei dischi di solito hanno una cache incorporata: il trasferimento dei dati nell'unità a disco avviene tra la cache e la superficie del disco; il trasferimento dei dati tra la cache e l'adattatore avviene alla velocità propria dei dispositivi elettronici.

### 12.1.2 Nastri magnetici

I **nastri magnetici** sono stati i primi supporti di memorizzazione secondaria. Pur avendo la capacità di memorizzare in modo permanente un'enorme quantità di dati, queste unità sono caratterizzate da un tempo d'accesso molto elevato rispetto a quello della memoria centrale e dei dischi magnetici. Inoltre il tempo d'accesso diretto dei nastri magnetici (essendo fisicamente ad accesso sequenziale) è migliaia di volte maggiore di quello dei dischi magnetici, e ciò li rende inadatti come supporto di memoria secondaria. Gli usi principali dei nastri sono la creazione di copie di riserva dei dati (*backup*), la registrazione di dati poco usati e il trasferimento di informazioni tra diversi sistemi di calcolo.

Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere alcuni minuti, anche se, una volta raggiunta la posizione desiderata, l'unità a nastro può leggere o scrivere informazioni a una velocità paragonabile a quella di un'unità a disco. La capacità varia secondo il particolare tipo di unità a nastro. I nastri hanno in genere una capacità compresa fra 20 e 200 GB. Alcune unità sono dotate di funzionalità di compressione dei dati che permettono di raddoppiare la capacità effettiva. Le unità a nastro e i loro driver sono solitamente classificate per larghezza: misure tipiche sono 4, 8 o 19 millimetri, e 1/4 o 1/2 pollice. Alcune unità prendono il nome dalla tecnologia su cui si fondano, come nel caso di LTO-2 e SDLT. Per altre informazioni sui nastri si veda il Paragrafo 12.9.

#### VELOCITÀ DI TRASFERIMENTO DEL DISCO

Come per molti altri aspetti dell'informatica, le prestazioni pubblicate relative ai dischi non coincidono con le cifre reali: sono sempre inferiori delle **velocità di trasferimento effettive**, per esempio. La velocità di trasferimento può essere considerata la velocità con cui la testina legge i bit dal supporto magnetico, ma questo aspetto va distinto dalla velocità con cui i blocchi sono consegnati al sistema operativo.

### FIREWIRE

**FireWire** è il nome di un'interfaccia progettata per collegare a un computer periferiche diverse, quali DVD, videocamere digitali e dischi rigidi. Si tratta di un prodotto originariamente sviluppato da Apple, e poi recepito come standard IEEE 1394 nel 1995. Nella sua prima versione, lo standard prevedeva una larghezza di banda massima di 440 megabit al secondo. Recentemente è apparso lo standard IEEE 1394b per FireWire2, che prevede fino a 800 megabit al secondo di larghezza di banda, circa il doppio dell'originale.

## 12.2 Struttura dei dischi

I moderni dischi sono considerati come grandi array monodimensionali di **blocchi logici**, dove un blocco logico è la minima unità di trasferimento. La dimensione di un blocco logico è di solito di 512 byte, sebbene alcuni dischi si possano **formattare a basso livello** allo scopo di ottenere una diversa dimensione dei blocchi logici, ad esempio 1024 byte; per altre informazioni su quest'opzione, si veda il Paragrafo 12.5.1. L'array monodimensionale di blocchi logici corrisponde in modo sequenziale ai settori del disco: il settore 0 è il primo settore della prima traccia sul cilindro più esterno; la corrispondenza prosegue ordinatamente lungo la prima traccia, quindi lungo le rimanenti tracce del primo cilindro, e così via di cilindro in cilindro dall'esterno verso l'interno.

Sfruttando questa corrispondenza sarebbe possibile – almeno in teoria – trasformare il numero di un blocco logico in un indirizzo fisico di vecchio tipo, consistente in un numero di cilindro, un numero di traccia concernente quel cilindro, e un numero di settore relativo a quella traccia. In pratica, però, vi sono due motivi che rendono difficile quest'operazione: in primo luogo, la maggior parte dei dischi contiene settori difettosi, ma la corrispondenza nasconde questo fatto sostituendo ai settori malfunzionanti settori sparsi in altre parti del disco; in secondo luogo, il numero di settori per traccia in certe unità a disco non è costante.

Nei supporti che impiegano la **velocità lineare costante** (*constant linear velocity*, CLV) la densità di bit per traccia è uniforme. Più è lontana dal centro del disco, tanto maggiore è la lunghezza della traccia, tanto maggiore è il numero di settori che essa può contenere. Spostandosi da aree esterne verso aree più interne il numero di settori per traccia diminuisce. Le tracce nell'area più esterna contengono in genere il 40 per cento in più dei settori contenuti nelle tracce dell'area più interna. L'unità aumenta la sua velocità di rotazione man mano che le testine si spostano dalle tracce esterne verso le tracce più interne per mantenere costante la quantità di dati che scorrono sotto le testine. Questo metodo si usa nelle unità per CD-ROM e DVD. In alternativa la velocità di rotazione dei dischi può rimanere costante, e la densità di bit decresce dalle tracce interne alle tracce più esterne per mantenere costante la quantità di dati che scorre sotto le testine. Questo metodo si usa nelle unità a disco magnetico ed è noto come **velocità angolare costante** (*constant angular velocity*, CAV).

Il numero di settori per traccia cresce in conseguenza alla tecnologia dei dischi, e l'area più esterna di un disco di solito contiene centinaia di settori per traccia. Anche il numero di cilindri è andato aumentando; le unità a disco contengono decine di migliaia di cilindri.

## 12.3 Connessione dei dischi

I calcolatori accedono alla memoria secondaria in due modi: nei sistemi di piccole dimensioni il modo più comune è tramite le porte di I/O (**memoria secondaria connessa alla macchina**); oppure ciò avviene in modo remoto per mezzo di un file system distribuito (**memoria secondaria connessa alla rete**).

### 12.3.1 Memoria secondaria connessa alla macchina

Alla memoria secondaria connessa alla macchina si accede dalle porte locali di I/O. Queste porte sono disponibili in diverse tecnologie; i comuni PC impiegano un'architettura per il bus di I/O chiamata IDE o ATA. Quest'architettura consente di avere non più di due unità per ciascun bus di I/O. Le stazioni di lavoro di fascia alta e i server impiegano architetture più raffinate come SCSI e FC (*fibre channel*).

L'architettura SCSI è un'architettura a bus il cui supporto fisico è di solito un cavo piatto con un gran numero di conduttori (in genere 50 o 68). Consente di avere sul bus fino a 16 dispositivi, comunemente suddivisi in una scheda con il controllore inserita nella macchina (*SCSI initiator*) e in 15 dispositivi di memorizzazione (*SCSI target*), in genere dischi SCSI. Il protocollo permette di accedere fino a 8 **unità logiche** per ciascun dispositivo di memorizzazione. Un uso tipico dell'accesso a unità logiche è l'invio di comandi ai componenti di una batteria RAID, o ai componenti di un archivio di supporti rimovibili (come un *jukebox* di CD che invia comandi al meccanismo per la sostituzione dei CD o a una delle unità).

L'FC è un'architettura seriale ad alta velocità che può funzionare sia su fibra ottica sia su un cavo con 4 conduttori di rame e ha due varianti. La prima è una grande struttura a commutazione con uno spazio d'indirizzi a 24 bit. Per il futuro ci si aspetta che questo metodo prevalga, ed è la base per le **reti di memoria secondaria** (*storage-area networks*, SAN), trattate nel Paragrafo 12.3.3. Grazie al vasto spazio d'indirizzi e alla natura a commutazione della comunicazione, si possono connettere più macchine e dispositivi di memorizzazione alla struttura a commutazione, permettendo una notevole flessibilità nella comunicazione di I/O. La seconda variante si chiama FC-AL (*arbitrated loop*) e può accedere fino a 126 dispositivi (unità e controllori).

C'è un gran numero di dispositivi utilizzabili come memoria secondaria connessa alla macchina, tra questi le unità a disco, le batterie RAID, le unità a CD, DVD e a nastri magnetici. I comandi di I/O che avviano trasferimenti di dati a un dispositivo di memoria connessa alla macchina sono letture e scritture di blocchi logici di dati, dirette a unità di memorizzazione specificamente identificate (ad esempio, tramite bus ID, SCSI ID e unità logica del dispositivo).

### 12.3.2 Memoria secondaria connessa alla rete

Un dispositivo di memoria secondaria connessa alla rete è un sistema di memoria speciale al quale si accede in modo remoto per mezzo di una rete di trasmissione di dati (Figura 12.2). I client accedono alla memoria connessa alla rete (*network-attached storage*, NAS) tramite un'interfaccia RPC, come l'NFS nel caso dei sistemi UNIX o come CIFS nel caso di sistemi Windows. Le chiamate di procedura remota (RPC) sono realizzate per mezzo dei protocolli TCP o UDP sopra una rete IP (di solito la stessa rete locale che porta tutto il traffico di dati ai client). La memoria secondaria connessa alla rete è normalmente realizzata come una batteria RAID con programmi di controllo che implementano l'interfaccia per le RPC. Conviene



**Figura 12.2** Memoria secondaria connessa alla rete.

pensare ai sistemi NAS semplicemente come a un altro protocollo per l'accesso alla memoria secondaria; ad esempio, anziché usare un driver per dispositivi SCSI e i relativi protocoli SCSI per accedere alla memoria secondaria, un sistema che usa sistemi NAS impiega le RPC sopra i protocolli TCP/IP.

La memoria secondaria connessa alla rete fornisce un modo semplice per condividere spazio di memorizzazione a tutti i calcolatori di una LAN, con la stessa facilità di gestione dei nomi e degli accessi caratteristica della memoria secondaria locale. Tuttavia, un sistema di questo genere tende a essere meno efficiente e ad avere prestazioni inferiori rispetto a sistemi che prevedono la connessione diretta alla memoria secondaria.

ISCSI è il più recente protocollo per la memoria connessa alla rete. Essenzialmente sfrutta il protocollo IP della rete per il trasporto del protocollo SCSI. Ne consegue la possibilità di usare cavi di rete invece che cavi SCSI per connettere le diverse macchine alla memoria secondaria. Uno dei vantaggi di questa tecnica è che le macchine sono in grado di trattare la memoria secondaria come se fosse direttamente collegata, sebbene possa essere collocata a distanza.

### 12.3.3 Reti di memoria secondaria

Uno svantaggio dei sistemi di memoria secondaria connessa alla rete è che le operazioni di I/O sulla memoria secondaria impegnano banda della rete e quindi aumentano la latenza della comunicazione nella rete. Questo problema può essere particolarmente grave per sistemi client-server di grandi dimensioni: l'ordinaria comunicazione tra i server e i client compete per la banda con la comunicazione tra i server e i dispositivi di memorizzazione.

Una rete di memoria secondaria (*storage-area network*, SAN) è una rete privata (che impiega protocolli specifici per la memorizzazione anziché protocolli di rete) tra i server e le unità di memoria secondaria, separata dalla LAN o WAN che collega i server ai client (Figura 12.3). La potenza di una SAN sta nella sua flessibilità: si possono connettere alla stessa SAN molte macchine e molte batterie di memoria; la memoria può essere allocata alle macchine dinamicamente. Uno switch SAN nega o concede alle macchine l'accesso alla memoria secondaria. Per fare un esempio, è possibile configurarlo di modo che allochi memoria secondaria aggiuntiva alle macchine che stanno esaurendo lo spazio sui propri dischi. Questi switch rendono anche possibile la condivisione della memoria di massa da parte di gruppi di server, e il collegamento diretto di più macchine alla memoria di massa. I SAN, infatti, hanno spesso un maggior numero di porte dei dispositivi per la memorizzazione secondaria, a un prezzo più contenuto. Il mezzo più comune per il collegamento tramite SAN è il FC.

Un'alternativa emergente è un'architettura specifica per SAN, chiamata InfiniBand, per reti d'interconnessione ad alta velocità tra server e unità di memoria secondaria.

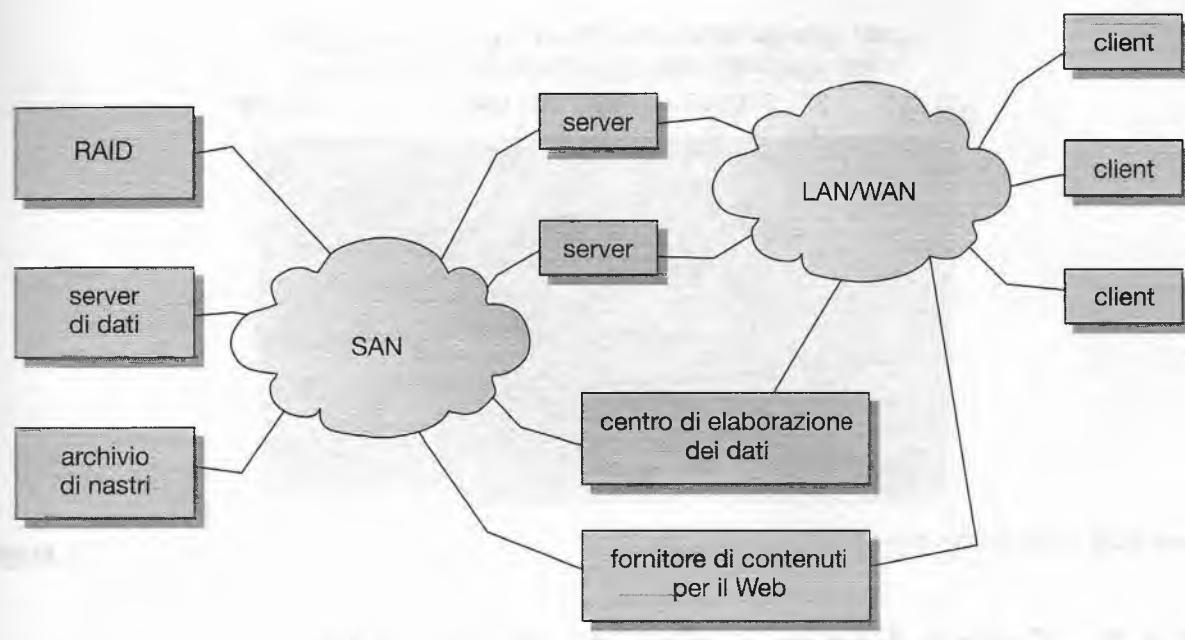


Figura 12.3 Rete di memoria secondaria.

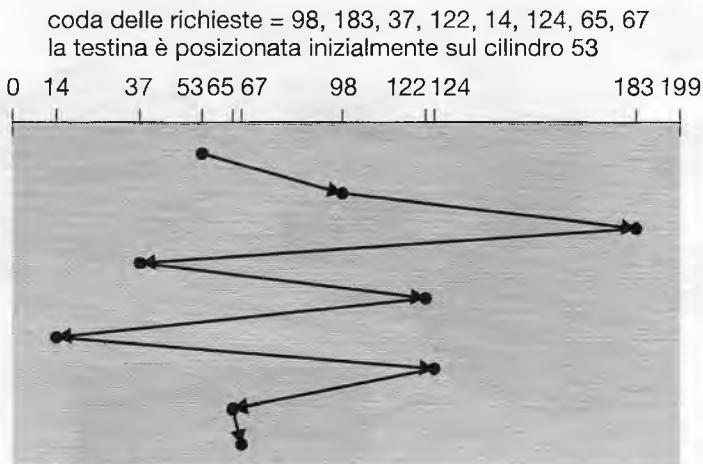
## 12.4 Scheduling del disco

Una delle responsabilità del sistema operativo è quella di fare un uso efficiente delle risorse fisiche: nel caso delle unità a disco, far fronte a questa responsabilità significa garantire tempi d'accesso contenuti e ampiezze di banda elevate. Il tempo d'accesso si può scindere in due componenti principali (si veda anche il Paragrafo 12.1.1): il **tempo di ricerca** (*seek time*), cioè il tempo necessario affinché il braccio dell'unità a disco sposti le testine fino al cilindro contenente il settore desiderato, e la **latenza di rotazione** (*rotational latency*), e cioè il tempo aggiuntivo necessario perché il disco ruoti finché il settore desiderato si trovi sotto la testina. L'**ampiezza di banda** (*bandwidth*) è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell'ultimo trasferimento. Per mezzo della gestione dell'ordine delle richieste di I/O relative al disco si possono migliorare sia il tempo d'accesso sia l'ampiezza di banda.

Ogni volta che devono compiere operazioni di I/O con un'unità a disco, un processo impedisce al sistema operativo una chiamata di sistema. La richiesta contiene diverse informazioni:

- ◆ se l'operazione sia di immissione o di emissione di dati;
- ◆ l'indirizzo nel disco rispetto al quale eseguire il trasferimento;
- ◆ l'indirizzo di memoria rispetto al quale eseguire il trasferimento;
- ◆ il numero di byte da trasferire.

Se l'unità a disco desiderata e il controllore sono disponibili, la richiesta si può immediatamente soddisfare; altrimenti le nuove richieste si aggiungono alla coda di richieste inevase relativa a quell'unità. La coda relativa a un'unità a disco in un sistema con multiprogrammazione può spesso essere piuttosto lunga, sicché il sistema operativo sceglie quale fra le richieste inevase conviene servire prima. Come il sistema operativo affronta tale scelta? Per mezzo di uno dei tanti algoritmi di scheduling che andiamo a presentare di seguito.



**Figura 12.4** Scheduling FCFS.

### 12.4.1 Scheduling in ordine d'arrivo – FCFS

La forma più semplice di scheduling è, naturalmente, l'algoritmo di servizio secondo l'ordine d'arrivo (*first come, first served*, FCFS). Si tratta di un algoritmo intrinsecamente equo, ma che in generale non garantisce la massima velocità del servizio. Si consideri, ad esempio, una coda di richieste per l'unità a disco che dia una lista di cilindri sui quali individuare i blocchi richiesti, nell'ordine seguente:

98, 183, 37, 122, 14, 124, 65, 67.

Se si trova inizialmente al cilindro 53, la testina dell'unità a disco dovrà prima spostarsi al cilindro 98, poi al 183, 37, 122, 14, 124, 65 e infine al 67, per una distanza totale, misurata in numero di cilindri visitati, di 640 cilindri. La sequenza è rappresentata nella Figura 12.4.

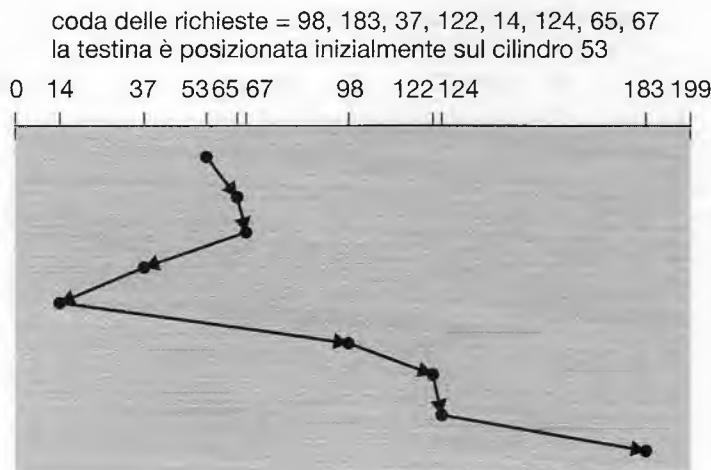
Le defezioni di quest'algoritmo sono palesate dal gigantesco salto da 122 a 14 e poi di nuovo a 124: se le richieste per i cilindri 37 e 14 si potessero soddisfare in sequenza, la distanza totale percorsa diminuirebbe notevolmente e le prestazioni migliorerebbero di conseguenza.

### 12.4.2 Scheduling per brevità – SSTF

Sembra ragionevole servire tutte le richieste vicine alla posizione corrente della testina prima di spostarla in un'area lontana per soddisfarne altre: questa considerazione è alla base dell'**algoritmo di servizio secondo il più breve tempo di ricerca** (*shortest seek time first*, SSTF), che sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina; poiché questo tempo aumenta al crescere della distanza dei cilindri dalla testina, l'algoritmo sceglie le richieste relative ai cilindri più vicini alla posizione della testina.

Se si considera nuovamente la sequenza di richieste presa ad esempio sopra, il cilindro più vicino alla posizione iniziale della testina (cioè 53) è il 65, la successiva richiesta più vicina è quella relativa al cilindro 67; da questo cilindro, la richiesta relativa al cilindro 37 è più vicina di quella relativa al cilindro 98, ed è quindi servita per terza. Continuando allo stesso modo, sarà soddisfatta la richiesta relativa al cilindro 14, poi 98, 122, 124 e infine 183 (Figura 12.5). Questo metodo di scheduling implica una distanza totale percorsa di soli 236 cilindri, poco più di un terzo della distanza ottenuta con lo scheduling FCFS di questa coda di richieste: esso porta quindi sostanziali miglioramenti d'efficienza.

Lo scheduling SSTF è essenzialmente una forma di scheduling per brevità (*shortest job first*, SJF), e al pari di questo, può condurre a situazioni d'attesa indefinita (*starvation*) di al-



**Figura 12.5** Scheduling SSTF.

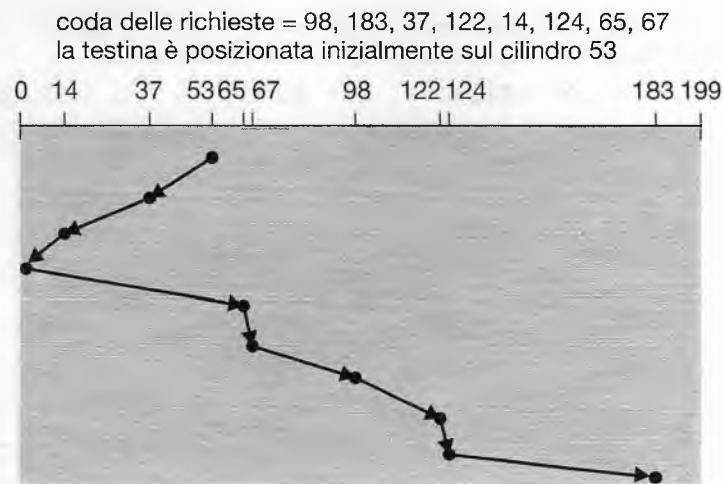
cune richieste. Si ricordi infatti che nuove richieste possono giungere in qualunque momento: si supponga di avere due richieste in coda, una per il cilindro 14 e l'altra per il 186, e che mentre si sta servendo la richiesta relativa al cilindro 14, arrivi una nuova richiesta per un cilindro vicino al 14. Questa sarà la prossima richiesta soddisfatta, mentre la richiesta per il cilindro 186 dovrà attendere. La stessa situazione potrebbe ripetersi, perché un'altra richiesta relativa a una posizione in prossimità del cilindro 14 potrà giungere mentre si sta servendo la precedente richiesta: in teoria, un flusso continuo di richieste riferite a posizioni le une vicine alle altre potrebbe causare l'attesa indefinita della richiesta relativa al cilindro 186. Quest'ipotesi diviene sempre più probabile man mano che la coda di richieste si allunga.

Sebbene l'algoritmo SSTF costituisca un miglioramento notevole rispetto al FCFS, esso non è ottimale: si può fare di meglio con uno spostamento dal cilindro 53 al 37, anche se questa non è la minima distanza possibile, e poi al 14, prima di invertire la marcia per servire i 65, 67, 98, 122, 124 e 183. L'adozione di questa strategia riduce la distanza totale percorsa a 208 cilindri.

### 12.4.3 Scheduling per scansione – SCAN

Secondo l'**algoritmo SCAN** il braccio dell'unità a disco parte da un estremo del disco e si sposta nella sola direzione possibile, servendo le richieste mentre attraversa i cilindri, finché non giunga all'altro estremo del disco: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni. L'algoritmo SCAN è a volte chiamato **algoritmo dell'ascensore**, perché il braccio dell'unità a disco si comporta proprio come un ascensore che serva prima tutte le richieste in salita e poi tutte quelle in discesa.

Si consideri ancora l'esempio precedente, prima di poter applicare lo scheduling SCAN alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, e 67, oltre la posizione corrente (53), occorre conoscere la direzione del movimento delle testine. Se lo spostamento è nella direzione del cilindro 0, l'unità a disco servirà prima la richiesta 37 e poi la 14; una volta giunto al cilindro 0, il braccio invertirà il movimento verso l'altro estremo del disco, servendo le richieste 65, 67, 98, 122, 124 e 183 (Figura 12.6). Se arriva una nuova richiesta riferita a uno dei cilindri posti davanti alla testina (relativamente alla sua direzione di moto), essa sarà quasi immediatamente soddisfatta; ma se la richiesta è riferita a uno dei cilindri appena sorpassati, essa dovrà attendere fino a che la testina non giunga alla fine del disco, inverta la direzione del moto, e torni indietro.

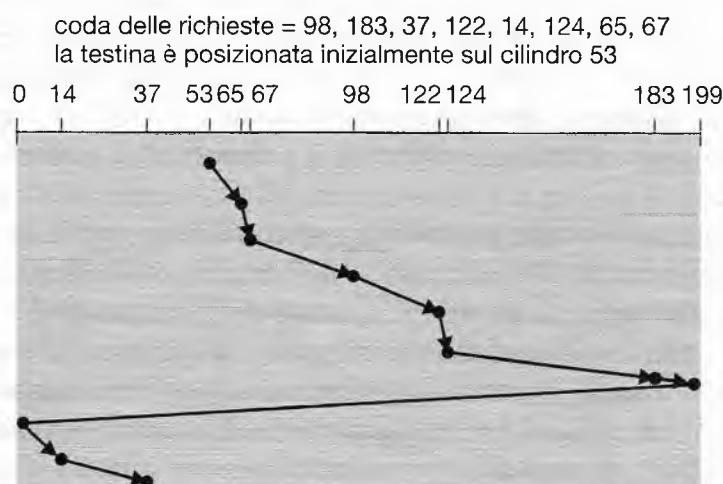


**Figura 12.6** Scheduling SCAN.

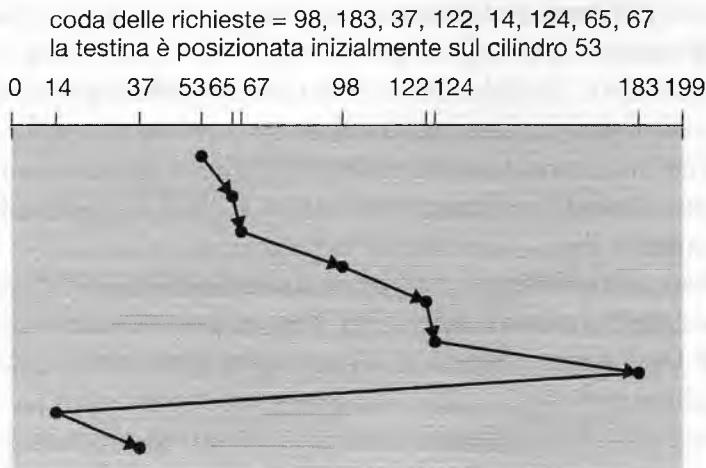
Assumendo una distribuzione uniforme delle richieste per i cilindri da visitare, si consideri la densità di richieste quando il braccio giunge a un estremo e inverte la direzione del moto: in quel momento, relativamente poche richieste sono riferite a cilindri posti vicino alla testina e davanti a essa, perché i cilindri in questione sono stati recentemente visitati. La massima densità di richieste si riferisce all'altro estremo del disco, e queste richieste sono anche quelle che hanno atteso più a lungo: sembra ragionevole spostarsi lì come prossima mossa. Questa è l'idea dell'algoritmo seguente.

#### 12.4.4 Scheduling per scansione circolare – C-SCAN

L'algoritmo SCAN circolare (*circular SCAN*, C-SCAN) è una variante dello scheduling SCAN concepita per garantire un tempo d'attesa meno variabile. Anche l'algoritmo C-SCAN, come lo SCAN, sposta la testina da un estremo all'altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all'altro estremo del disco, ritorna immediatamente all'inizio del disco stesso, senza servire richieste durante il viaggio di ritorno (Figura 12.7). L'algoritmo di scheduling C-SCAN, essenzialmente, tratta il disco come una lista circolare, cioè come se il primo e l'ultimo cilindro fossero adiacenti.



**Figura 12.7** Scheduling C-SCAN.



**Figura 12.8** Scheduling C-LOOK.

### 12.4.5 Scheduling LOOK

Secondo la descrizione appena fatta, sia l'algoritmo SCAN sia il C-SCAN spostano il braccio dell'unità attraverso tutta l'ampiezza del disco; all'atto pratico, nessuno dei due algoritmi è codificato in questo modo: più comunemente, il braccio si sposta solo finché ci sono altre richieste da servire in quella direzione, dopo di che cambia immediatamente direzione, senza giungere all'estremo del disco. Queste versioni dello SCAN e del C-SCAN sono dette LOOK e C-LOOK, perché “guardano” (in inglese, *look*) se ci sono altre richieste da soddisfare lungo la direzione attuale prima di continuare a spostare la testina in quella direzione (Figura 12.8).

### 12.4.6 Scelta di un algoritmo di scheduling

Giacché esistono tanti diversi algoritmi di scheduling, ci si può chiedere come si faccia a scegliere il migliore. Un algoritmo molto comune e naturalmente attraente è l'SSTF poiché aumenta le prestazioni rispetto all'FCFS; lo SCAN e il C-SCAN danno migliori prestazioni in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità a situazioni d'attesa infinita. Per una data ma arbitraria lista di richieste si può definire un ordine ottimo di servizio, ma la computazione richiesta può non essere giustificata dal miglioramento in prestazioni rispetto agli algoritmi SSTF o SCAN.

Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste. Ad esempio, si supponga che la coda sia costituita in genere di una sola richiesta inevasa: tutti gli algoritmi danno allora luogo allo stesso comportamento, perché hanno una sola scelta possibile relativamente al prossimo spostamento della testina. In questo caso, tutti gli algoritmi si comportano come l'FCFS.

Le richieste di I/O per l'unità a disco possono essere notevolmente influenzate dal metodo adottato per l'assegnazione dei file. Un programma che legga un file allocato in modo contiguo genererà molte richieste raggruppate, con un conseguente limitato spostamento della testina. Un file con allocatione concatenata o indicizzata, d'altro canto, potrebbe includere blocchi sparsi per tutto il disco, e richiedere quindi un maggiore movimento della testina.

Anche la posizione delle directory e dei blocchi indice è importante: poiché ogni file deve essere aperto per essere usato, e visto che l'apertura di un file richiede una ricerca attraverso la struttura delle directory, vi saranno frequenti accessi alle directory. Si supponga che

un elemento di directory risieda nel primo cilindro e che i dati del file relativo si trovino nell'ultimo cilindro; la testina dovrà allora percorrere l'intera ampiezza del disco nel caso di apertura del file in questione. Se l'elemento della directory che rappresenta logicamente il file fosse nel cilindro di mezzo, la testina dovrebbe spostarsi al più della metà dell'ampiezza. Anche l'uso della memoria centrale come cache delle directory e dei blocchi indice può contribuire a ridurre i movimenti del braccio dell'unità a disco, in particolare quando si tratta di operazioni di lettura.

A causa di queste complicazioni, l'algoritmo di scheduling del disco dovrebbe costituire un modulo a sé stante del sistema operativo, così da poter essere sostituito da un altro algoritmo qualora ciò fosse necessario; come algoritmo di partenza è ragionevole la scelta dell'SSTF o del LOOK.

Gli algoritmi di scheduling descritti tengono conto solamente del tempo di ricerca, mentre nelle moderne unità a disco la latenza di rotazione può essere lunga quasi quanto il tempo medio di ricerca. Tuttavia è difficile per il sistema operativo adottare una strategia di scheduling che porti a miglioramenti dei tempi di latenza di rotazione, perché le moderne unità a disco non rivelano la posizione fisica dei blocchi logici. I produttori di unità a disco hanno collaborato alla limitazione di questo problema incorporando algoritmi di scheduling all'interno dei controllori contenuti nelle unità a disco: se il sistema operativo invia un gruppo di richieste al controllore, esso può organizzarle in una coda e poi applicare algoritmi di scheduling che riducano sia i tempi di ricerca sia la latenza di rotazione.

Se l'unico elemento di cui tener conto fossero le prestazioni dell'I/O, il sistema operativo scaricherebbe volentieri la responsabilità dello scheduling per il disco sull'apparato elettronico del dispositivo. In pratica, però, il sistema operativo può dover considerare altri vincoli relativi all'ordine in cui si devono servire le richieste: ad esempio, la richiesta di una pagina di memoria virtuale potrebbe avere maggiore priorità rispetto all'I/O delle applicazioni, e le scritture divengono più urgenti delle letture quando la cache sta per esaurire le pagine disponibili. Inoltre, può essere auspicabile mantenere l'ordine naturale delle richieste di scrittura al fine di rendere il file system robusto rispetto ai crolli del sistema: si consideri cosa accadrebbe se il sistema operativoassegnasse un blocco di disco a un file, e un'applicazione scrivesse dati in quel blocco; se il sistema crollasse a questo punto, il sistema operativo potrebbe non essere riuscito a copiare l'*inode* modificato e la nuova lista dello spazio libero sul disco. Per conciliare queste esigenze, il sistema operativo può scegliere di accollarsi la responsabilità dello scheduling del disco e, per alcuni tipi di I/O, fornire le richieste al controllore una alla volta.

## 12.5 Gestione dell'unità a disco

Il sistema operativo è anche responsabile di molti altri aspetti della gestione delle unità a disco. In questo paragrafo si discutono l'inizializzazione del disco, l'avviamento del sistema basato sull'unità a disco, e la gestione dei blocchi difettosi.

### 12.5.1 Formattazione del disco

Un disco magnetico nuovo è *tabula rasa*: un insieme di uno o più piatti sovrapposti ricoperti di materiale magnetico; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Questo processo si chiama formattazione di basso livello, o **formattazione fisica**. La **formattazione di basso livello** riempie il disco con una speciale struttura dati per ogni settore, tipicamente consistente di un'intestazione, un'area per i dati (di solito di 512 byte), e una coda. L'intestazione e la coda contengono in-

formazioni usate dal controllore del disco, ad esempio il numero del settore e un codice per la correzione degli errori (*error-correcting code*, ECC). Quando il controllore scrive dati in un settore nel corso di un'ordinaria operazione di I/O, aggiorna il valore dell'ECC secondo il contenuto dell'area di dati del settore. Quando il controllore legge dati da quel settore, calcola anche l'ECC e lo confronta con il suo valore memorizzato: se risulta una discrepanza, l'area dei dati del settore non è integra, e il settore del disco potrebbe essere difettoso (si veda il Paragrafo 12.5.3). L'ECC è un codice per la *correzione* degli errori: se solo alcuni bit di dati sono stati alterati, esso contiene sufficienti informazioni affinché il controllore possa identificare i bit in questione e ricalcolare il loro corretto valore. Il controllore esegue automaticamente l'elaborazione descritta ogni volta che accede a un settore del disco.

La formattazione fisica dei dischi è eseguita nella maggior parte dei casi dal costruttore come parte del processo produttivo; ciò permette al costruttore di provare il disco, e di instaurare la corrispondenza fra blocchi logici e settori correttamente funzionanti del disco. In molte unità a disco, quando si richiede al controllore di formattare fisicamente il disco, si può anche specificare il numero di byte delle aree di dati comprese fra l'intestazione e la coda di un settore. La scelta è di solito ristretta a poche opzioni, come 256, 512 o 1024 byte. La formattazione in settori più grandi implica la presenza di meno settori su ogni traccia, ma anche meno intestazioni e code, e quindi maggior spazio per i dati veri e propri. Alcuni sistemi operativi gestiscono solo settori di 512 byte.

Per usare un disco come contenitore d'informazioni, il sistema operativo deve ancora registrare le proprie strutture dati nel disco, cosa che fa in due passi. Il primo consiste nel suddividere il disco in uno o più gruppi di cilindri, detti **partizioni**. Il sistema operativo può trattare ogni gruppo come se fosse un'unità a disco a sé stante: ad esempio, una partizione può contenere una copia del codice eseguibile del sistema operativo, mentre un'altra contiene i file degli utenti. Il passo successivo alla suddivisione in partizioni è la **formattazione logica**, cioè la creazione di un file system: il sistema operativo registra nel disco le strutture dati iniziali relative al file system. Le strutture dati in questione possono includere descrizioni dello spazio libero e dello spazio allocato (FAT o *inode*) e una directory iniziale vuota.

Per una maggior efficienza, molti file system accorpano i blocchi in gruppi, detti **cluster**. L'I/O del disco procede per blocchi, ma l'I/O del file system procede invece per cluster, di modo che l'I/O mutui sempre più caratteristiche dall'accesso sequenziale che non dall'accesso casuale.

Alcuni sistemi operativi danno l'opportunità a certi programmi speciali di impiegare una partizione del disco come un grande array sequenziale di blocchi logici, non contenente alcuna struttura dati relativa al file system, detto *disco di basso livello* (*raw disk*); l'I/O relativo si chiama I/O di basso livello (*raw I/O*). Alcuni sistemi per la gestione di basi di dati, ad esempio, preferiscono questo tipo di I/O perché permette di controllare l'esatta posizione nel disco d'ogni informazione trattata. Esso scavalca tutti i servizi del file system: gestione delle *buffer cache*, prelievo anticipato, allocazione dello spazio, nomi dei file, directory, e così via. È possibile rendere certe applicazioni più efficienti includendovi servizi di memorizzazione secondaria specializzati che usino una partizione di basso livello, ma la maggior parte delle applicazioni è in realtà migliore quando usufruisce degli ordinari servizi del file system.

## 12.5.2 Blocco d'avviamento

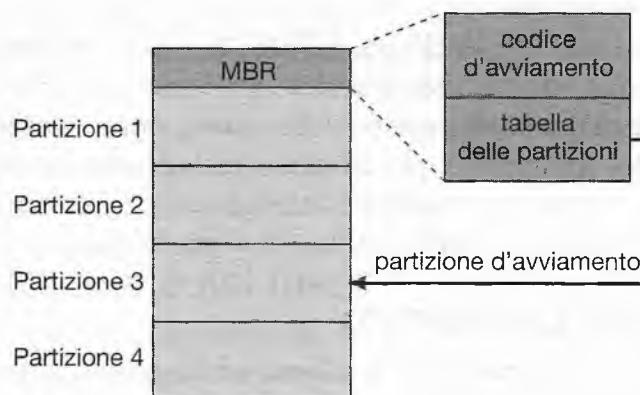
Affinché un calcolatore possa entrare in funzione, ad esempio quando viene acceso o riavviato, è necessario che esegua un programma iniziale; di solito, questo programma d'avviamento iniziale è piuttosto semplice. Esso inizializza il sistema in tutti i suoi aspetti, dai registri della CPU ai controllori dei dispositivi e al contenuto della memoria centrale, quindi avvia il sistema operativo. Per far ciò, il programma d'avviamento trova il kernel del sistema

operativo nei dischi, lo carica nella memoria, e salta a un indirizzo iniziale per avviare l'esecuzione del sistema operativo.

Per la maggior parte dei calcolatori, il programma d'avviamento è memorizzato in una **memoria a sola lettura** (*read-only memory*, ROM), il che è conveniente, perché la ROM non richiede inizializzazione, e ha un indirizzo iniziale fisso dal quale la CPU può cominciare l'esecuzione ogniqualvolta si accende o si riavvia la macchina. Inoltre, visto che la ROM è a sola lettura, non può essere contaminata da un virus informatico. Il problema, però, è che cambiare il programma d'avviamento richiede in questo caso la sostituzione dei circuiti integrati ROM. A causa di questo inconveniente, molti sistemi memorizzano nella ROM un piccolo caricatore d'avviamento (*bootstrap loader*) il cui solo compito è quello di caricare da un disco il programma d'avviamento completo. Quest'ultimo si può facilmente modificare: se ne scrive semplicemente una nuova versione nel disco. Il programma d'avviamento completo è registrato in una partizione del disco denominata blocchi d'avviamento, posta in una locazione fissata del disco; un disco contenente una tale partizione si chiama **disco d'avviamento** o **disco di sistema**.

Il codice contenuto nella ROM d'avviamento istruisce innanzi tutto il controllore dell'unità a disco affinché trasferisca il contenuto dei blocchi d'avviamento nella memoria (si noti che a questo fine non si carica alcun driver di dispositivo), quindi comincia a eseguire il codice. Il programma d'avviamento completo è più complesso del suo caricatore, ed è capace di trasferire nella memoria l'intero sistema operativo inizialmente residente in un disco in una locazione non definitivamente fissata, e di avviare il sistema operativo stesso. Il programma d'avviamento potrà comunque essere breve.

Consideriamo come esempio il processo d'avviamento in Windows 2000. Questo sistema colloca il proprio codice d'avviamento nel primo settore del disco rigido, denominato **MBR** (*master boot record*). Esso, inoltre, consente di suddividere il disco rigido in una o più partizioni; in una di loro, detta **partizione d'avviamento**, sono contenuti il sistema operativo e i driver dei dispositivi. La procedura d'avviamento di Windows 2000 inizia con l'esecuzione del codice residente nella memoria ROM del sistema. Questa parte del codice guida il sistema a leggere il codice d'avviamento dall'MBR. Oltre al codice d'avviamento, l'MBR contiene una tabella che elenca le partizioni del disco rigido e un flag che indica da quale partizione si debba avviare il sistema; ciò è illustrato nella Figura 12.9. Dopo aver identificato la partizione d'avviamento, il sistema legge da tale partizione il primo settore (chiamato **settore d'avviamento**) e svolge le restanti procedure d'avviamento, tra cui il caricamento dei vari sottosistemi e dei servizi del sistema.



**Figura 12.9** Avviamento dal disco di Windows 2000.

### 12.5.3 Blocchi difettosi

Le unità a disco sono strutturalmente portate ai malfunzionamenti perché sono costituite da parti mobili a bassa tolleranza (si ricordi che una testina è sospesa appena sopra la superficie del disco). A volte si può verificare un guasto irreparabile, e l'unità a disco deve essere sostituita: le informazioni contenute nel disco dovranno essere recuperate da una copia di riserva mantenuta separatamente e trasferite nella nuova unità a disco. Più di frequente, uno o più settori divengono malfunzionanti; in effetti, la maggior parte dei dischi messi in commercio contiene già **blocchi difettosi**. Essi sono trattati in diversi modi secondo il controllore e l'unità a disco presenti nel sistema.

Nel caso di dischi semplici come quelli gestiti da un controllore IDE, i blocchi difettosi sono gestiti "manualmente". Ad esempio, il comando `format` dell'MS-DOS esegue una formattazione logica, e come parte del processo esamina il disco per rilevare la presenza di blocchi difettosi: se ne trova qualcuno, scrive un valore speciale nel corrispondente elemento della FAT al fine di segnalare alle procedure di allocazione di non usare il blocco in questione. Se qualche blocco diviene malfunzionante nel corso dell'ordinario uso del sistema, un programma speciale (ad esempio `chkdsk`) deve essere eseguito a cura dell'utente per individuare i blocchi difettosi e isolarli come appena descritto. Di solito, i dati residenti nei blocchi difettosi vanno perduti.

Unità a disco più complesse come i dischi SCSI in uso nei PC di alto livello e nella maggior parte delle stazioni di lavoro e server, hanno strategie di recupero dei blocchi difettosi più raffinate. Il controllore mantiene una lista dei blocchi malfunzionanti dell'unità a disco che è inizializzata durante la formattazione fisica eseguita dal produttore, ed è aggiornata per tutto il periodo in cui l'unità a disco è operativa. La formattazione fisica mette anche da parte dei settori di riserva non visibili al sistema operativo: si può istruire il controllore affinché sostituiscia da un punto di vista logico un settore difettoso con uno dei settori di riserva inutilizzati. Questa strategia è nota come **accantonamento di settori** (*sector sparing* o *sector forwarding*).

Un tipico esempio di attuazione di questa strategia è il seguente:

- ◆ il sistema operativo tenta di leggere il blocco logico 87;
- ◆ il controllore calcola l'ECC e scopre che il settore è difettoso; quindi segnala questo malfunzionamento al sistema operativo;
- ◆ la volta successiva che il sistema viene riavviato, si esegue un comando speciale al fine di comunicare al controllore SCSI la necessità di sostituire il settore difettoso con uno di riserva;
- ◆ dopo di ciò, ogni volta che il sistema tenta di leggere il contenuto del blocco 87, il controllore traduce la richiesta nell'indirizzo del settore di rimpiazzo.

Un tale reindirizzamento da parte del controllore potrebbe inficiare ogni ottimizzazione fornita dall'algoritmo di scheduling del disco del sistema operativo. Per questa ragione la maggior parte dei dischi si formatta in modo tale da mantenere qualche settore di riserva in ogni cilindro, e anche un intero cilindro di riserva. Quando un numero di blocco logico è allocato a dei settori di riserva, il controllore usa settori di riserva presenti nello stesso cilindroogniqualvolta ciò sia possibile.

Un'alternativa all'accantonamento dei settori è data da quei controllori capaci di sostituire i settori difettosi tramite la tecnica della **traslazione dei settori** (*sector slipping*). Si supponga ad esempio che il blocco logico 17 divenga malfunzionante, e che il primo settore di riserva disponibile sia quello successivo al settore 202. La traslazione dei settori sposterebbe in avanti di un posto tutti i settori dal 17 al 202: quindi, il settore 202 sarebbe copiato sul

settore di riserva, il settore 201 sul 202, il 200 sul 201, e così via, fino a che il settore 18 non sia stato copiato sul 19. Questa traslazione dei settori libera lo spazio del settore 18, e il settore 17 può essere fatto corrispondere a quest'ultimo.

La sostituzione di un blocco difettoso non è in genere un processo totalmente automatico, perché i dati contenuti nel blocco in questione di solito vanno perduti. Un file che usava quel blocco deve quindi essere riparato (ad esempio, ricopiandolo da un nastro contenente le copie di riserva), e ciò richiede un intervento manuale.

## 12.6 Gestione dell'area d'avvicendamento

L'avvicendamento (*swapping*) è stato introdotto, inizialmente, nel Paragrafo 8.2, dove abbiamo trattato lo spostamento di interi processi tra disco e memoria centrale. In quel contesto, l'avvicendamento interviene quando l'ammontare della memoria fisica si abbassa fino al punto di raggiungere la soglia critica e i processi (solitamente scelti in quanto meno attivi) passano dalla memoria all'area d'avvicendamento, per liberare memoria. Nella pratica, pochissimi sistemi operativi moderni realizzano l'avvicendamento nel modo descritto: essi, infatti, combinano l'avvicendamento con tecniche di memoria virtuale (Capitolo 9) per coinvolgere nell'operazione solo alcune pagine, e non necessariamente interi processi. Tant'è che alcuni sistemi considerano *avvicendamento* e *paginazione* termini intercambiabili, a riprova della moderna tendenza a convergere di questi due concetti.

La gestione dell'area d'avvicendamento è un altro compito di basso livello del sistema operativo. La memoria virtuale usa lo spazio dei dischi come estensione della memoria centrale: poiché l'accesso alle unità a disco è molto più lento dell'accesso alla memoria centrale, l'uso di un'area d'avvicendamento riduce notevolmente le prestazioni del sistema. L'obiettivo principale nella progettazione e realizzazione di un'area d'avvicendamento è di fornire la migliore produttività per il sistema della memoria virtuale. In questo paragrafo sono trattati l'uso, la collocazione nei dischi e la gestione dell'area d'avvicendamento.

### 12.6.1 Uso dell'area d'avvicendamento

L'area d'avvicendamento è usata in modi diversi da sistemi operativi diversi, in funzione degli algoritmi di gestione della memoria applicati. I sistemi che adottano l'avvicendamento dei processi nella memoria, ad esempio, possono usare l'area d'avvicendamento per mantenere l'intera immagine del processo, inclusi i segmenti dei dati e del codice; i sistemi a paginazione, invece, possono semplicemente memorizzarvi pagine non contenute nella memoria centrale. Lo spazio richiesto dall'area d'avvicendamento per un sistema può quindi variare secondo la quantità di memoria fisica, la quantità di memoria virtuale che esso deve sostenere, e il modo in cui quest'ultima è usata.

Si noti che una stima per eccesso delle dimensioni dell'area d'avvicendamento è più prudente di una per difetto, perché un sistema che esaurisca l'area d'avvicendamento potrebbe essere costretto a terminare forzatamente i processi o ad arrestarsi completamente: una stima per eccesso spreca spazio dei dischi che si potrebbe usare per i file, ma non provoca altri danni. Alcuni sistemi consigliano la quantità da stanziare per l'area d'avvicendamento. Solaris, ad esempio, raccomanda di riservare a tal fine uno spazio uguale alla quantità di memoria virtuale che eccede la memoria fisica paginabile. Linux ha sempre suggerito di raddoppiare l'area d'avvicendamento rispetto alla memoria fisica, quantunque molti sistemi Linux usino attualmente un'area d'avvicendamento notevolmente minore. Addirittura, al-

l'interno della comunità Linux, è vivacemente discussa la mera opportunità di destinare o meno un'area all'avvicendamento!

Alcuni sistemi operativi, fra i quali Linux, permettono l'uso di aree d'avvicendamento multiple, poste di solito in unità a disco distinte per distribuire su più dispositivi il carico della paginazione e dell'avvicendamento dei processi gravante sul sistema per l'I/O.

## 12.6.2 Collocazione dell'area d'avvicendamento

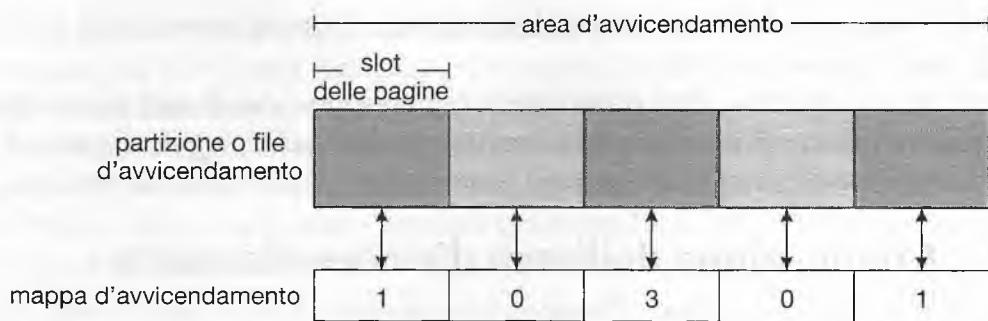
Le possibili collocazioni per un'area d'avvicendamento sono due: all'interno del normale file system, o in una partizione del disco a sé stante. Se l'area d'avvicendamento è semplicemente un grande file all'interno del file system, si possono usare le ordinarie funzioni del file system per crearla, assegnargli un nome, e allocare spazio per essa. Questo criterio sebbene sia semplice da realizzare è inefficiente: l'attraversamento della struttura delle directory e l'uso delle strutture dati per l'allocazione dello spazio nei dischi richiede tempo, oltre che, almeno potenzialmente, accessi ai dischi aggiuntivi. La frammentazione esterna può aumentare molto i tempi d'avvicendamento causando ricerche multiple durante la scrittura o la lettura dell'immagine di un processo. Le prestazioni si possono migliorare impiegando la memoria fisica come cache per le informazioni relative alla posizione dei blocchi, e anche usando strumenti speciali per l'allocazione in blocchi fisicamente contigui del file d'avvicendamento, ma il costo dovuto all'attraversamento del file system e delle sue strutture dati permane.

In alternativa, l'area d'avvicendamento si può creare in un'apposita partizione del disco non formattata: in essa non è presente alcuna struttura relativa al file system e alle directory, ma si usa uno speciale gestore dell'area d'avvicendamento per allocare e rimuovere i blocchi. Esso adotta algoritmi ottimizzati rispetto alla velocità, e non rispetto allo spazio impiegato, dato che all'area d'avvicendamento (quando viene utilizzata) si accede molto più frequentemente che ai file system. La frammentazione interna può aumentare, ma questo prezzo da pagare è ragionevole perché i dati nell'area d'avvicendamento hanno una vita media molto più breve dei file ordinari, e gli accessi all'area d'avvicendamento sono in genere molto più frequenti. Poiché l'area d'avvicendamento è inizializzata all'avvio, la frammentazione ha vita breve. Questo metodo assegna una dimensione fissa all'area d'avvicendamento al momento della creazione delle partizioni del disco, e l'aumento delle dimensioni dell'area d'avvicendamento deve quindi passare attraverso il ripartizionamento del disco (che implica lo spostamento o l'eliminazione e la sostituzione con copie di riserva delle altre partizioni del disco), o attraverso la creazione di un'altra area d'avvicendamento in qualche altra unità a disco del sistema.

Alcuni sistemi operativi non adottano una strategia rigida e possono costruire aree d'avvicendamento sia su partizioni specifiche sia all'interno del file system: Linux ne è un esempio. I metodi di gestione e la loro concreta realizzazione sono diversi nei due casi, e la scelta fra le due soluzioni è lasciata all'amministratore del sistema: sul piatto della bilancia pesano da un lato la convenienza dell'allocazione e della gestione dell'area d'avvicendamento all'interno del file system, e dall'altro le migliori prestazioni ottenibili grazie all'uso di una partizione specifica.

## 12.6.3 Gestione dell'area d'avvicendamento: un esempio

Si può comprendere come l'area d'avvicendamento venga applicata ripercorrendo la sua evoluzione e quella della paginazione nei vari sistemi UNIX. Il kernel tradizionale di UNIX implementava inizialmente una tecnica d'avvicendamento che spostava interi processi tra le regioni contigue del disco e la memoria. Più tardi, con la disponibilità dei dispositivi per la paginazione, UNIX progredì verso una combinazione d'avvicendamento e paginazione.



**Figura 12.10** Strutture dati per l'avvicendamento nei sistemi Linux.

Per migliorare l'efficienza e assimilare le innovazioni tecnologiche, in Solaris 1 (SunOS) i progettisti cambiarono le tecniche tradizionali di UNIX. Quando un processo va in esecuzione, le pagine contenenti il codice eseguibile sono prelevate dal file system, poste in memoria centrale e, qualora fossero state selezionate per l'espulsione, eliminate. Rileggere una pagina dal file system è più efficiente che scriverla nell'area d'avvicendamento e rileggerla da lì: l'area d'avvicendamento è adoperata esclusivamente come area di stoccaggio per le pagine della **memoria anonima**, di cui fanno parte la memoria allocata per la pila, quella per lo heap e i dati non inizializzati dei processi.

Ulteriori cambiamenti sono stati introdotti nelle versioni più recenti di Solaris. Il più importante prevede che l'area d'avvicendamento sia allocata solo quando una pagina è espulsa dalla memoria fisica, anziché quando la pagina è creata per la prima volta in memoria virtuale. Questa regola produce un miglioramento delle prestazioni nei calcolatori moderni, i quali, rispetto ai sistemi più vecchi, possono contare su una maggiore quantità di memoria fisica e limitare il ricorso alla paginazione.

Così come per Solaris, l'area d'avvicendamento di Linux è utilizzata soltanto per la memoria anonima o per regioni di memoria condivise tra numerosi processi. Linux permette di istituire una o più aree d'avvicendamento, sia in file di avvicendamento (*swap file*) del file system regolare, sia in una partizione a basso livello dedicata all'area di avvicendamento (*swap area*). Un'area d'avvicendamento è formata da una serie di moduli di 4 KB detti **slot delle pagine**, la cui funzione è di conservare le pagine scambiate. Ogni area d'avvicendamento ha una **mappa d'avvicendamento** (*swap map*) associata, ovvero un array di contatori interi, ciascuno dei quali corrisponde a uno slot nell'area d'avvicendamento. Se un contatore segna il valore 0, la pagina che gli corrisponde è disponibile. Valori superiori a 0 indicano che lo slot è occupato da una delle pagine scambiate. Il valore del contatore indica il numero di collegamenti alla pagina scambiata; se esso è 3, per esempio, allora la pagina scambiata è associata a tre processi differenti, eventualità possibile nel caso che la pagina rappresenti una regione di memoria condivisa da tre processi. Le strutture dati relative all'avvicendamento nei sistemi Linux sono rappresentate dalla Figura 12.10.

## 12.7 Strutture RAID

L'evoluzione tecnologica ha reso le unità a disco progressivamente più piccole e meno costose, tanto che oggi è possibile, senza eccessivi sforzi economici, equipaggiare un sistema di calcolo con molti dischi. La presenza di più dischi, qualora si possano usare in parallelo, rende possibile l'aumento della frequenza a cui i dati si possono leggere o scrivere. Inoltre, una

## STRUTTURA DEI DISPOSITIVI RAID

Le memorie di massa RAID si prestano a essere strutturate con modalità diverse. Un sistema, ad esempio, può collegare direttamente i dischi ai propri bus, nel qual caso la funzionalità RAID può essere realizzata dal sistema operativo o dai programmi di sistema. In alternativa, un controllore intelligente può gestire diversi dischi collegati alla stessa macchina e implementare una struttura RAID per quei dischi a livello hardware. Infine, si può ricorrere a una batteria RAID (*RAID array*), un'unità a sé stante, dotata di un controllore, di una cache (nella maggioranza dei casi) e di dischi autonomi. La batteria è collegata alla macchina attraverso uno o più controllori standard ATA SCSI o FC. Questa diffusa organizzazione consente a programmi e sistemi operativi di per sé privi della funzionalità RAID di usufruirne comunque. Persino sistemi che, in realtà, implementano le funzionalità RAID a livello software adottano a volte la tecnica descritta, per via della sua semplicità e flessibilità.

configurazione di questo tipo permette di migliorare l'affidabilità della memoria secondaria, poiché diventa possibile memorizzare le informazioni in più dischi in modo ridondante. In questo caso, un guasto a uno dei dischi non comporta la perdita di dati. Ci sono varie tecniche per l'organizzazione dei dischi, note col nome comune di **batterie ridondanti di dischi** (*redundant array of independent [inexpensive] disks*, RAID), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

Nel passato, strutture RAID composte da piccoli dischi economici erano viste come un'alternativa economicamente vantaggiosa rispetto a costosi dischi di grande capacità; oggi, le strutture RAID s'impiegano per la loro maggiore affidabilità e velocità di trasferimento dei dati, piuttosto che per ragioni economiche. Quindi, la *I* in RAID attualmente andrebbe letta *independent* anziché *inexpensive* com'era interpretata originariamente.

### 12.7.1 Miglioramento dell'affidabilità tramite la ridondanza

Consideriamo in primo luogo l'affidabilità. La possibilità che uno dei dischi in un insieme di  $n$  dischi si guasti è molto più alta della possibilità che uno specifico disco isolato presenti un guasto. Si supponga che il **tempo medio di guasto** di un singolo disco sia 100.000 ore. In questo caso, il tempo medio di guasto per un qualsiasi disco in una batteria di 100 dischi sarebbe  $100.000/100 = 1000$  ore, o 41,66 giorni; cioè non molto tempo. Se si memorizzasse una sola copia dei dati, allora ogni guasto di un disco comporterebbe la perdita di una notevole quantità di dati; una frequenza di perdita di dati così alta sarebbe inaccettabile.

La soluzione al problema dell'affidabilità sta nell'introdurre una certa **ridondanza**, cioè nel memorizzare informazioni che non sono normalmente necessarie, ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse.

Il metodo più semplice (ma anche il più costoso) di introduzione di ridondanza è quello della **copiatura speculare** (*mirroring* o *shadowing*); ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Se uno dei dischi si guasta, i dati si possono leggere dall'altro. I dati si perdono solo se il secondo disco si guasta prima della sostituzione del disco già guasto.

Il tempo medio di guasto di un disco con copiatura speculare, dove per *guasto* s'intende ora la perdita di dati, dipende da due fattori: il tempo medio di guasto di un singolo disco e il **tempo medio di riparazione**, cioè il tempo richiesto (in media) per sostituire un disco guasto e ripristinarvi i dati. Supponendo che i possibili guasti dei due dischi siano indi-

pendenti, vale a dire che il guasto di un disco non sia mai legato a quello dell'altro, se il tempo medio di guasto di un singolo disco è 100.000 ore e il tempo medio di riparazione è di 10 ore, allora il tempo medio di perdita di dati di un sistema con copiatura speculare dei dischi è  $100.000^2/(2 \times 10) = 500 \times 10^6$  ore, che corrispondono a 57.000 anni!

Occorre però notare che l'ipotesi di indipendenza tra i guasti dei dischi non è in realtà valida, poiché improvvisi cali di tensione e disastri naturali, quali terremoti, incendi e alluvioni, danneggierebbero con tutta probabilità entrambi i dischi. Inoltre, difetti di fabbricazione in una partita di dischi possono causare guasti simili e correlati. Con l'invecchiamento del disco, la probabilità di un guasto aumenta, accrescendo la probabilità che un secondo disco si guasti mentre il primo è in riparazione. Tuttavia, nonostante tutte queste considerazioni, i sistemi con copiatura speculare dei dischi offrono un'affidabilità assai più alta dei sistemi a disco singolo.

I casi di improvvisa mancanza di tensione elettrica costituiscono un problema particolarmente sentito, poiché avvengono con una frequenza molto più alta dei disastri naturali. Tuttavia, anche impiegando la copiatura speculare dei dischi, se si sta svolgendo un'operazione di scrittura nello stesso blocco in entrambi i dischi e si verifica una mancanza di tensione prima che sia completata la scrittura dell'intero blocco, i due blocchi possono ritrovarsi in uno stato incoerente. Una soluzione prevede la scrittura di una delle due copie e solo successivamente la scrittura della seconda, così che una delle due copie sia sempre coerente. Un'altra soluzione è aggiungere una memoria cache non volatile (NVRAM, *non-volatile RAM*) alla batteria RAID, protetta dalla perdita di dati causata dalle cadute di tensione; se è dotata di forme di correzione d'errore come ECC o copiatura speculare i dati nella cache possono essere considerati completi.

## 12.7.2 Miglioramento delle prestazioni tramite il parallelismo

L'accesso in parallelo a più dischi può portare vari vantaggi. Con la copiatura speculare dei dischi, la frequenza con la quale si possono gestire le richieste di lettura raddoppia, poiché ciascuna richiesta si può inviare indifferentemente a uno dei due dischi (sempre che entrambi i dischi siano funzionanti, condizione che è quasi sempre soddisfatta). La capacità di trasferimento di ciascuna lettura è la stessa di quella di un sistema a singolo disco, ma il numero di letture per unità di tempo raddoppia.

Attraverso l'uso di più dischi è possibile anche (o alternativamente) migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi. Nella sua forma più semplice questa distribuzione, chiamata **sezionamento dei dati** (*data striping*), consiste nel distribuire i bit di ciascun byte su più dischi; in questo caso si parla di **sezionamento a livello dei bit**. Ad esempio, se il sistema impiega una batteria di otto dischi, si scriverà il bit  $i$  di ciascun byte nel disco  $i$ . La batteria di otto dischi si può trattare come un unico disco avente settori che hanno una dimensione otto volte superiore a quella normale e, soprattutto, che hanno una capacità di trasferimento otto volte superiore. In un'organizzazione di questo tipo, ogni disco è coinvolto in ogni accesso (lettura o scrittura che sia), così che il numero di accessi che si possono gestire nell'unità di tempo è circa lo stesso di quello per un sistema a disco singolo, ma ogni accesso permette di leggere una quantità di dati pari a otto volte quella che si può leggere con un singolo disco.

Il sezionamento a livello dei bit si può generalizzare a un numero di dischi multiplo di 8 o che divide 8. Ad esempio, se un sistema adopera una batteria di quattro dischi, i bit  $i$  e  $i + 4$  di ciascun byte si memorizzano nel disco  $i$ . Inoltre, il sezionamento non si deve realizzare necessariamente a livello dei bit di un byte: nel **sezionamento a livello dei blocchi**, ad

esempio, i blocchi di un file si distribuiscono su più dischi; con  $n$  dischi, il blocco  $i$  di un file si memorizza nel disco  $(i \bmod n) + 1$ . Sono possibili anche altri livelli di sezionamento, come quelli basati sui byte di un settore o sui settori di un blocco.

Riassumendo, gli obiettivi principali riguardo al parallelismo in un sistema di dischi sono due:

1. l'aumento, tramite il bilanciamento del carico, della produttività per accessi multipli a piccole porzioni di dati (cioè accessi a pagine);
2. la riduzione del tempo di risposta relativo ad accessi a grandi quantità di dati.

### 12.7.3 Livelli RAID

La tecnica di copiatura speculare offre un'alta affidabilità ma è costosa; la tecnica del sezionamento offre un'alta capacità di trasferimento dei dati, ma non migliora l'affidabilità. Sono stati proposti numerosi schemi per fornire ridondanza usando l'idea del sezionamento combinata con i bit di parità. Questi schemi realizzano diversi compromessi tra costi e prestazioni e sono stati classificati in livelli chiamati **livelli RAID**, che la Figura 12.11 mostra

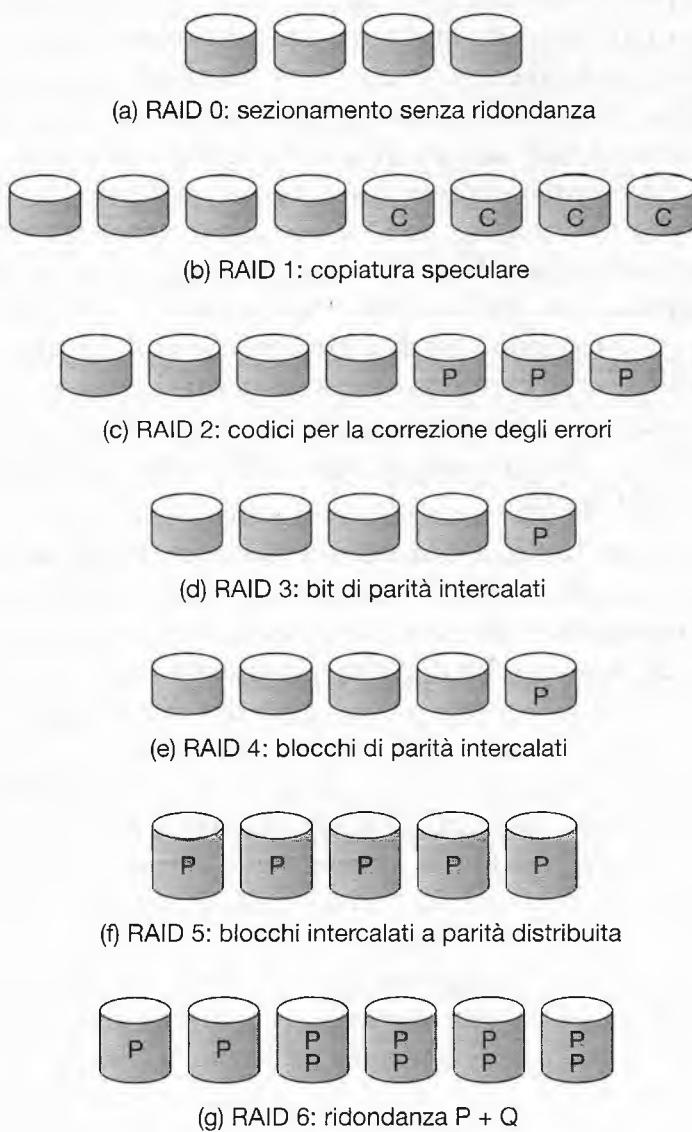


Figura 12.11 Livelli RAID.

graficamente (nella figura, la lettera  $P$  indica i bit di correzione degli errori, la lettera  $C$  indica una seconda copia dei dati). In tutti i casi riportati nella figura, sono presenti quattro dischi di dati, mentre i dischi supplementari s'impiegano per memorizzare le informazioni ridondanti per il ripristino dai guasti.

- **RAID di livello 0.** Il livello 0 si riferisce a batterie di dischi con sezionamento a livello dei blocchi, ma senza ridondanza (come la copiatura speculare o i bit di parità), come illustrato nella Figura 12.11(a).
- **RAID di livello 1.** Il livello 1 si riferisce alla tecnica della copiatura speculare. La Figura 12.11(b) mostra un'organizzazione basata sulla copiatura speculare.
- **RAID di livello 2.** Il livello 2 è anche noto come **organizzazione con codici per la correzione degli errori** (*error-correcting codes*, ECC). Da molto tempo i sistemi di memorizzazione impiegano tecniche di riconoscimento degli errori basate sui bit di parità. In un sistema di questo tipo, ogni byte di memoria ha associato un bit di parità che indica se i bit con valore 1 nel byte sono in numero pari (parità = 0) oppure dispari (parità = 1). Se si altera uno dei bit nel byte (un valore 1 diventa 0 o viceversa), la parità del byte cambia e quindi non concorda più con la parità memorizzata. Analogamente, se si altera il bit di parità, esso non concorda più con la parità calcolata. In questo modo s'identificano tutti gli errori di un singolo bit nel sistema di memoria. Gli schemi di correzione degli errori memorizzano due o più bit supplementari e possono ricostruire i dati nel caso di un singolo bit danneggiato. La stessa idea alla base dell'ECC si può usare immediatamente nelle batterie di dischi eseguendo il sezionamento dei byte presenti nei dischi. Ad esempio, il primo bit di ogni byte si potrebbe memorizzare nel disco 1, il secondo bit nel disco 2, e così via fino alla memorizzazione dell'ottavo bit nel disco 8 e alla memorizzazione dei bit di correzione degli errori in ulteriori dischi. Questo schema è rappresentato graficamente nella Figura 12.11(c), dove i dischi etichettati con la lettera  $P$  contengono i bit di correzione. Se uno dei dischi si guasta, i bit rimanenti del byte e i bit di correzione a esso associati si possono leggere dagli altri dischi e usare per ricostruire i dati danneggiati. Si noti che il RAID di livello 2 richiede tre soli dischi in più per quattro dischi di dati, a differenza del RAID di livello 1, che richiede quattro dischi in più.
- **RAID di livello 3.** Con il livello 3, o **organizzazione con bit di parità intercalati**, si migliora l'organizzazione del livello 2 considerando che, a differenza dei sistemi di memoria centrale, i controllori dei dischi possono rilevare se un settore è stato letto correttamente, così che un unico bit di parità si può usare sia per individuare gli errori sia per correggerli. L'idea è la seguente: se uno dei settori è danneggiato, si conosce esattamente di quale settore si tratta e, per ogni bit nel settore, è possibile determinare se debba avere valore 1 o 0 calcolando la parità dei bit corrispondenti dai settori negli altri dischi. Se la parità dei rimanenti bit è uguale a quella memorizzata, il bit mancante è 0, altrimenti è 1. Il RAID di livello 3 è altrettanto valido del livello 2 ma richiede un solo disco supplementare, quindi il RAID di livello 2 non è utilizzato nella pratica. La Figura 12.11(d) illustra questo schema.

Il livello 3 presenta due vantaggi rispetto a livello 1. Il primo vantaggio è che si usa un solo disco per la parità dei dati memorizzati in diversi dischi di dati, anziché un ulteriore disco per ciascun disco di dati come nel livello 1. Il secondo vantaggio è che, essendo le letture e le scritture dei byte distribuite su più dischi con un sezionamento dei dati a  $n$  vie, la velocità di trasferimento di un singolo blocco è pari a  $n$  volte quella dei RAID di livello 1. D'altro canto, però, il livello 3 permette meno operazioni di I/O al secondo, perché ogni disco è coinvolto da tutte le richieste.

Un altro problema di prestazioni riguardante il RAID di livello 3 (come per tutti i livelli RAID basati sui bit di parità) è il tempo richiesto dal calcolo e dalla scrittura della parità. Questo tempo aggiuntivo determina operazioni di scrittura significativamente più lente rispetto a batterie RAID senza parità. Per limitare questo calo di prestazioni, molte batterie RAID dispongono di un controllore capace di gestire il calcolo della parità. Questo sposta il carico dovuto al calcolo della parità dalla CPU alla batteria di dischi. La batteria ha anche una cache RAM **non volatile** (NVRAM) per memorizzare i blocchi mentre viene calcolata la parità e per memorizzare transitoriamente le scritture dal controllore ai dischi. Questa combinazione può rendere la tecnica RAID con parità altrettanto veloce di quella senza parità; infatti, una batteria RAID con cache e bit di parità può avere prestazioni migliori di un'organizzazione RAID senza cache e senza parità.

- ◆ **RAID di livello 4.** Nel livello 4, o **organizzazione con blocchi di parità intercalati**, s'impiega il sezionamento a livello dei blocchi, come nel RAID di livello 0 e inoltre si tiene un blocco di parità in un disco separato per i blocchi corrispondenti presenti in  $n$  dischi diversi da questo. Tale schema è illustrato nella Figura 12.11(e). Se uno dei dischi si guasta, il blocco di parità si può usare insieme ai blocchi corrispondenti degli altri dischi per ripristinare i blocchi nel disco guasto.

La lettura di un blocco richiede l'accesso a un solo disco, permettendo la gestione di altre richieste da parte di altri dischi. Quindi, la capacità di trasferimento dei dati per ciascun accesso è minore, ma gli accessi per la lettura possono procedere in modo parallelo ottenendo una rapidità complessiva nell'I/O più alta. La capacità di trasferimento per la lettura di molti dati è alta, poiché si possono leggere in modo parallelo tutti i dischi e anche le operazioni di scrittura di grandi quantità di dati presentano un'alta capacità di trasferimento, poiché i dati e i bit di parità si possono scrivere in parallelo.

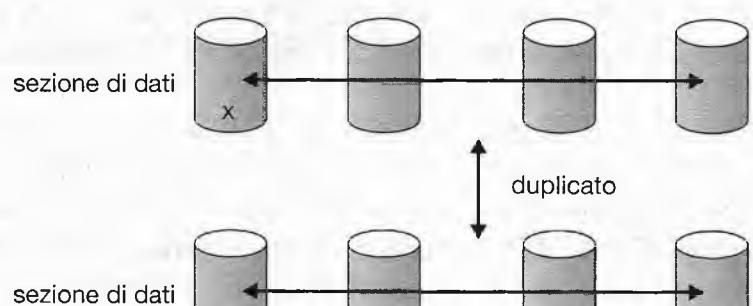
Scritture indipendenti di modesta entità non si possono eseguire in parallelo. La scrittura di dati pari a meno di un blocco richiede la lettura del blocco, la sua modifica, e la scrittura del blocco modificato; anche il blocco di parità deve essere aggiornato. Si parla a questo proposito del **ciclo lettura-modifica-scrittura**. Una singola richiesta di scrittura comporta pertanto quattro accessi al disco, due in lettura e due in scrittura.

Nel Capitolo 11 è stato presentato WAFL: esso adotta RAID di livello 4, che permette l'aggiunta di dischi al sistema senza soluzione di continuità. Inizializzando i nuovi dischi a zero, la parità non cambia, e la batteria RAID è ancora nello stato corretto.

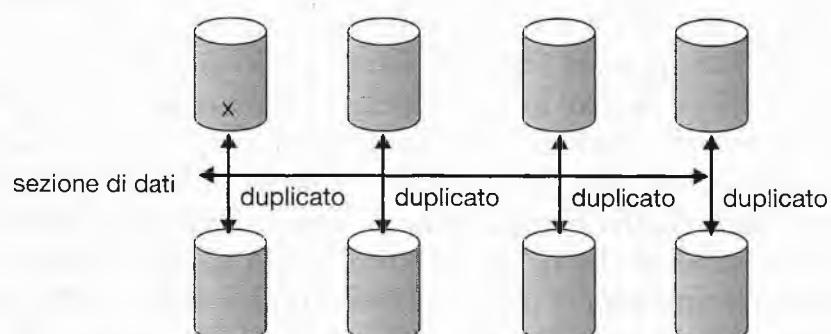
- ◆ **RAID di livello 5.** Il livello 5, o **organizzazione con blocchi intercalati a parità distribuita**, differisce dal livello 4 per il fatto che, invece di memorizzare i dati in  $n$  dischi e la parità in un disco separato, i dati e le informazioni di parità sono distribuite tra gli  $n + 1$  dischi. Per ogni blocco, uno dei dischi memorizza la parità e gli altri i dati. Ad esempio, considerando una batteria di cinque dischi, la parità per il blocco  $m$ -esimo si memorizza nel disco  $(m \bmod 5) + 1$ , mentre i blocchi  $m$ -esimi degli altri quattro dischi contengono i dati effettivi per quel blocco. Questo schema è illustrato nella Figura 12.11(f), dove i simboli  $P$  sono distribuiti su tutti i dischi. Un blocco di parità non può contenere informazioni di parità per blocchi che risiedono nello stesso disco, poiché un guasto al disco provocherebbe sia la perdita di dati sia la perdita dell'informazione di parità e quindi i dati non sarebbero ripristinabili. Con la distribuzione della parità sui diversi dischi, il RAID di livello 5 evita un uso intensivo del disco dove risiede la parità, che invece si ha con il RAID di livello 4. RAID 5 è il più comune sistema di parità RAID.

- ◆ **RAID di livello 6.** Il livello 6, detto anche schema di ridondanza P + Q, è molto simile al RAID di livello 5, ma memorizza ulteriori informazioni ridondanti per poter gestire guasti contemporanei di più dischi. Invece di usare la parità, s'impiegano codici per la correzione degli errori come i **codici di Reed-Solomon**. Nello schema mostrato nella Figura 12.11(g), sono memorizzati 2 bit di dati ridondanti ogni 4 bit di dati effettivi (a differenza di 1 bit di parità usato nel livello 5) e il sistema risultante può tollerare due guasti dei dischi.
- ◆ **RAID di livello 0 + 1.** Il livello 0 + 1 consiste in una combinazione dei livelli RAID 0 e 1. Il livello 0 fornisce le prestazioni, mentre il livello 1 l'affidabilità. Di solito, questo schema porta a prestazioni migliori rispetto a livello 5 e si usa prevalentemente negli ambienti in cui sono importanti sia le prestazioni sia l'affidabilità. Sfortunatamente, questo schema richiede, come il RAID di livello 1, un raddoppio del numero di dischi necessario per memorizzare i dati, quindi è anche più costoso del RAID di livello 5. Nel RAID di livello 0 + 1, si sezionano i dati presenti in un insieme di dischi e si duplica ogni sezione con la tecnica della copiatura speculare.

Un altro metodo che sta diventando disponibile commercialmente è il RAID di livello 1 + 0, in cui si fa prima la copiatura speculare dei dischi a coppie, e poi il sezionamento su queste coppie. Questo schema RAID ha alcuni vantaggi teorici rispetto al RAID 0 + 1. Ad esempio, se si guasta un singolo disco nel RAID 0 + 1, l'intera sezione di dati diventa inaccessibile, lasciando disponibile solo l'altra sezione. Con un guasto nel RAID 1 + 0, il singolo disco diventa inaccessibile, ma il suo duplice è ancora disponibile, come tutti gli altri dischi (Figura 12.12).



a) RAID 0 + 1 con guasto di un solo disco



b) RAID 1 + 0 con guasto di un solo disco

**Figura 12.12** RAID 0 + 1 e 1 + 0.

Infine, si deve considerare che sono state proposte numerose altre varianti agli schemi RAID di base illustrati sopra e questo ha portato anche una certa confusione nelle precise definizioni dei diversi livelli RAID.

Un altro punto soggetto a molte varianti è l'implementazione del RAID. Esaminiamo i diversi livelli a cui è possibile implementare un sistema RAID.

- ◆ Il software per la gestione dei volumi può implementare un sistema RAID all'interno del kernel o a livello dei programmi di sistema. In questo caso, nonostante i dispositivi per la memorizzazione possano fornire funzionalità minime, è possibile ottenere un sistema RAID completo. Il metodo RAID con parità è alquanto lento se realizzato tramite software, quindi gli si preferisce solitamente RAID 0,1 oppure 0 + 1.
- ◆ Il metodo RAID può essere implementato a livello hardware dall'adattatore del bus della macchina (*host bus adapter*, HBA). Solo i dischi connessi direttamente all'HBA possono costituire parte integrante di una data batteria RAID. Questa soluzione è a basso costo, ma non può dirsi molto flessibile.
- ◆ Il metodo RAID può essere implementato a livello hardware dalla batteria di dischi. È così possibile creare sistemi RAID a vari livelli, e persino ricavare da essi volumi più piccoli, che sono quindi presentati al sistema operativo, che avrà solo da realizzare il file system su ciascuno dei volumi. Le batterie possono disporre di connessioni multiple o far parte di una rete di memorizzazione secondaria (SAN), consentendo a vari terminali di sfruttare le funzionalità della batteria.
- ◆ Il metodo RAID può essere implementato dai dispositivi di virtualizzazione del disco a livello di interconnessione SAN. In questo caso, un dispositivo funge da intermediario tra i terminali e l'area di memorizzazione, accettando istruzioni dai server e gestendo l'accesso alla memoria secondaria. Esso potrebbe, ad esempio, attuare la copiatura speculare, trascrivendo ciascun blocco su due dispositivi distinti per la memorizzazione.

Ulteriori funzionalità, come quella di istantanea e di replica, possono essere implementate a ognuno di questi livelli. La **replica**, che può essere sincrona o asincrona, concerne la duplicazione automatica di scritture su siti diversi, per finalità di ridondanza, o di ripristino in caso di danneggiamenti. Se è sincrona, ciascun blocco deve essere scritto sia localmente, sia in remoto, prima che la scrittura sia considerata completa; se è asincrona, si effettuano scritture per gruppi e a cicli periodici. La replica asincrona espone al rischio di perdere i dati, se il sito principale fallisce, ma è più veloce e non ha limiti di distanza.

L'implementazione di queste funzionalità varia a seconda del livello scelto per realizzare il sistema RAID. Qualora RAID, per esempio, sia implementato a livello software, ciascuna macchina può aver necessità di implementare e gestire la replica per proprio conto. Tuttavia, se la replica avviene a livello della batteria di dischi o dell'interconnessione SAN, si possono replicare i dati della macchina a prescindere dalla piattaforma del sistema operativo e dalle sue funzionalità.

Un'altra caratteristica spesso presente nei sistemi RAID è la previsione di **dischi di scorta** (*hot spare*), che possono sostituire quelli normali. Quando, infatti, un disco presenta difetti di funzionamento, subentra al suo posto un disco di scorta, ad esempio per ricostruire un disco danneggiato, che appartiene a una coppia configurata per la copiatura speculare. In questo modo, si può ristabilire automaticamente lo stato corretto del livello RAID, senza attendere che il disco difettoso sia sostituito. È possibile riparare più di un guasto, senza l'intervento di un operatore, con l'allocazione di più dischi di scorta.

### 12.7.4 Scelta di un livello RAID

Viste le svariate possibilità esistenti, ci si potrebbe chiedere quali siano i criteri di scelta dei progettisti nei confronti del livello RAID. Se un disco si guasta, il tempo necessario a ricostruire i dati che contiene può essere rilevante. Questo fattore può essere importante nel caso in cui venga richiesto un flusso continuo di dati, come nei sistemi ad alte prestazioni o nei sistemi interattivi di basi di dati. Inoltre, le prestazioni del processo di ricostruzione influenzano il tempo medio di guasto.

Queste prestazioni possono variare a seconda del livello RAID utilizzato. La ricostruzione più semplice si ha per RAID di livello 1, poiché i dati possono essere copiati da un altro disco; per gli altri livelli, per ricostruire i dati in un disco guasto è necessario accedere a tutti gli altri dischi della batteria. Il tempo necessario per la ricostruzione dei dati può variare nell'ordine di ore nel caso di sistemi RAID di livello 5 con molti dischi.

RAID di livello 0 si usa nelle applicazioni ad alte prestazioni in cui le perdite di dati non sono critiche. Il RAID di livello 1 si usa comunemente nelle applicazioni che richiedono un'alta affidabilità e un rapido ripristino. I livelli RAID 0 + 1 e 1 + 0 si usano dove le prestazioni e l'affidabilità sono importanti, ad esempio per piccole basi di dati. A causa dell'elevata richiesta di spazio del RAID di livello 1, per la memorizzazione di grandi quantità di dati, spesso si preferisce impiegare il RAID di livello 5. Il livello 6, attualmente non disponibile in molti sistemi RAID, dovrebbe offrire una migliore affidabilità rispetto a livello 5.

Progettisti e amministratori di sistemi RAID devono prendere anche altre decisioni importanti, ad esempio riguardo al numero ottimale di dischi in una batteria e al numero di bit che ciascun bit di parità deve proteggere. Maggiore è il numero di dischi in una batteria, maggiore sarà la capacità di trasferimento dei dati, ma il sistema sarà anche più costoso. Maggiore è il numero di bit protetti da un singolo bit di parità, minore sarà lo spazio richiesto dai bit di parità; sarà però maggiore anche la probabilità che un secondo disco si guasti prima che un disco guasto sia riparato e questo porterebbe alla perdita di dati.

### 12.7.5 Estensioni

I concetti relativi ai sistemi RAID sono stati generalizzati ad altri dispositivi di memorizzazione, comprese le batterie di nastri e anche alla diffusione dei dati tramite sistemi senza fili (*wireless*). Con strutture RAID applicate alle batterie di nastri si possono ripristinare i dati anche se uno dei nastri della batteria è danneggiato. Se applicate alla trasmissione dei dati, si divide ogni blocco di dati in unità più piccole che si trasmettono insieme a un'unità di parità; se per qualsiasi ragione una delle unità non viene ricevuta, può essere ricostruita dalle altre. Di solito, con l'uso di unità automatiche dotate di molte unità a nastro si esegue il sezionamento dei dati su tutte le unità per aumentare la produttività e diminuire il tempo di trasferimento dei dati.

### 12.7.6 Problemi connessi a RAID

I sistemi RAID, purtroppo, non assicurano sempre la disponibilità dei dati al sistema operativo e ai loro utenti. Un puntatore potrebbe indicare il file sbagliato, per esempio, e lo stesso potrebbe accadere ai puntatori nella struttura interna dei file. Le operazioni incomplete di scrittura, se non ripristinate in maniera adeguata, possono alterare i dati. Altri processi, inoltre, potrebbero scrivere accidentalmente sulle strutture del file system. Il metodo RAID protegge dagli errori derivanti dai supporti fisici per la memorizzazione, ma non da altri tipi di errori dovuti ai dispositivi e ai programmi. I pericoli potenziali, per i dati di un sistema, si estendono alla totalità degli errori derivanti dal software e dall'hardware.

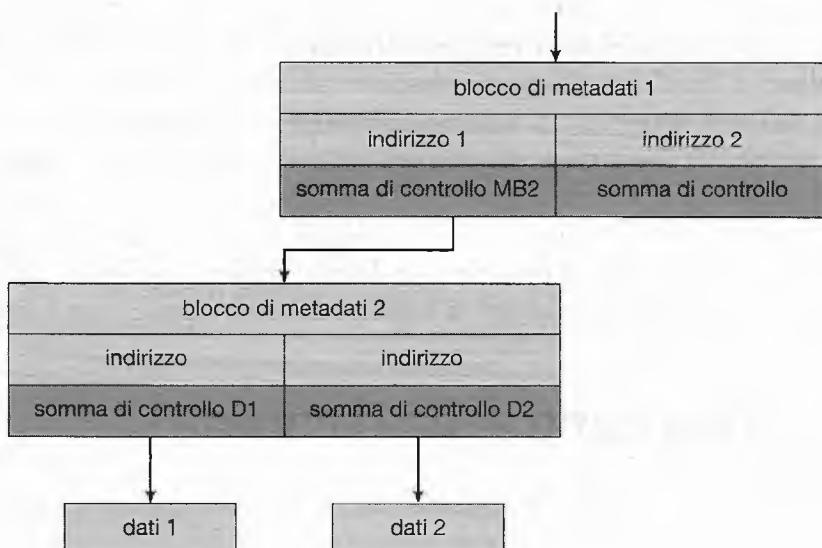
## LA BATTERIA InServ

Il progresso, grazie al quale sono di continuo introdotte soluzioni migliori, più veloci e meno costose, ridefinisce spesso i confini che separano le tecnologie già esistenti. Si consideri, per esempio, la batteria InServ di 3Par. A differenza di molte altre, essa non richiede la configurazione di un insieme di dischi a un livello RAID specifico, ma scomponete invece ogni disco in porzioni da 256 MB. Il metodo RAID è pertanto applicato a livello di queste porzioni. Di conseguenza, vari livelli RAID possono interessare lo stesso disco, visto che le porzioni servono per formare volumi multipli.

La batteria InServ mette inoltre a disposizione le istantanee, una funzionalità simile a quella del file system WAFL. Le istantanee di InServ prevedono sia il formato lettura-scrittura sia il formato a sola lettura, consentendo a utenti multipli di montare copie di un dato file system senza doverlo possedere integralmente. Le modifiche eventualmente apportate dagli utenti alla propria copia sono di copiatura su scrittura, ragion per cui non hanno effetto sulle altre copie.

Un'altra innovazione è il cosiddetto *utility storage*. Alcuni file system non possono essere espansi, né compressi. I file system di questo genere mantengono costantemente le dimensioni originali: per qualsiasi modifica è necessaria la copiatura di dati. Un amministratore può configurare InServ per fornire a una macchina cospicue quantità di memoria logica, che all'inizio occupano solamente un piccolo spazio di memoria fisica. Mentre la macchina comincia a usare lo spazio di memorizzazione, dischi non ancora utilizzati sono assegnati alla macchina, fino a raggiungere il livello logico originale. In tal modo, la macchina utente è indotta a credere di possedere un vasto spazio di memorizzazione permanente, dove creare i propri file system, e così via. InServ può aggiungere o rimuovere dischi dal file system, senza che se ne accorga. Questa caratteristica può ridurre il numero di unità a disco necessarie agli utenti, o perlomeno ritardare l'acquisizione di nuovi dischi finché non divengano realmente necessari.

Per risolvere tali problemi, il file system Solaris ZFS ricorre a una strategia innovativa. Esso applica una somma di controllo (*checksum*) interna a ogni blocco, dati e metadati inclusi. Un'ulteriore funzionalità deriva dalla collocazione delle somme di controllo, che non risiedono nel blocco sottoposto a controllo: ciascuna di loro, invece, è memorizzata insieme al puntatore a quel blocco (Figura 12.13). Si consideri un inode con puntatori ai propri dati. All'interno dell'inode si trova la somma di controllo per ciascun blocco di dati. Se si verifica



**Figura 12.13** ZFS applica una somma di controllo a tutti i metadati e i dati.

un problema con i dati, la somma di controllo darà un valore errato e il file system verrà a conoscenza del problema. Qualora sia attiva la copiatura speculare, il sistema ZFS, in presenza di un blocco con una somma di controllo corretta e di uno con una somma errata, sostituirà automaticamente il blocco errato con quello valido. In maniera simile, l'elemento della directory che punta all'inode possiede una somma di controllo relativa all'inode. Qualunque problema riguardi l'inode, quindi, è rilevato dall'accesso alla directory. Queste somme di controllo, che sono applicate a tutte le strutture di ZFS, producono risultati molto più efficaci degli ambienti RAID o dei file system tradizionali, per livello di coerenza, rilevazione degli errori e capacità di correggerli. Il sovraccarico di gestione determinato dalle somme di controllo e dai cicli supplementari di lettura-modifica-scrittura dei blocchi non condiziona il funzionamento complessivo di ZFS, che mantiene un'alta velocità nelle prestazioni.

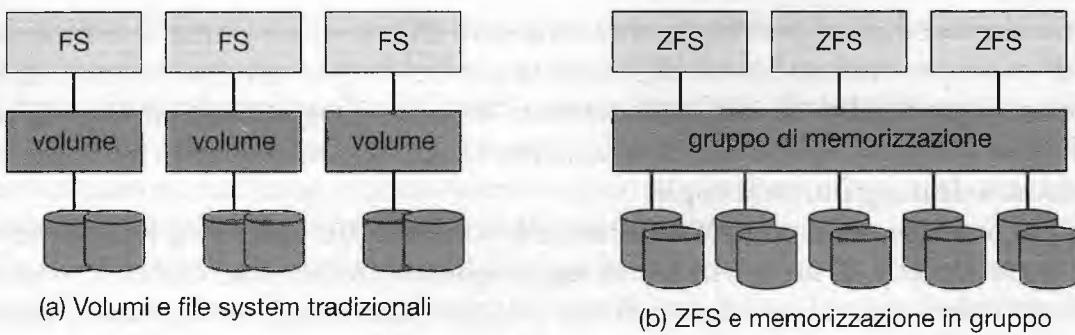
La maggior parte delle implementazioni RAID è caratterizzata dalla mancanza di flessibilità. Considerate una batteria di memorizzazione dotata di venti dischi divisi in quattro insiemi da cinque dischi. Ogni gruppo di cinque dischi è un insieme RAID di livello 5. Ne risultano quattro volumi separati, ciascuno contenente un proprio file system. Ma che cosa succede se un file system ha una dimensione eccessiva per un insieme RAID di livello 5 a cinque dischi? E se un altro file system necessita di un'area ridottissima? Se tali fattori sono noti in anticipo, allora i dischi e i volumi possono essere allocati adeguatamente. L'utilizzo del disco e le richieste variano però molto frequentemente nel tempo.

Anche se la batteria di memorizzazione ha permesso all'intero insieme di venti dischi di essere creato come un grande insieme RAID, potrebbero insorgere altre problematiche. Nell'insieme potrebbero essere costruiti diversi volumi di dimensioni differenti. Alcuni gestori del volume non ci permettono però di cambiare la dimensione di un volume. In quel caso si ripresenterebbe la stessa situazione descritta in precedenza, ovvero dimensioni discordanti di file system. Alcuni gestori di volume permettono cambiamenti di dimensione, ma alcuni file system non permettono l'aumento o la diminuzione della dimensione del file system stesso. I volumi potrebbero cambiare dimensione, ma i file system dovrebbero essere ricreati per poter usufruire di quei cambiamenti.

ZFS combina la gestione dei file system e quella dei volumi in una unità in grado di offrire una maggiore funzionalità rispetto a quella permessa dalla tradizionale separazione di tali funzioni. I dischi, o le partizioni di dischi, sono riuniti in **gruppi di memorizzazione** (*pools of storage*) attraverso insiemi RAID. Un gruppo può contenere uno o più file system ZFS. Tutta l'area libera di un gruppo è a disposizione di tutti i file system contenuti all'interno di quel gruppo. Quando i blocchi vengono utilizzati e liberati all'interno del file system ZFS utilizza "malloc" e "free" per allocare e rilasciare memoria in ogni file system. Di conseguenza non ci sono limiti artificiali all'utilizzo della memoria e non sussiste la necessità di redistribuire i file system tra i volumi né di ridimensionare i volumi. ZFS stabilisce delle quote per limitare la dimensione di un file system e definisce dei parametri per assicurarsi che il file system possa aumentare all'interno di una dimensione specificata, ma queste variabili possono essere sempre cambiate dal proprietario del file system. La Figura 12.14(a) illustra i volumi e i file system tradizionali, mentre la Figura 12.14(b) rappresenta il modello ZFS.

## 12.8 Realizzazione della memoria stabile

Nel Capitolo 6 è stata presentata la registrazione con scrittura anticipata (*write-ahead logging*), che richiede la disponibilità di una memoria stabile. Per definizione, le informazioni residenti in questo tipo di memoria non vanno *mai* perse. Per realizzare un tale tipo di



**Figura 12.14** (a) Volumi e file system tradizionali. (b) Gruppo e file system ZFS.

memoria si devono replicare le informazioni necessarie in più dispositivi di memorizzazione (di solito dischi) con modalità di guasto indipendenti. Inoltre è necessario coordinare l'aggiornamento delle informazioni in modo tale che un eventuale malfunzionamento durante l'aggiornamento non lasci tutte le copie in uno stato danneggiato, e che durante il ripristino delle informazioni a seguito di un guasto sia possibile riportare ogni copia alla sua forma corretta anche se si verifica un altro malfunzionamento proprio durante il ripristino. Nel resto del paragrafo si presentano le possibili soluzioni a queste esigenze.

Un'operazione di scrittura in un disco può avere uno dei seguenti esiti.

1. **Operazione riuscita.** I dati sono stati scritti correttamente nel disco.
2. **Insuccesso parziale.** Si è verificato un guasto durante il trasferimento, e solo alcuni tra i settori coinvolti sono stati correttamente aggiornati, mentre il settore interessato dalla scrittura al momento del malfunzionamento può essere stato danneggiato.
3. **Insuccesso totale.** Il malfunzionamento è avvenuto prima dell'avvio del processo di scrittura nel disco; i dati già residenti nel disco sono rimasti inalterati.

Si richiede che il sistema riconosca un guasto verificatosi durante il trasferimento e invochi una procedura di ripristino per riportare il blocco in uno stato coerente. A tale scopo il sistema deve mantenere due blocchi fisici per ciascun blocco logico, eseguendo ogni operazione nel modo seguente:

1. scrittura delle informazioni nel primo blocco fisico;
2. completata con successo la prima scrittura, scrittura delle stesse informazioni nel secondo blocco fisico;
3. l'operazione è considerata completa solamente dopo che la seconda scrittura è stata eseguita correttamente.

Durante il ripristino dovuto a un malfunzionamento si esamina ogni coppia di blocchi fisici; se essi contengono gli stessi dati e non c'è traccia d'errori, non è necessario intraprendere alcuna azione ulteriore. Se un blocco contiene un errore riscontrabile, se ne sostituisce il contenuto con quello del secondo blocco. Se in nessuno dei due blocchi si riscontra un errore, ma ciascuno contiene informazioni differenti, si sostituisce il contenuto del primo blocco con quello del secondo. Questa procedura di ripristino assicura che gli unici due esiti possibili di un'operazione di scrittura nella memoria stabile siano o la completa riuscita dell'operazione oppure il suo insuccesso totale, in quest'ultimo caso i dati memorizzati non subiscono alcun cambiamento.

Questa procedura si può facilmente estendere all'uso di un numero arbitrariamente grande di copie per ciascun blocco di memoria stabile. Benché un gran numero di queste copie riduca la probabilità di malfunzionamenti, di solito è ragionevole simulare la memoria stabile con due sole copie. I dati di una memoria stabile non andranno mai persi purché un guasto non distrugga tutte le copie.

Poiché le attese per il completamento delle scritture (I/O sincrono) richiedono molto tempo, molte batterie di memorizzazione aggiungono NVRAM come cache. Poiché la memoria è non volatile (di solito è dotata di una pila per l'alimentazione elettrica di riserva), si può ritenere affidabile come un disco per la memorizzazione dei dati, e si può considerare parte della memoria stabile. Le scritture in essa sono molto più rapide di quelle nei dischi, quindi le prestazioni migliorano notevolmente.

## 12.9 Strutture per la memorizzazione terziaria

Nessuno comprerebbe un videoregistratore contenente un'unica cassetta non sostituibile o non estraibile, o un lettore di CD o DVD con un disco bloccato all'interno: ci si attende di poter usare un videoregistratore o un lettore di CD con videocassette o dischi diversi e relativamente poco costosi. Anche nei calcolatori l'uso di supporti di memorizzazione rimovibili economici e di un solo dispositivo di lettura o scrittura riduce i costi complessivi. Il costo contenuto è la caratteristica distintiva della memoria terziaria, argomento di questo paragrafo.

### 12.9.1 Dispositivi per la memorizzazione terziaria

La memoria terziaria consiste di mezzi rimovibili, di cui dischetti, CD-ROM e DVD sono gli esempi più comuni. Sul mercato sono disponibili anche molti altri prodotti, fra i quali dispositivi rimovibili che interagiscono con il computer tramite un'interfaccia USB, basati su memorie flash.

#### 12.9.1.1 Dischi rimovibili

I dischi rimovibili sono un tipo di memoria terziaria. I dischetti sono un esempio di dischi magnetici rimovibili, costituiti da un disco sottile e flessibile, ricoperto di materiale magnetico e racchiuso in un involucro protettivo di plastica. I comuni dischetti hanno una capacità di circa 1 MB, ma una tecnologia simile si usa per costruire dischi magnetici rimovibili della capacità di più di 1 GB. I dischi magnetici rimovibili possono funzionare a una velocità quasi pari a quella di un'unità a disco, anche se il rischio che la loro superficie sia danneggiata da graffi è maggiore.

I dischi magneto-ottici sono un altro tipo di dischi rimovibili: anche in questo caso i dati sono registrati su un disco ricoperto di materiale magnetico, ma la tecnologia impiegata per la registrazione è molto diversa da quella dei dischi magnetici. La testina di un disco magneto-ottico è sospesa a una distanza dalla superficie del disco molto maggiore rispetto alla testina di un disco magnetico, e il materiale magnetico è protetto da uno spesso strato di plastica o di vetro; di conseguenza, il disco è molto più resistente a eventuali collisioni della testina.

L'unità per i dischi magneto-ottici ha una bobina capace di produrre un campo magnetico, ma a temperature ordinarie il campo è troppo diffuso e debole per poter magnetizzare un bit sul disco; per rimediare a questo fatto la testina emette un raggio laser verso la superficie del disco, puntandolo sulla piccolissima area dove si vuole scrivere un bit. Il surriscaldamento dell'area provocato dal laser fa sì che essa divenga sensibile al campo magnetico, ed è in questo modo che un campo magnetico debole e diffuso riesce a registrare un minuscolo bit.

La testina di un disco magneto-ottico è troppo lontana dalla superficie del disco per poter leggere i dati rilevando i piccolissimi campi magnetici in modo analogo a quanto fa la testina di un disco magnetico. Perciò l'unità a disco legge i bit sfruttando una proprietà della luce laser detta **effetto Kerr**; quando un raggio laser è riflesso da un punto magnetizzato la sua polarizzazione è ruotata in senso orario o antiorario secondo l'orientazione del campo magnetico: per leggere i bit la testina rileva questa rotazione.

Un'altra categoria di dischi rimovibili è quella dei **dischi ottici**, i quali non sfruttano per niente il magnetismo, ma usano materiali speciali che la luce laser può alterare in modo da creare punti relativamente chiari o scuri. Un esempio di tecnologia per dischi ottici sono i **dischi a cambio di fase**, ricoperti da un materiale che può solidificare passando a uno stato cristallino o a uno stato amorfo. Lo stato cristallino è più trasparente, perciò un raggio laser è più luminoso quando attraversa il materiale a cambio di fase ed è riflesso dall'apposito strato. Le unità per dischi a cambio di fase impiegano laser capaci di emettere raggi a tre differenti livelli di potenza: bassa, per leggere i dati; media, per cancellare il disco fondendo e facendo solidificare il supporto di memorizzazione nello stato cristallino; alta, per scrivere nel disco fondendo e facendo solidificare il supporto nello stato amorfo. Gli esempi più comuni di questo tipo di tecnologia sono i dischi ottici riscrivibili CD-RW e DVD-RW.

Il tipo di dischi fin qui descritti si possono riutilizzare: per questo sono detti **dischi a lettura e scrittura**. Per contro, i **dischi monoscrivibili** o (*write once, read many times*, WORM) costituiscono una categoria distinta. Un vecchio modo di costruire un disco WORM è di inserire una pellicola d'alluminio tra due piatti di plastica o di vetro. Per scrivere un bit l'unità usa un raggio laser per praticare un piccolo foro nell'alluminio: poiché questo processo non è reversibile, si può scrivere una sola volta su un qualunque settore del disco. Sebbene sia possibile distruggere l'informazione contenuta in un disco WORM, ad esempio praticando fori dappertutto, è praticamente impossibile alterare i dati in esso contenuti, perché l'unica azione possibile è quella di aggiungere fori, ed è assai probabile che il codice ECC associato a ogni settore rilevi le modifiche. I dischi WORM sono considerati durevoli e affidabili: lo strato di metallo è protetto dalla copertura di vetro o plastica, e i campi magnetici non possono danneggiare la registrazione. Una più recente tecnologia di monoscrittura compie la registrazione su un pigmento polimerico invece che su uno strato d'alluminio: il pigmento forma dei punti assorbendo la luce del laser. Questa tecnologia s'impiega nei dischi ottici scrivibili CD-R e DVD-R.

I **dischi a sola lettura**, ad esempio i CD-ROM e i DVD, sono commercializzati con un contenuto preregistrato: fanno uso di una tecnologia simile a quella dei dischi WORM (sebbene in questo caso i fori siano stampati) e sono assai durevoli.

Generalmente i dischi rimovibili sono più lenti dei corrispondenti dischi fissi. Il processo di scrittura è più lento così come la rotazione e talvolta la ricerca.

### 12.9.1.2 Nastri

I nastri magnetici sono un altro tipo di memorizzazione terziaria. In linea generale un nastro può contenere più dati di un disco ottico o magnetico rimovibile. Le unità a nastro e quelle a disco hanno velocità di trasferimento simili ma, poiché richiedono operazioni di avanzamento rapido o di riavvolgimento che possono durare decine di secondi o addirittura interi minuti, l'accesso diretto è molto più lento per un nastro che per un disco.

Anche se un'unità a nastro è più costosa di un'unità a disco, una cartuccia a nastro è più economica di un disco magnetico della stessa capacità: i nastri magnetici sono quindi un mezzo conveniente qualora non si richieda la possibilità di rapidi accessi diretti. I nastri si usano comunemente per contenere copie di riserva dei dati presenti nei dischi, ma si usano

anche nei grandi centri di calcolo, dotati di supercalcolatori, per memorizzare le enormi quantità di dati che s'impiegano nella ricerca scientifica o per la gestione di grandi aziende.

Grandi stazioni di registrazione a nastri usano tipicamente meccanismi automatici per spostare i nastri dalle unità ad appositi contenitori in un archivio di nastri; questi meccanismi offrono ai calcolatori l'accesso automatico a un elevato numero di nastri.

Un archivio automatizzato riduce i costi totali della registrazione dei dati. Un file non immediatamente necessario residente in un disco si può **archiviare** in un nastro a un costo per gigabyte che può essere inferiore; quando il file si renderà necessario, il calcolatore potrà installarlo nuovamente nel disco. Un archivio automatizzato realizza un tipo di memorizzazione talvolta detto **quasi in linea**, perché si situa fra le alte prestazioni della memorizzazione **in linea** nei dischi magnetici e il basso costo di una memorizzazione **non in linea** in nastri archiviati in qualche deposito.

### 12.9.1.3 Tecnologie future

È possibile che in futuro altre tecnologie di memorizzazione acquisiscano importanza. A volte sono le vecchie tecnologie a essere utilizzate in nuovi modi, parallelamente ai cambiamenti economici e all'evoluzione tecnologica. Ad esempio, i **dischi a stato solido**, o SSD, stanno diventando sempre più importanti e diffusi. In breve, un SSD è un disco utilizzato come un disco rigido che, a seconda della tecnologia di memorizzazione utilizzata, può essere volatile o non volatile. La tecnologia di memorizzazione incide inoltre sulle prestazioni. I dischi a stato solido non volatili presentano le stesse caratteristiche dei dischi rigidi tradizionali, ma possono essere più affidabili, perché non hanno parti mobili, e più veloci, perché non hanno tempo di ricerca né di latenza. Richiedono inoltre meno energia. Rispetto alle unità a disco tradizionali i dischi a stato solido hanno però un costo al megabyte più elevato, una minore capacità dei dischi rigidi più grandi e una durata di vita a volte più breve; per questi motivi, il loro utilizzo è limitato. Per fare un esempio, i dischi a stato solido sono impiegati nelle batterie di memorizzazione per contenere metadati che richiedono prestazioni elevate, come il log di un file system con annotazioni delle modifiche. I dischi a stato solido vengono anche montati sui computer portatili per renderli più leggeri, più veloci e più efficienti in termini energetici.

Un'altra promettente tecnologia per la memorizzazione di massa è la **memoria olografica**, che sfrutta la luce laser per registrare fotografie olografiche su supporti dedicati.

Un **ologramma** può essere pensato come una matrice tridimensionale di pixel da 1 bit ciascuno, i cui valori 0 e 1 rappresentano il nero e il bianco, rispettivamente. Poiché è possibile trasferire tutti i pixel tramite un solo fascio di luce laser, la velocità di trasmissione è altissima. In futuro, grazie ai progressi della ricerca, la memoria olografica potrebbe diventare una concreta opzione commerciale.

Un'altra tecnologia di memorizzazione oggetto di attive ricerche si fonda sui **sistemi meccanici microelettronici** (*microelectronic mechanical systems*, MEMS). L'idea consiste nell'applicare le tecnologie che s'impiegano nella fabbricazione dei circuiti integrati per costruire piccole macchine di memorizzazione. Una proposta prevede la fabbricazione di una matrice di 10.000 minuscole testine di disco, con un centimetro quadrato di materiale magnetico di registrazione sospeso sopra tale matrice. Quando si muove il materiale di registrazione lungo le testine, ognuna di loro accede alla propria traccia lineare di dati presente sul materiale magnetico. Il materiale di registrazione si può traslare leggermente in modo laterale per consentire a tutte le testine di accedere alla loro traccia successiva. Sebbene resti da vedere se questa tecnologia possa avere successo, essa può offrire sistemi di memorizzazione non volatile più rapidi dei dischi magnetici e più economici delle memorie a semiconduttori DRAM.

Indipendentemente dal fatto che il supporto di memorizzazione in uso sia un disco magnetico rimovibile, un DVD, o un nastro magnetico, il sistema operativo deve offrire una serie di funzioni affinché i mezzi rimovibili siano utilizzabili per contenere dati: questo argomento è trattato nel Paragrafo 12.9.2.

## 12.9.2 Compiti del sistema operativo

Due tra gli obiettivi primari di un sistema operativo sono la gestione dei dispositivi fisici e la presentazione di una macchina virtuale alle applicazioni. Il sistema operativo realizza due astrazioni concernenti i dischi: una è il dispositivo a basso livello, un semplice array di blocchi di dati; l'altra è il file system. Se il file system è relativo a un disco magnetico il sistema operativo accoda e organizza le richieste provenienti da diverse applicazioni. Nel seguito si espone il comportamento del sistema operativo nel trattare i supporti di memorizzazione rimovibili.

### 12.9.2.1 Interfaccia per le applicazioni

La maggior parte dei sistemi operativi gestisce i dischi rimovibili quasi come i dischi fissi. Quando s'inserisce un nuovo disco nella relativa unità, esso deve essere formattato, quindi si crea sul disco rimovibile un file system vuoto che si usa proprio come il file system di un'ordinaria unità a disco.

La gestione dei nastri, invece, è spesso differente; il sistema operativo di solito presenta un nastro come un supporto di memorizzazione a basso livello. Un'applicazione non apre un file presente nel nastro: apre l'intera unità nastro come dispositivo a basso livello. In questo caso, di solito, l'unità a nastro si riserva per l'uso esclusivo da parte di tale applicazione fino a che essa termina o chiude il dispositivo. L'esclusività è ragionevole perché l'accesso diretto ai dati presenti in un nastro può richiedere decine di secondi o persino qualche minuto, sicché intercalare gli accessi diretti a un nastro determinerebbe probabilmente enormi tempi d'accesso per un ridottissimo lavoro utile.

Quando un'unità a nastro è presentata come dispositivo a basso livello, il sistema operativo non fornisce i servizi del file system: è l'applicazione che deve decidere come usare l'array di blocchi. Un programma che crea una copia di riserva di un disco su un nastro potrebbe ad esempio scrivere una lista dei nomi e delle dimensioni dei file all'inizio del nastro, e poi copiare i dati dei file sul nastro in quell'ordine.

È facile rendersi conto dei problemi che possono sorgere quando si usi l'unità a nastro in questo modo: visto che ogni applicazione stabilisce i propri criteri di organizzazione del nastro, un nastro contenente dati può essere generalmente usato solo dal programma che lo ha creato. Se anche si sapesse, ad esempio, che un nastro con copie di riserva di file contiene una lista dei nomi e delle dimensioni dei file seguita dai dati dei file in quell'ordine, ci sarebbero comunque difficoltà nell'uso del nastro: non è nota l'esatta maniera in cui i nomi dei file sono registrati, né se le dimensioni siano espresse nel codice binario o ASCII, o ancora se i file siano scritti uno per blocco o invece concatenati assieme in una lunghissima sequenza di byte. Non è nota neanche la dimensione dei blocchi del nastro, perché questo è un parametro che si può fissare indipendentemente per ogni blocco scritto.

Le operazioni fondamentali relative a un'unità a disco sono `read()`, `write()` e `seek()`; per le unità a nastro s'impiega un insieme di operazioni fondamentali diverso. Anziché usare l'operazione `seek()`, si usa l'operazione `locate()`. Si tratta di un'operazione più precisa dell'operazione `seek()` per il disco, perché posiziona il nastro in corrispondenza di uno specifico blocco logico, e non di un'intera traccia: localizzare il blocco 0 equivale a riavvolgere il nastro.

Nella maggior parte delle unità nastro è possibile localizzare qualunque blocco sia stato scritto in un nastro, ma se il nastro non è ancora completamente pieno non si può eseguire un'operazione `locate()` nell'area vuota oltre l'area registrata; ciò poiché la maggior parte delle unità a nastro gestisce lo spazio fisico diversamente dalle unità a disco. I settori di un disco hanno una dimensione fissa, e si deve usare il processo di formattazione per assegnare la posizione definitiva ai settori vuoti prima che possano contenere qualunque informazione. La maggior parte delle unità a nastro ha una dimensione dei blocchi variabile, e la dimensione di ogni blocco si determina al momento della scrittura del blocco in questione. Se durante la scrittura s'incontra una regione difettosa del nastro, la si salta e si riscrive il blocco. Tutto ciò spiega perché non sia possibile compiere un'operazione `locate()` nella regione di nastro vuota presente oltre l'area già registrata: le posizioni e la numerazione dei blocchi logici non sono ancora state determinate.

Nella maggior parte dei casi le unità a nastro dispongono di un'operazione `read_position()` che riporta il numero del blocco logico in corrispondenza del quale si trova la testina. Molte unità a nastro hanno anche un'operazione `space()` per gli spostamenti relativi: l'operazione `space(-2)`, ad esempio, riavvolge il nastro di due blocchi logici.

In molti tipi di unità a nastro la scrittura di un blocco produce come effetto collaterale la cancellazione logica di tutto ciò che si trova oltre la posizione della scrittura. Ciò significa che nella maggior parte dei casi le unità a nastro sono dispositivi a solo accodamento di dati (*append-only devices*); in altre parole, l'aggiornamento di un blocco posto in mezzo al nastro comporta la cancellazione di tutto ciò che segue tale blocco. L'unità a nastro realizza l'accodamento scrivendo un simbolo di fine nastro (*end of tape*, EOT) dopo l'ultimo blocco registrato: l'unità rifiuta di compiere un'operazione `locate()` oltre il simbolo EOT, ma può localizzare l'EOT stesso e poi cominciare a scrivere. Quest'azione ha l'effetto di sovrascrivere il simbolo EOT e di accodarne uno nuovo alla fine dei blocchi appena scritti.

In linea di principio si potrebbero realizzare file system per i nastri ma, poiché le unità a nastro sono dispositivi a solo accodamento, molte strutture dati e algoritmi sarebbero diversi da quelli che si usano per i dischi.

### 12.9.2.2 Nomi dei file

Un'altra questione che il sistema operativo deve affrontare è l'assegnazione dei nomi ai file residenti nei mezzi rimovibili. Nel caso di un disco fisso ciò non è difficile: nei PC i nomi dei file consistono di una lettera rappresentante un'unità seguita da un nome di percorso; nel sistema operativo UNIX i nomi dei file non contengono riferimenti alle unità, ma la tabella di montaggio permette al sistema operativo di identificare l'unità contenente ciascun file. Se però il disco è rimovibile, il fatto che un'unità abbia ospitato un certo disco non facilita la ricerca del file. Se ogni disco rimovibile avesse un numero di serie diverso, un file residente in un'unità potrebbe presentare come prefisso il proprio numero di serie, ma in questo caso sarebbe necessario usare circa 12 cifre per evitare che due dischi possano avere lo stesso numero di serie: non è pensabile che gli utenti ricordino numeri di 12 cifre per identificare i loro file.

Le cose si complicano ulteriormente nel caso s'intenda scrivere dati su un supporto rimovibile in un certo calcolatore e poi riutilizzare lo stesso mezzo in un altro calcolatore: se entrambi i calcolatori sono dello stesso tipo e hanno lo stesso tipo di unità, l'unico problema è quello di conoscere i contenuti e l'organizzazione dei dati contenuti nel mezzo in questione; ma se i calcolatori o le unità sono di diverso tipo possono sorgere molte altre difficoltà. Anche se le unità fossero compatibili, calcolatori diversi potrebbero memorizzare i dati secondo ordini diversi, o usare codifiche diverse per i numeri binari e persino per le lettere (ASCII nei PC, EBCDIC nei mainframe).

In genere gli attuali sistemi operativi lasciano irrisolto il problema dei nomi per i mezzi rimovibili, confidando nel fatto che le applicazioni o gli utenti forniranno una chiave di lettura e di interpretazione dei dati. Per fortuna alcuni tipi di mezzi rimovibili sono così ben standardizzati da essere usati allo stesso modo da tutti i calcolatori. Un esempio è dato dai CD: i CD musicali sono registrati in un formato universalmente noto e leggibile da ogni riproduttore di CD. I CD di dati sono disponibili in pochi formati diversi, ed è normale che sia l'unità di lettura sia il sistema operativo siano in grado di gestirli. Anche i formati dei DVD sono ben standardizzati.

### 12.9.2.3 Gestione gerarchica della memoria

Un juke-box automatizzato permette a un calcolatore di cambiare un nastro o un disco rimovibile senza l'intervento di un utente. Due tra le principali applicazioni di questa tecnica sono relative alla realizzazione di copie di riserva e ai sistemi di gestione gerarchica della memoria. L'uso dei juke-box per la creazione di copie di riserva è molto semplice: quando un nastro o un disco sono pieni, il calcolatore richiede al juke-box di passare al successivo. Alcuni juke-box contengono decine di unità e migliaia di nastri, con bracci automatizzati che spostano i nastri nelle unità.

Un sistema di gestione gerarchica della memoria estende la gerarchia di memorizzazione oltre la memoria centrale e secondaria (cioè, i dischi magnetici) comprendendo la memoria terziaria; quest'ultima è di solito costituita di un juke-box di nastri o di dischi rimovibili. Si tratta del livello di memoria meno costoso e più capiente, ma probabilmente anche più lento.

Sebbene il sistema della memoria virtuale si possa estendere senza difficoltà alla memoria terziaria, in pratica ciò avviene raramente; infatti recuperare dati per mezzo di un juke-box può richiedere decine di secondi o addirittura minuti: attese così lunghe sono inconciliabili con le tecniche di paginazione su richiesta e con altri modi d'uso della memoria virtuale.

La tecnica più comune per estendere la gerarchia di memorizzazione fino alla memoria terziaria consiste nell'ampliare il file system. I file piccoli e frequentemente usati rimangono nei dischi magnetici, mentre i file vecchi, ingombranti e raramente necessari si archiviano nel juke-box. In alcuni sistemi per l'archiviazione dei file gli elementi di directory corrispondenti ai nomi dei file continuano a comparire nelle directory anche dopo l'archiviazione, ma i contenuti dei file non occupano più spazio nella memoria secondaria. Quando un'applicazione tenta di aprire un file archiviato, la chiamata di sistema `open()` rimane sospesa finché i contenuti del file possono essere reinstallati dalla memoria terziaria; una volta nuovamente disponibili nei dischi magnetici, la `open()` restituisce il controllo all'applicazione, e quest'ultima può ora accedere ai contenuti del file.

Al giorno d'oggi, la **gestione gerarchica della memoria secondaria** (*hierarchical storage management*, HSM) si applica solitamente a sistemi con grandi quantità di dati usati di rado, sporadicamente o periodicamente. Le ricerche correnti in questo campo tentano di estendere HSM con una piena **gestione del ciclo di vita delle informazioni** (*information life-cycle management*, ILM). I dati si spostano dai dischi ai nastri e di nuovo ai dischi, a seconda delle necessità, e sono cancellati secondo una scaletta o in accordo con una certa strategia. Alcuni siti, ad esempio, conservano i messaggi di posta elettronica per sette anni, ma pretendono l'assicurazione che dopo questo periodo siano distrutti. Allo scadere dei sette anni, i dati rilevanti potrebbero risiedere sui dischi, sui nastri HSM o sui nastri per le copie di riserva. Una buona ILM centralizza le informazioni sulla collocazione dei dati. Ciò permette di applicare la stessa strategia di gestione a specifici insiemi di dati, indipendentemente dalla loro collocazione fisica.

## 12.9.3 Prestazioni

Così come avviene per ogni componente del sistema operativo, i tre aspetti più importanti riguardanti le prestazioni della memorizzazione terziaria sono velocità, affidabilità e costi.

### 12.9.3.1 Velocità

La velocità della memoria terziaria è definita da due fattori: ampiezza di banda e latenza. La prima si misura in byte al secondo; in particolare, l'**ampiezza di banda sostenuta** è la velocità media di trasferimento nel caso di una rilevante quantità di dati – in altre parole, il numero di byte diviso il tempo di trasferimento; l'**ampiezza di banda effettiva** è invece il numero di byte trasferiti rapportato al tempo di I/O totale, inclusi il tempo richiesto da una `seek()` o una `locate()` e l'attesa eventualmente dovuta a cambi di dischi o nastri eseguiti dal juke-box. Essenzialmente, l'ampiezza di banda sostenuta è la velocità di trasferimento nel momento in cui i dati stanno effettivamente fluendo, mentre l'ampiezza di banda effettiva è la velocità di trasferimento complessiva fornita dall'unità. Con l'espressione *ampiezza di banda di un'unità* s'intende generalmente l'ampiezza di banda sostenuta.

L'ampiezza di banda per le unità a dischi rimovibili varia da pochi megabyte al secondo nei tipi più lenti, a più di 40 MB al secondo nei più veloci. Le unità a nastro hanno un'ampiezza di banda simile, da pochi megabyte fino a 30 MB al secondo.

Il secondo fattore è la latenza d'accesso. Rispetto a questo parametro i dischi sono molto più veloci dei nastri: la memorizzazione nei dischi è essenzialmente bidimensionale – tutti i bit sono, per così dire, all'aperto; un accesso al disco si compie semplicemente spostando il braccio verso il cilindro selezionato e aspettando che il settore interessato ruoti sotto la testina, il che può avvenire in meno di 5 millisecondi. Per contro, la memorizzazione nei nastri è tridimensionale: a ogni dato istante solo una piccola parte del nastro è accessibile alla testina, mentre il resto dei bit è sepolto sotto centinaia o migliaia di strati di nastro avvolto in una bobina. Un accesso diretto a un nastro richiede lo svolgimento o il riavvolgimento della bobina finché il blocco richiesto raggiunge la testina, cosa che può richiedere decine o centinaia di secondi. Si può quindi affermare in linea generale che l'accesso diretto a un nastro è oltre mille volte più lento dell'accesso diretto a un disco.

Se in tutto ciò è coinvolto anche un juke-box la latenza d'accesso può crescere notevolmente: per cambiare un disco rimovibile l'unità deve fermare la rotazione del motore, il braccio automatizzato deve scambiare i dischi, e l'unità deve avviare la rotazione del nuovo disco. Questa operazione richiede parecchi secondi, un tempo pari a circa cento volte il tempo medio d'accesso diretto a un disco singolo: lo scambio di dischi in un juke-box comporta una penalizzazione relativamente alta delle prestazioni.

Il tempo impiegato da un braccio automatizzato nel caso dei nastri è circa lo stesso che nel caso dei dischi; in genere, però, prima di poter essere estratto dall'unità un nastro deve essere completamente riavvolto, e quest'operazione può richiedere anche 4 minuti. Inoltre, dopo che un nuovo nastro è stato caricato, l'unità può aver bisogno di molti secondi per calibrarsi rispetto al nuovo nastro e prepararsi all'I/O. Sebbene un lento juke-box per nastri possa richiedere 1 o 2 minuti per scambiare i nastri, questo tempo non è sproporzionalmente lungo rispetto al tempo necessario per l'accesso diretto a un singolo nastro.

Generalizzando, quindi, si può dire che l'accesso diretto a un disco in un juke-box ha una latenza dell'ordine delle decine di secondi, mentre nel caso dei nastri in un juke-box la latenza è dell'ordine delle centinaia di secondi; lo scambio dei dischi è oneroso, mentre non lo è quello dei nastri. Si tratta ovviamente di un discorso di carattere generale: alcuni costosi juke-box per nastri sono capaci di riavvolgere ed estrarre un nastro, caricarne uno nuovo e posizionarlo a uno specifico punto impiegando complessivamente meno di 30 secondi.

Se si considerano soltanto le prestazioni delle unità di lettura e scrittura di un juke-box, i tempi di latenza e l'ampiezza di banda appaiono ragionevoli; non appena si concentra l'attenzione sui mezzi rimovibili si riscontra però un tremendo collo di bottiglia per le prestazioni. Si consideri in primo luogo l'ampiezza di banda: il rapporto fra l'ampiezza della banda e la capacità di memorizzazione di una biblioteca automatizzata è molto più sfavorevole di quello di un disco fisso. La lettura di tutti i dati contenuti in un disco di grandi dimensioni potrebbe richiedere circa un'ora, ma leggere tutti i dati memorizzati in un ingombrante archivio di nastri potrebbe richiedere anni. Per ciò che riguarda la latenza d'accesso la situazione è quasi altrettanto grama: se 100 richieste sono in coda per un'unità a disco fisso, il tempo d'attesa medio sarà di circa 1 secondo, ma se 100 richieste sono in coda per un archivio di nastri, il tempo d'attesa medio potrebbe essere più di 1 ora. La convenienza economica della memoria terziaria è dovuta alla possibilità di usare molte cartucce (a disco o a nastro) a basso costo con poche costose unità di lettura e scrittura. Un archivio di supporti rimovibili, però, è soprattutto adatto alla registrazione di dati usati raramente, perché il numero di richieste di I/O soddisfacibili per ogni ora d'uso di un tale archivio è relativamente basso.

### 12.9.3.2 Affidabilità

Nonostante si tenda a identificare il concetto di *buone prestazioni* con quello di *alta velocità*, un altro importante aspetto delle prestazioni è l'*affidabilità*: se non si riuscisse a leggere dati a causa di un guasto dell'unità o del supporto di memorizzazione, il tempo d'accesso sarebbe infinitamente lungo e l'ampiezza di banda infinitamente bassa. È quindi importante analizzare l'affidabilità dei mezzi rimovibili.

I dischi magnetici rimovibili sono meno affidabili dei dischi fissi, perché è più probabile che siano esposti a condizioni ambientali dannose come polvere, sbalzi di temperatura o umidità e forze meccaniche come urti o piegature. I dischi ottici sono considerati molto affidabili, perché lo strato che memorizza le informazioni è protetto da uno strato trasparente di plastica o vetro. L'affidabilità dei nastri magnetici è molto variabile poiché dipende dal tipo di unità di lettura e scrittura: alcune unità poco costose consumano un nastro dopo averlo usato poche decine di volte, mentre altre sono così delicate da permettere il reimpiego del nastro milioni di volte. Rispetto alla testina di un'unità a disco, la testina di un'unità a nastro è meno affidabile: la prima è sospesa sopra il disco, mentre la seconda è a stretto contatto col nastro, e l'attrito conseguente può rovinarla dopo un elevato numero di ore d'uso.

Riassumendo si può dire che le unità a disco fisso sono più affidabili delle unità a nastri o a dischi rimovibili, e che i dischi ottici sono probabilmente più affidabili dei dischi e dei nastri magnetici. Anche le unità a disco fisso hanno punti deboli: la collisione della testina col disco in genere distrugge i dati, mentre il guasto di un'unità a nastro o di un'unità a dischi ottici lascia spesso intatto il supporto di memorizzazione in uso al momento del guasto.

### 12.9.3.3 Costi

Il costo della memoria è un altro fattore importante: l'esempio seguente mostra concreteamente come i mezzi rimovibili possano ridurre i costi totali di memorizzazione. Si supponga che un'unità a disco di  $x$  GB costi 200 dollari; di questa cifra, 190 dollari sono dovuti al controllore, al motore e al contenitore esterno, e 10 dollari ai piatti magnetici. Il costo della memoria fornita da questa unità è quindi di  $200/x$  dollari al gigabyte. Si supponga ora di poter incapsulare i piatti magnetici in un disco rimovibile: il costo complessivo di un'unità e di 10 dischi è allora di  $190 + 100$  dollari, e la capacità di memoria totale è di  $10x$  GB, cosicché il costo per gigabyte scende a  $29/x$  dollari al gigabyte. Sebbene il costo di produzione

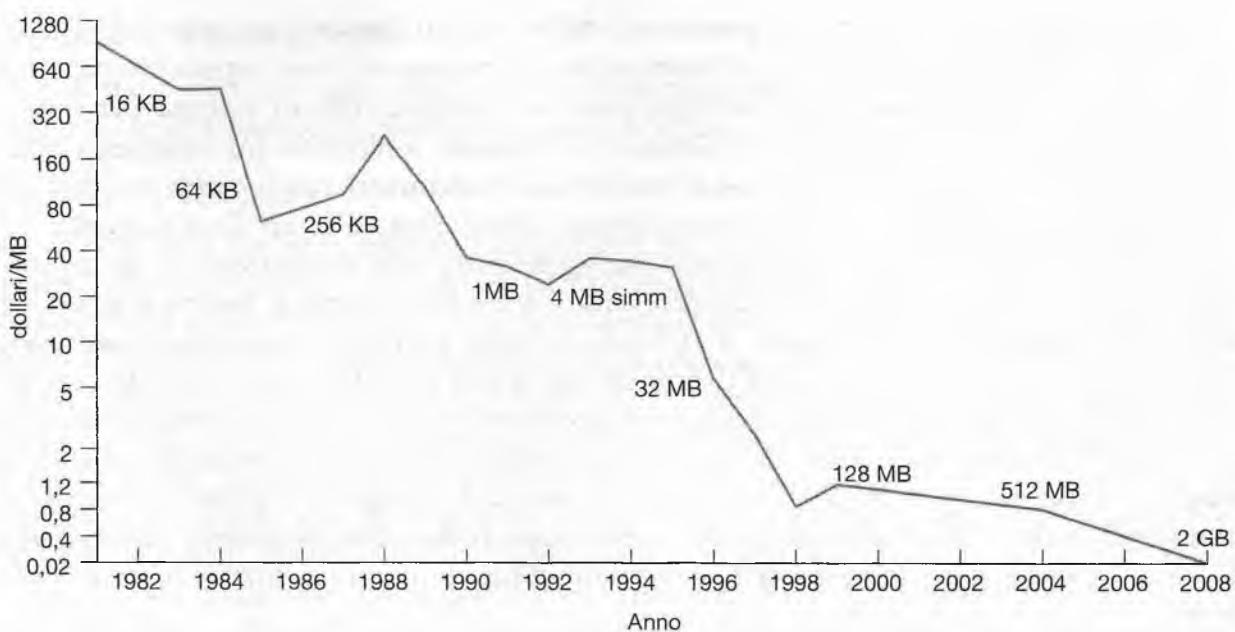


Figura 12.15 Prezzo al MB della memoria DRAM, dal 1981 al 2008.

delle unità a dischi rimovibili sia un po' più alto, la maggior spesa per un'unità è bilanciata dal basso costo di molte cartucce rimovibili, quindi il costo per gigabyte della memoria rimovibile può essere ben inferiore a quello delle unità a disco fisso.

Nelle Figure 12.15, 12.16 e 12.17 sono mostrate rispettivamente le tendenze dei prezzi al megabyte della memoria DRAM, delle unità a disco e delle unità a nastro. I prezzi riportati nei grafici sono i più bassi riscontrati fra tutte le inserzioni pubblicitarie apparse in varie riviste di settore e sul Web alla fine di ogni anno. Essi riflettono le caratteristiche del mercato dei piccoli calcolatori, oggetto dell'interesse dei lettori, i cui prezzi sono più bassi rispetto a mini-calcolatori e mainframe. Nel caso dei nastri, il prezzo si riferisce a un'unità dotata di un solo nastro; poiché il costo di un nastro è in genere una piccola frazione del costo di un'unità, il

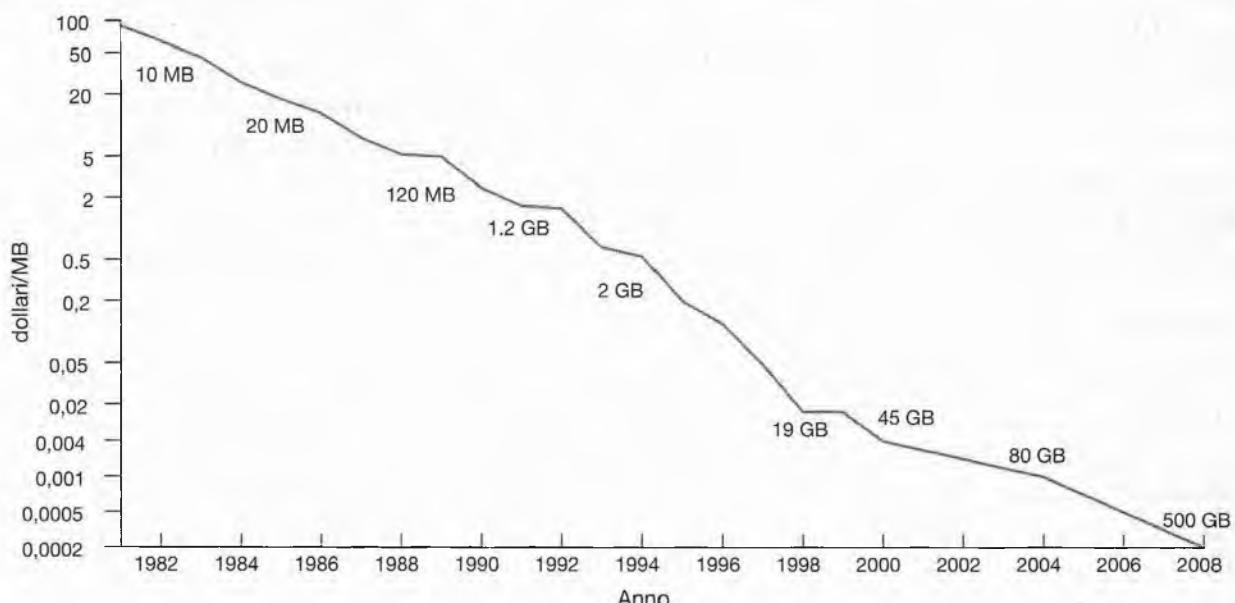
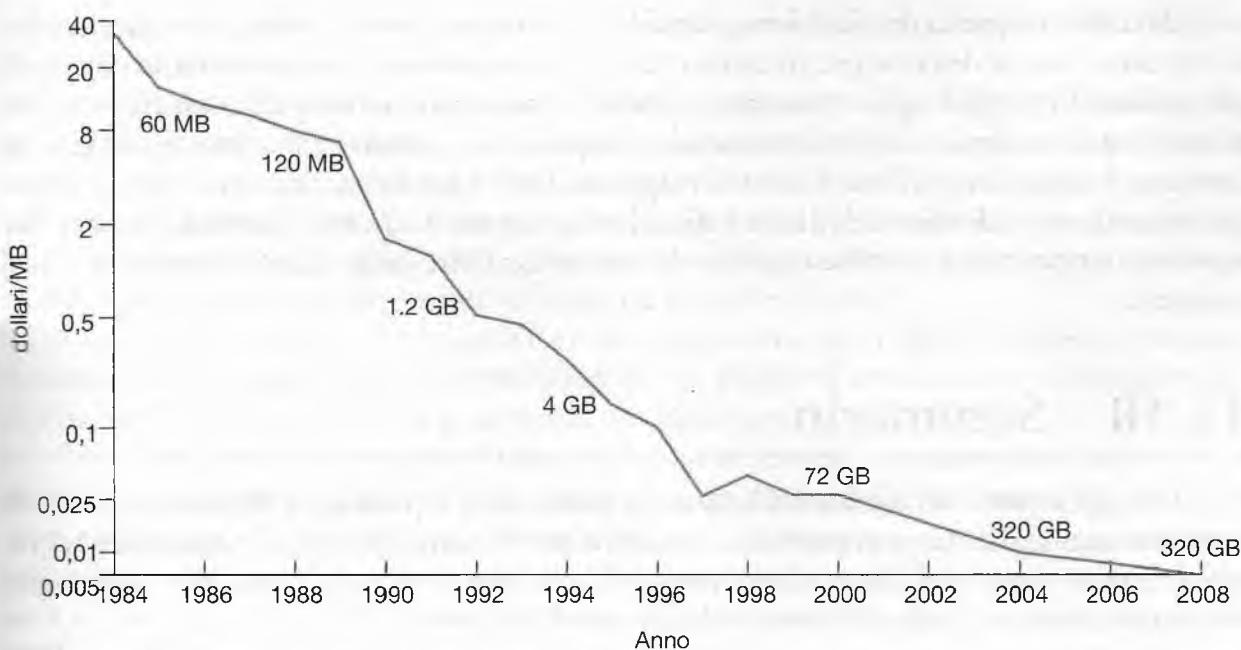


Figura 12.16 Prezzo al MB delle unità a disco magnetico, dal 1981 al 2008.



**Figura 12.17** Prezzo al MB delle unità a nastro, dal 1984 al 2008.

costo totale della memorizzazione nei nastri diviene progressivamente molto più basso in seguito all'acquisto di nuovi nastri da usare in una stessa unità. Infatti in un grande archivio automatizzato contenente migliaia di nastri, il costo di memorizzazione è dominato dal costo dei nastri. Nel corso del 2004 il costo al GB dei nastri si aggirava intorno ai 40 dollari.

Come mostra la Figura 12.15, il costo della DRAM ha subito notevoli fluttuazioni: fra il 1981 e il 2004 si possono notare tre crolli del prezzo (attorno al 1981, al 1989 e al 1996) dovuti a una sovrapproduzione che causò la saturazione del mercato; si notano anche due periodi (attorno al 1987 e al 1993) durante i quali la scarsità d'offerta ha provocato un notevole aumento dei prezzi. Per quel che riguarda le unità a disco, (Figura 12.16), invece, la diminuzione dei prezzi è stata molto più regolare, anche se dal 1992 la corsa al ribasso ha avuto un'accelerazione. I prezzi delle unità a nastro sono scesi con regolarità fino al 1997 (Figura 12.17). In seguito il prezzo al GB delle unità a nastro economiche ha cessato di precipitare, sebbene i prezzi dei prodotti di tecnologie di medio livello, come i DAT/DDS, siano continuati a scendere, e attualmente siano vicini alle unità economiche. I prezzi delle unità a nastro sono indicati a partire dal 1984, perché, come già detto, la stampa di settore si occupa del mercato dei piccoli calcolatori, dei quali le unità a nastro non erano un accessorio comune prima di questa data.

Da questi grafici si nota quanto il prezzo della memorizzazione su disco sia crollato negli ultimi vent'anni; confrontando i grafici si deduce anche quanto, rispetto a quello delle memorie DRAM e dei nastri, si sia notevolmente abbassato.

Il prezzo al megabyte dei dischi magnetici si è ridotto di oltre quattro ordini di grandezza dal 1981 al 2004, mentre il prezzo corrispondente per gli elementi di memoria centrale è sceso di tre ordini di grandezza. La memoria centrale è attualmente 100 volte più costosa dei dischi.

Il prezzo al megabyte delle unità a dischi magnetici si avvicina al prezzo dei nastri magnetici, escludendo il costo delle unità a nastro. Ne consegue che gli archivi di nastri medi e piccoli hanno un costo di memorizzazione dei dati maggiore di un sistema a dischi di corrispondente capacità.

Il crollo dei prezzi dei dischi magnetici ha reso in gran parte obsoleta la memorizzazione terziaria: non si dispone più di alcuna tecnologia di memorizzazione terziaria che sia di più ordini di grandezza più economica dei dischi magnetici. Sembra che una ripresa della memorizzazione terziaria debba attendere un importante passo avanti in una tecnologia innovativa. Nel frattempo, l'uso dei nastri magnetici resterà per lo più limitato a scopi come la creazione di copie di riserva di dischi e di archivi composti di enormi quantità di nastri, che superano largamente l'effettiva capacità di memorizzazione delle grandi batterie di dischi magnetici.

## 12.10 Sommario

Per la maggior parte dei calcolatori le unità a dischi sono il principale dispositivo di I/O di memoria secondaria. La gran parte dei dispositivi per la memorizzazione secondaria usa i dischi o i nastri magnetici. Le moderne unità a disco sono strutturate come un grande array monodimensionale di blocchi logici, solitamente di 512 byte.

I dischi possono essere collegati al computer in due modi: (1) tramite le porte per l'I/O locali della macchina o (2) tramite connessioni di rete.

Le richieste di I/O sui dischi sono generate sia dal file system sia dai sistemi di memoria virtuale, e ognuna di esse specifica l'indirizzo cui fare riferimento nel disco sotto forma di numero di un blocco logico. Gli algoritmi di scheduling per i dischi possono aumentare l'ampiezza di banda, e ridurre il tempo di risposta medio e la variabilità del tempo di risposta. Algoritmi come l'SSTF, SCAN, C-SCAN, LOOK e C-LOOK sono progettati per realizzare questi miglioramenti tramite criteri di ordinamento della coda di richieste di operazioni sui dischi.

Le prestazioni possono essere influenzate negativamente dalla frammentazione esterna. Alcuni sistemi includono programmi rivolti a questo problema che esaminano l'intero file system alla ricerca di file frammentati, e spostano i blocchi tentando di ridurne la frammentazione. La deframmentazione di un file system può portare a un notevole incremento delle prestazioni, almeno quando esso si trova in avanzato stato di frammentazione; tuttavia, le prestazioni del sistema durante tale operazione si riducono. File system raffinati, come il Fast File System di UNIX, adottano molte tecniche per controllare la frammentazione durante l'allocazione dello spazio, rendendo superflua la riorganizzazione del disco.

Il sistema operativo gestisce i blocchi di un disco: innanzitutto si deve formattare fisicamente il disco per creare i settori direttamente sui suoi piatti – i dischi nuovi sono generalmente venduti già formattati; in seguito, il disco può essere diviso in partizioni, si può creare il file system, e si possono assegnare i blocchi d'avviamento che conterranno il programma d'avviamento del sistema; infine, quando un blocco diviene difettoso, il sistema deve essere in grado di isolarlo o di sostituirlo logicamente con un blocco di riserva.

Poiché un'area d'avvicendamento efficiente è essenziale per un sistema dalle buone prestazioni, molti sistemi usano un accesso di basso livello per l'I/O di paginazione. Certi sistemi riservano all'area d'avvicendamento una partizione di basso livello, altri impiegano un ordinario file all'interno del file system. Altri sistemi lasciano la scelta fra queste due possibilità all'utente o all'amministratore di sistema.

A causa della quantità di spazio per la registrazione dei dati richiesta nei grandi sistemi, i dischi sono spesso resi ridondanti tramite algoritmi RAID. Questi algoritmi permettono l'uso di più dischi per una data operazione, e consentono la prosecuzione del funzionamento del sistema e anche il ripristino automatico dei dati a fronte del guasto di un disco.

Gli algoritmi RAID sono classificati in livelli che offrono diverse combinazioni di affidabilità ed elevate velocità di trasferimento.

Lo schema di registrazione con scrittura anticipata (*write-ahead logging*) richiede la disponibilità di spazio di memorizzazione stabile, che va implementato replicando le informazioni necessarie su memorie non volatili multiple (di solito dischi) i cui guasti siano indipendenti gli uni dagli altri. È anche necessario aggiornare le informazioni in modo control-lato, sì da assicurare la possibilità di recuperare i dati stabili a seguito di qualsivoglia malfunzionamento durante il trasferimento o il recupero dei dati.

La memoria terziaria è realizzata da unità a nastro o a disco capaci di operare con mezzi rimovibili. Le tecnologie disponibili sono molte; alcune di esse sono i nastri magnetici, i dischi rimovibili magnetici e magneto-ottici e i dischi ottici.

Nel caso dei dischi rimovibili il sistema operativo fornisce in genere tutti i servizi di un file system, compresa la gestione dello spazio e lo scheduling delle richieste di I/O. Per molti sistemi operativi il nome di un file residente in una cartuccia rimovibile è una combinazione del nome dell'unità e del nome di un file contenuto in quell'unità; questa convenzione è più semplice, ma anche potenzialmente più ambigua, dell'uso di un numero di serie per l'identificazione di una specifica cartuccia.

Nel caso dei nastri il sistema operativo in genere fornisce semplicemente un'interfaccia a basso livello. Molti sistemi operativi, inoltre, non incorporano metodi specifici per la gestione dei juke-box: questi ultimi devono essere controllati da un driver di dispositivo apposito o da un'applicazione che si occupa della creazione delle copie di riserva o della HSM.

Tre importanti fattori delle prestazioni sono l'ampiezza di banda, la latenza e l'affidabilità. Esistono unità a dischi e nastri con un'ampia gamma di diverse ampiezze di banda, ma la latenza d'accesso dei nastri è generalmente molto più elevata di quella dei dischi. Anche lo scambio di cartucce eseguito da un juke-box è relativamente lento; visto poi che per un juke-box il rapporto fra unità e cartucce è basso, la lettura di una gran parte dei dati archiviati può richiedere un tempo molto lungo. I dischi ottici sono in generale più affidabili di quelli magnetici, perché nei primi lo strato fisico contenente le informazioni è protetto da due strati di materiale trasparente, mentre nei secondi il materiale magnetico è maggiormente esposto a eventuali danni. Infine, negli ultimi vent'anni il costo della memoria è diminuito sensibilmente, in particolare per quanto riguarda la memorizzazione su disco.

## Esercizi pratici

- 12.1 Lo scheduling del disco, fatta eccezione per l'FCFS, è utile in un ambiente con un solo utente? Argomentate.
- 12.2 Spiegate perché l'SSTF tende a favorire i cilindri centrali rispetto a quelli più interni e più esterni.
- 12.3 Perché la latenza di rotazione non viene solitamente presa in considerazione nello scheduling del disco? Come potreste modificare lo scheduling SSTF, lo SCAN e quello C-SCAN per includervi l'ottimizzazione della latenza?
- 12.4 In che modo l'utilizzo di un disco RAM potrebbe condizionare la vostra scelta di un algoritmo di scheduling del disco? Quali fattori dovrebbero essere considerati? Gli stessi fattori sono applicabili anche allo scheduling dell'unità a disco, dato che il file system memorizza blocchi usati recentemente in una buffer cache nella memoria centrale?
- 12.5 Perché è importante bilanciare gli I/O del file system tra i dischi e i controllori su un sistema in un ambiente multitasking?

- 12.6 Quali tradeoff implica la rilettura delle pagine di codice da un file system rispetto all'utilizzo dell'area di avvicendamento per memorizzarli?
- 12.7 Esiste un modo per realizzare una memorizzazione delle informazioni veramente stabile? Argomentate.
- 12.8 Il termine "Fast Wide SCSI-II" denota un bus SCSI che opera a una velocità di dati di 20 megabyte al secondo mentre sposta pacchetti di byte tra un host e un dispositivo. Supponete che una unità a disco Fast Wide SCSI-II giri a 7200 RPM, abbia una dimensione dei settori di 512 byte e contenga 160 settori per traccia.
- Stimate la velocità di trasferimento sostenuta da questa unità a disco in megabyte al secondo.
  - Supponete che l'unità abbia 7000 cilindri, 20 tracce per cilindro, un tempo di spostamento della testina (da un piatto all'altro) di 0,5 millisecondi e un tempo di ricerca del cilindro adiacente di 2 millisecondi. Utilizzate questa informazione per dare una stima accurata della velocità di trasferimento sostenuta per un trasferimento di grandi dimensioni.
  - Supponete che il tempo di ricerca medio per l'unità sia di 8 millisecondi. Stimate le operazioni di I/O al secondo e la velocità di trasferimento effettiva per un carico di lavoro ad accesso diretto che legge settori individuali distribuiti sul disco.
  - Calcolate il numero di operazioni di I/O ad accesso diretto al secondo e la velocità di trasferimento per I/O con dimensioni di 4, 8 e 64 kilobyte.
  - Se vi sono richieste multiple in coda, un algoritmo di scheduling come quello per scansione dovrebbe essere in grado di ridurre la distanza media di ricerca. Supponete che un carico di lavoro ad accesso diretto stia leggendo pagine di 8 kilobyte, la lunghezza media della coda sia 10, e l'algoritmo di scheduling riduca il tempo di ricerca medio a 3 millisecondi. Calcolate ora il numero di operazioni di I/O al secondo e la velocità di trasferimento effettiva dell'unità.
- 12.9 A un bus SCSI può essere connessa più di una unità a disco. In particolare, un bus Fast Wide SCSI-II (si veda Esercizio 12.8) può essere connesso al massimo a 15 unità a disco. Ricordate che questo bus ha un'ampiezza di banda di 20 megabyte al secondo. In ogni momento, solo un pacchetto può essere trasferito sul bus tra la cache interna del disco e l'host. Tuttavia, un disco può muovere il proprio braccio mentre altri dischi stanno trasferendo un pacchetto sul bus. Inoltre, mentre un disco sta trasferendo dati tra i propri piatti magnetici e la propria cache interna, altri dischi trasferiscono un pacchetto sul bus. In riferimento alle velocità di trasferimento che avete calcolato per i vari carichi di lavoro nell'Esercizio 12.8, discutete quanti dischi possono essere effettivamente utilizzati da un bus Fast Wide SCSI-II.
- 12.10 Rimappare blocchi difettosi tramite l'accantonamento o la traslazione di settori può influenzare le prestazioni. Supponete che l'unità dell'Esercizio 12.8 abbia in totale 100 settori difettosi in locazioni casuali e che ogni settore difettoso sia mappato in un settore di riserva posizionato su una traccia differente all'interno dello stesso cilindro. Stimate il numero di operazioni di I/O al secondo e la velocità di trasferimento effettiva per un carico di lavoro ad accesso diretto che consiste in letture da 8 kilobyte, assumendo che la lunghezza di coda sia 1 (e quindi che la scelta dell'algoritmo di scheduling sia ininfluente). Qual è l'effetto di un settore difettoso sulle prestazioni?

- 12.11 Quale effetto si produce in un juke-box di dischi se il numero dei file aperti è superiore rispetto al numero delle unità contenute nel juke-box?
- 12.12 Se un disco fisso magnetico avesse lo stesso costo al gigabyte di un nastro, i nastri diventerebbero obsoleti o sarebbero ancora necessari? Spiegate la vostra risposta.
- 12.13 A volte un nastro è detto mezzo ad accesso sequenziale, mentre un disco magnetico è considerato un mezzo ad accesso diretto. In realtà, l'idoneità di un dispositivo di memorizzazione all'accesso diretto dipende dalla grandezza del trasferimento. Il termine *velocità di trasferimento in streaming* denota la velocità di un trasferimento di dati che è in corso, escluso l'effetto della latenza di accesso. La *velocità effettiva di trasferimento*, invece, è il rapporto dei byte totali per il totale dei secondi, incluso il tempo di overhead come la latenza di accesso. Ipotizzate che, in un computer, la cache di livello 2 abbia una latenza di accesso di 8 nanosecondi e una velocità di trasferimento in streaming di 800 megabyte al secondo, che la memoria principale abbia una latenza di accesso di 60 nanosecondi e una velocità di trasferimento in streaming di 80 megabyte al secondo, che il disco magnetico abbia una latenza di accesso di 15 millisecondi e una velocità di trasferimento in streaming di 5 megabyte al secondo, e che una unità a nastro abbia una latenza di accesso di 60 secondi e una velocità di trasferimento in streaming di 2 megabyte al secondo.
- L'accesso diretto causa la diminuzione della velocità effettiva di trasferimento di un dispositivo, perché non vengono trasferiti dati durante il tempo di accesso. Per il disco descritto, qual è la velocità di trasferimento effettiva se l'accesso medio è seguito da un trasferimento in streaming di (1) 512 byte, (2) 8 kilobyte, (3) 1 megabyte, e (4) 16 megabyte?
  - L'utilizzo di un dispositivo è definito come il rapporto tra la velocità effettiva di trasferimento e la velocità di trasferimento in streaming. Calcolate l'utilizzo dell'unità a disco per ognuna delle quattro grandezze di trasferimento indicate al punto a.
  - Supponete che un utilizzo del 25 per cento o più sia considerato accettabile. Utilizzando i valori di prestazione indicati, calcolate la dimensione minima di trasferimento per un disco che dia un utilizzo accettabile.
  - Completate la frase seguente: Un disco è un dispositivo ad accesso diretto per trasferimenti superiori a \_\_\_\_\_ byte ed è un dispositivo ad accesso sequenziale per trasferimenti inferiori.
  - Calcolate le dimensioni minime di trasferimento che diano utilizzi accettabili per cache, memoria e nastro.
  - Quando un nastro può essere considerato un dispositivo ad accesso diretto e quando invece è un dispositivo ad accesso sequenziale?
- 12.14 Supponete di concordare sul fatto che 1 kilobyte sia 1024 byte, 1 megabyte sia  $1024^2$  byte e che 1 gigabyte sia  $1024^3$  byte. La serie continua con terabyte, petabyte ed esabyte ( $1024^6$ ). Diversi progetti scientifici prevedono di arrivare a registrare e a memorizzare qualche esabyte di dati nei prossimi dieci anni. Per rispondere alle seguenti domande, dovete fare un certo numero di ipotesi ragionevoli: enunciatele esplicitamente.
- Quante unità a disco sarebbero necessarie per contenere 4 esabyte di dati?
  - Quanti dischi magnetici sarebbero necessari per contenere 4 esabyte di dati?
  - Quanti nastri ottici sarebbero necessari per contenere 4 esabyte di dati (si veda l'Esercizio 12.35)?

- d. Quante cartucce olografiche di memorizzazione sarebbero necessarie per contenere 4 esabyte di dati (si veda l'Esercizio 12.34)?
- e. Quanti centimetri cubi di spazio di memorizzazione richiederebbe ciascuna delle opzioni citate?

## Esercizi

**12.15** Eccetto l'FCFS, nessuno tra i criteri di scheduling del disco descritti è veramente *equo* (si può avere un blocco indefinito).

- a. Spiegate perché questa affermazione è vera.
- b. Descrivete una maniera di modificare gli algoritmi come lo SCAN in modo che risultino equi.
- c. Spiegate perché l'equità è un obiettivo importante in un sistema a partizione del tempo.
- d. Date tre o più esempi di circostanze in cui è importante che il sistema operativo sia iniquo nel servire le richieste di I/O.

**12.16** Supponete che un'unità a disco abbia 5000 cilindri numerati da 0 a 4999. L'unità serve attualmente una richiesta relativa al cilindro 143, e la richiesta precedente era relativa al cilindro 125. La coda di richieste inevase, in ordine FIFO, è composta di richieste riguardanti i cilindri

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Assumendo come punto di partenza la posizione attuale della testina, calcolate la distanza totale (in cilindri) che il braccio del disco percorre per soddisfare tutte le richieste inevase usando i seguenti algoritmi di scheduling:

- a. FCFS;
- b. SSTF;
- c. SCAN;
- d. LOOK;
- e. C-SCAN;
- f. C-LOOK.

**12.17** È noto dalla fisica elementare che quando un oggetto è sottoposto a un'accelerazione costante  $a$ , la relazione fra la distanza  $d$  e il tempo  $t$  è data da  $d = 1/2 at^2$ . Supponete che l'unità a disco dell'Esercizio 12.16, durante la ricerca di un cilindro, imprima al braccio del disco un'accelerazione costante per la prima metà del tragitto richiesto dalla ricerca, e imprima invece una decelerazione costante e della stessa intensità dell'accelerazione precedente per la seconda metà del tragitto. Ipotizzate che l'unità possa portare a termine in 1 millisecondo la ricerca di un cilindro adiacente, e in 18 millisecondi una ricerca a tutto raggio lungo i 5000 cilindri.

- a. La distanza di una ricerca è il numero di cilindri attraverso i quali la testina deve passare. Spiegate perché il tempo di ricerca è proporzionale alla radice quadrata della distanza percorsa.
- b. Scrivete il tempo di ricerca in funzione della distanza da percorrere. L'equazione dovrebbe avere la forma  $t = x + y \sqrt{l}$ , dove  $t$  è il tempo in millisecondi ed  $l$  è la distanza da percorrere in cilindri.

- c. Calcolate il tempo totale di ricerca per ogni algoritmo di scheduling dell'Esercizio 12.16 relativamente alla coda di richieste lì descritta. Determinate quale algoritmo sia più veloce, cioè implica un tempo di ricerca totale minore.
- d. L'aumento percentuale di velocità è il tempo risparmiato diviso il tempo originariamente necessario. Calcolate l'aumento percentuale di velocità dell'algoritmo più veloce rispetto all'FCFS.
- 12.18 Supponete che il disco dell'Esercizio 12.17 ruoti alla velocità di 7200 giri al minuto.
- Calcolate la latenza di rotazione media di quest'unità a disco.
  - Dite quale distanza di ricerca è possibile coprire nel tempo calcolato al punto a).
- 12.19 Una ricerca di dati in accelerazione come quella descritta nell'Esercizio 12.17 è tipica delle unità a disco rigido. I floppy disk, invece, come pure molti dischi rigidi prodotti prima della metà degli anni '80, eseguono solitamente delle ricerche a velocità fissa. Supponete che il disco dell'Esercizio 12.17 abbia una ricerca a velocità costante piuttosto che una ricerca ad accelerazione costante, cosicché il tempo di ricerca sia formulabile come  $t = x + yL$ , dove  $t$  è il tempo in millisecondi mentre  $L$  è la distanza di ricerca. Supponete che il tempo per ricercare un cilindro adiacente sia di 1 millisecondo, come prima, mentre il tempo per ricercare ogni cilindro aggiuntivo sia di 0,5 millisecondi.
- Scrivete un'equazione che esprima il tempo di ricerca in funzione della distanza di ricerca.
  - Utilizzando il punto a), calcolate il tempo di ricerca totale per ognuna delle richieste dell'Esercizio 12.16. La risposta ottenuta è uguale a quella dell'Esercizio 12.17(c)?
  - Qual è l'aumento percentuale di velocità dell'algoritmo di scheduling rispetto all'FCFS in questo caso?
- 12.20 Scrivete un programma che simuli gli algoritmi di scheduling del disco trattati nel Paragrafo 12.4.
- 12.21 Confrontate le prestazioni di SCAN e C-SCAN assumendo una distribuzione uniforme delle richieste di I/O. Considerate il tempo di risposta medio (cioè il tempo che intercorre fra l'arrivo di una richiesta e il completamento dell'operazione a essa associata), la variazione del tempo di risposta, e l'effettiva ampiezza di banda. Analizzate come le prestazioni dipendano dalle dimensioni relative del tempo di ricerca e della latenza di rotazione.
- 12.22 Le richieste non sono di solito uniformemente distribuite: ad esempio un cilindro che contiene FAT o inode del file system potrebbe essere visitato più spesso di un cilindro che contiene solo file. Supponete di sapere che il 50 per cento delle richieste sia relativo a un piccolo numero fisso di cilindri.
- Dite se uno fra gli algoritmi illustrati in questo capitolo sia particolarmente adatto a questa circostanza. Motivate la risposta.
  - Proponete un algoritmo di scheduling che offra prestazioni anche migliori sfruttando questo "punto caldo" del disco.
  - Di solito i file system localizzano i blocchi di dati usando una tabella indiretta, come la FAT nell'MS-DOS o gli inode in UNIX. Descrivete uno o più modi di sfruttare queste tabelle per migliorare le prestazioni del disco.

- 12.23 Spiegate se e come sia possibile conseguire, per le richieste di lettura, migliori prestazioni con il sistema RAID a livello 1, piuttosto che con il sistema RAID a livello 0 (con il sezionamento non ridondante dei dati).
- 12.24 Considerate un sistema RAID a livello 5, comprensivo di cinque dischi; nel quinto disco risiedono le parità per gli insiemi di quattro blocchi di quattro dischi. A quanti blocchi si deve accedere per effettuare le operazioni seguenti:
- scrittura di un blocco di dati;
  - scrittura di sette blocchi contigui di dati.
- 12.25 Ponete a confronto il throughput ottenuto attraverso un sistema RAID a livello 5 con quello ottenuto mediante un sistema RAID a livello 1, in merito a:
- operazioni di lettura su blocchi singoli;
  - operazioni di lettura su blocchi multipli contigui.
- 12.26 Paragonate le prestazioni raggiunte da un sistema RAID a livello 5 con quelle di un sistema RAID a livello 1, relativamente alle operazioni di scrittura.
- 12.27 Assumete di avere una configurazione mista, che comprenda dischi strutturati secondo il sistema RAID a livello 1, e altri dischi a livello RAID 5. Supponete che il sistema, per memorizzare un determinato file, sia libero di optare per l'una o l'altra soluzione. Quali file dovrebbero essere memorizzati nei dischi a livello RAID 1 e quali nei dischi a livello RAID 5, allo scopo di ottimizzare le prestazioni?
- 12.28 L'affidabilità di un'unità a disco generalmente si quantifica usando il **tempo medio fra due guasti** (*mean time between failures*, MTBF); sebbene sia chiamata "tempo", questa quantità in effetti si misura in ore di funzionamento d'unità per guasto.
- Dato un gruppo di 1000 unità a disco, ognuna delle quali ha un MTBF di 750.000 ore, dite con quale frequenza avverrà il guasto di un'unità del gruppo, scegliendo fra le seguenti possibilità quella che si adatta meglio alla situazione descritta: una volta ogni mille anni, una volta ogni cento anni, una volta ogni dieci anni, una volta al mese, una volta alla settimana, una volta al giorno, una volta ogni ora, una volta al minuto, una volta al secondo.
  - Le statistiche di mortalità indicano che in media un individuo residente negli Stati Uniti d'America ha circa 1 probabilità su 1000 di morire fra i 20 e i 21 anni d'età. Deducete l'MTBF di un ventenne, e convertite il risultato da ore in anni. Spiegate che cosa dice questo MTBF sulle aspettative di vita di un ventenne.
  - Un produttore asserisce che un certo modello di unità a disco abbia un MTBF di 1 milione di ore. Dite cosa si può concludere circa il numero di anni per cui una di queste unità a disco è coperta dalla garanzia.
- 12.29 Esponete vantaggi e svantaggi relativi all'accantonamento dei settori e alla traslazione dei settori.
- 12.30 Esponete i motivi per cui il sistema operativo potrebbe necessitare di informazioni accurate sulle modalità di memorizzazione dei blocchi sul disco. In termini di miglioramento delle prestazioni, in che modo il sistema operativo può mettere a frutto queste informazioni?
- 12.31 In genere il sistema operativo tratta i dischi rimovibili come un file system condiviso, ma assegna l'unità a nastro a una sola applicazione alla volta. Elencate tre moti-

vi che spieghino questa disparità di trattamento. Descrivete le caratteristiche aggiuntive che il sistema operativo dovrebbe avere per poter gestire un juke-box di nastri come un file system condiviso. Dite se anche le applicazioni che condividono il juke-box devono avere particolari caratteristiche, o se possono usare i file come se risiedessero in un disco; motivate la risposta.

- 12.32 Dite che effetto avrebbe sul costo e sulle prestazioni una densità superficiale dei dati nei nastri pari a quella dei dischi. (Con “densità superficiale” si intende il numero di gigabit per pollice quadrato.)
- 12.33 In questo esercizio userete alcune semplici stime per confrontare il costo e le prestazioni dei due seguenti sistemi: un sistema di memorizzazione con una capacità dell’ordine del terabyte interamente costituito di dischi; e un altro sistema che sfrutta la memoria terziaria. Supponete che ogni disco magnetico abbia una capacità di 10 GB, un costo di 1000 dollari, una velocità di trasferimento di 5 MB al secondo, e una latenza d’accesso media di 15 millisecondi. Supponete inoltre che un archivio di nastri costi 10 dollari per gigabyte, trasferisca 10 MB di dati al secondo e abbia una latenza d’accesso media di 20 secondi. Calcolate il costo totale, la massima velocità totale di trasferimento e il tempo d’attesa medio per il sistema a soli dischi. Se fate qualche ipotesi sul carico di lavoro, descrivetevole e giustificatele. Supponete adesso che il 5 per cento dei dati sia usato di frequente e risieda quindi in dischi, ma che il restante 95 per cento sia memorizzato nell’archivio di nastri; supponete inoltre che il sistema dei dischi gestisca il 95 per cento delle richieste, e l’archivio di nastri il restante 5 per cento. Calcolate il costo totale, la massima velocità totale di trasferimento e il tempo d’attesa medio per questo sistema di gestione gerarchica della memoria.
- 12.34 Immaginate che qualcuno inventi un dispositivo di memorizzazione olografica. Supponete che il suo costo sia di 10.000 dollari, il tempo medio d’accesso di 40 millisecondi e che impieghi cartucce da 100 dollari della dimensione di un CD. Ogni cartuccia può contenere 40.000 immagini, e ciascuna immagine è un quadратo in bianco e nero composto di  $6000 \times 6000$  pixel (ogni pixel memorizza il valore di un bit). Supponete infine che il dispositivo possa leggere o scrivere un’immagine in 1 millisecondo. Analizzate i seguenti aspetti:
- i possibili usi di un tale dispositivo;
  - l’impatto di un tale dispositivo sulle prestazioni dell’I/O di un sistema;
  - l’eventuale possibilità che altri dispositivi di memorizzazione divengano obsoleti.
- 12.35 Supponete che un disco ottico di 5,25 pollici abbia una densità superficiale di dati di 1 gigabit per pollice quadrato. Supponete inoltre che un nastro magnetico abbia una densità superficiale di dati pari a 20 megabit per pollice quadrato, sia largo 1/2 pollice e lungo 1800 piedi. Stimate la capacità di memorizzazione dei due supporti. Ora immaginate che esista un nastro ottico delle stesse dimensioni fisiche del nastro magnetico descritto, ma con la stessa densità superficiale di dati del disco ottico. Calcolate la quantità di dati memorizzabile nel nastro ottico; posto che il nastro magnetico descritto costi 25 dollari, fissate un prezzo di mercato per il nastro ottico.
- 12.36 Valutate i modi in cui un sistema operativo potrebbe mantenere una lista dello spazio libero relativa a un file system residente in un nastro. Supponete che il dispositivo a nastro permetta il solo accodamento, e che usi il simbolo EOT e le operazioni `locate()`, `space()` e `read_position()` descritte nel Paragrafo 12.9.2.1.

## 12.11 Note bibliografiche

Le batterie ridondanti di dischi (RAID) sono presentate da [Patterson et al. 1988] e nella dettagliata rassegna di [Chen et al. 1994]. Le architetture delle unità a disco per elaborazioni ad alte prestazioni sono trattate da [Katz et al. 1989]. Approfondimenti sui sistemi RAID sono offerti da [Wilkes et al. 1996] e [Yu et al. 2000]. [Teorey e Pinkerton 1972] presentano una delle prime analisi comparate degli algoritmi di scheduling del disco; usano un modello di unità a disco con tempo di ricerca lineare rispetto al numero di cilindri attraversati. Per questo tipo di disco l'algoritmo LOOK è una buona scelta per code di richieste innevase lunghe meno di 140 elementi, mentre l'algoritmo C-LOOK lo è per code che superano i 100 elementi. [King 1990] illustra alcuni modi per ridurre il tempo di ricerca, ad esempio con lo spostamento del braccio quando l'unità a disco sarebbe altrimenti inattiva. [Seltzer et al. 1990], [Jacobson e Wilkes 1991] descrivono algoritmi di scheduling incentrati sulla latenza di rotazione, oltre che sui tempi di ricerca. L'ottimizzazione dello scheduling per impiegare al meglio i tempi morti del disco è trattata da [Lumb et al. 2000]. [Worthington et al. 1994] esamina le prestazioni dei dischi, mostrando il trascurabile effetto sulle prestazioni della gestione dei blocchi difettosi. Lo sfruttamento dell'esistenza di un "punto caldo" al fine di ridurre i tempi di ricerca è stato considerato da [Ruemmler e Wilkes 1991], nonché da [Akyurek e Salem 1993]. [Ruemmler e Wilkes 1994] descrivono un accurato modello delle prestazioni di una moderna unità a disco. [Worthington et al. 1995] spiega come determinare caratteristiche del disco di basso livello quali la struttura delle aree all'interno dei cilindri; questo lavoro è ulteriormente sviluppato in [Schindler e Gregory 1999]. Problemi connessi alla gestione della potenza del disco sono esaminati in [Douglis et al. 1994], [Douglis et al. 1995], [Greenawalt 1994], [Golding et al. 1995].

La dimensione dei trasferimenti richiesti e la casualità del carico di lavoro hanno un'influenza considerevole sulle prestazioni dell'unità a disco. [Ousterhout et al. 1985], e Ruemmler e [Wilkes 1993] riportano numerose interessanti caratteristiche dei carichi di lavoro, ad esempio che la maggior parte dei file sono brevi, la maggior parte dei file creati di recente è eliminata assai presto, che il più delle volte i file aperti per lettura sono letti in modo sequenziale nella loro interezza, e che nella maggior parte dei casi le distanze di ricerca sono brevi. [McKusick et al. 1984] descrive il Berkeley Fast File System (FFS), che adotta molte tecniche raffinate aventi lo scopo di ottenere buone prestazioni per parecchi tipi di carichi di lavoro. [McVoy e Kleiman 1991] illustra ulteriori miglioramenti dell'FFS originario. [Quinlan 1991] presenta la realizzazione di un file system per mezzi WORM con cache su dischi magnetici; [Richards 1990] propone un approccio alla memoria terziaria basato sul concetto di file system. [Maher et al. 1994] traccia una panoramica dell'integrazione di file system distribuiti e memoria terziaria.

Il concetto di gestione gerarchica della memorizzazione è studiato da più di un quarto di secolo: un articolo di [Mattson et al. 1970], ad esempio, descrive un metodo matematico per prevedere le prestazioni di un sistema di gestione gerarchica della memorizzazione. [Alt 1993] descrive l'integrazione della memoria rimovibile in un sistema operativo commerciale, mentre [Miller e Katz 1993] descrivono le caratteristiche dell'accesso alla memoria terziaria in un ambiente basato su di un supercalcolatore. [Benjamin 1990] fornisce una panoramica dei problemi connessi alla necessità di un vastissimo spazio di archiviazione per il progetto EOSDIS della NASA. La gestione e l'uso dei dischi collegati in rete, nonché dei dischi programmabili sono tematiche analizzate da [Gibson et al. 1997b], [Gibson et al. 1997a], [Riedel et al. 1998] e [Lee e Thekkath 1996].

La tecnologia della memorizzazione olografica è l'argomento dell'articolo di [Psaltis e Mok 1995]; una raccolta di articoli su quest'argomento, a partire dal 1963, è stata curata da [Sincerbox 1994]. [Asthana e Finkelstein 1995] descrivono diverse tecnologie di memorizzazione emergenti, compresa la memoria olografica, i nastri ottici e tecniche basate sul confinamento dell'elettrone. [Toigo 2000] offre un'approfondita descrizione delle moderne tecnologie dei dischi e di diverse future potenziali tecniche di registrazione dei dati.